MODERN OPERATING SYSTEMS

Fifth Edition

Andrew S. Tanenbaum
Herbert Bos

MODERN OPERATING SYSTEMS

FIFTH EDITION

ANDREW S. TANENBAUM
HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

11

CASE STUDY 2: WINDOWS 11

Windows is a modern operating system that runs on consumer PCs, laptops, tablets, and phones as well as business desktop PCs, workstations, and enterprise servers. Windows is also the operating system used in Microsoft's Xbox gaming system, the Hololens mixed-reality device, and Azure cloud computing infrastructure. The most recent version is Windows 11, released in 2021. In this chapter, we will examine various aspects of Windows, starting with a brief history, then moving on to its architecture. After this, we will look at processes, memory management, caching, I/O, the file system, power management, and finally, security.

11.1 HISTORY OF WINDOWS THROUGH WINDOWS 11

Microsoft's development of the Windows operating system for PC-based computers as well as servers can be divided into four eras: **MS-DOS**, **MS-DOS-based Windows**, **NT-based Windows**, and **Modern Windows**. Technically, each of these systems is substantially different from the others. Each was dominant during different decades in the history of the personal computer. Figure 11-1 shows the dates of the major Microsoft operating system releases for desktop computers. Below we will briefly sketch each of the eras shown in the table.

Year	MS–DOS	MS-DOS based Windows	NT-based Windows	Modern Windows	Notes
1981	1.0				Initial release for IBM PC
1983	2.0				Support for PC/XT
1984	3.0				Support for PC/AT
1990		3.0			Ten million copies in 2 years
1991	5.0				Added memory management
1992		3.1			Ran only on 286 and later
1993			NT 3.1		Supported 32-bit x86, MIPS, Alpha
1995	7.0	95	NT 3.51		MS-DOS embedded in Win 95 NT supports PowerPC
1996			NT 4.0		NT has Windows 95 look and feel
1998		98			
2000	8.0	Me	2000		Win Me was inferior to Win 98 NT supports IA-64
2001			XP		Replaced Win 98. NT supports x64
2006			Vista		Vista could not supplant XP
2009			7		Significantly improved upon Vista
2012				8	First Modern version, supports ARM
2013				8.1	Fixed complaints about Windows 8
2015–2020				10	Unified OS for multiple devices Rapid releases every 6 months Reached 1.3B devices
2021				11	Fresh new UI Broader application support Higher security baseline

Figure 11-1. Major releases in the history of Microsoft operating systems for desktop PCs.

11.1.1 1980s: MS-DOS

In the early 1980s IBM, at the time the biggest and most powerful computer company in the world, was developing a **personal computer** based the Intel 8088 microprocessor. Since the mid-1970s, Microsoft had become the leading provider of the BASIC programming language for 8-bit microcomputers based on the 8080 and Z-80. When IBM approached Microsoft about licensing BASIC for the new IBM PC, Microsoft readily agreed to the deal and suggested that IBM contact Digital Research to license its CP/M operating system since Microsoft was not then in the operating system business. IBM did that, but the president of Digital Research, Gary Kildall, was too busy to meet with IBM. This was probably the worst blunder

in all of business history. Had he licensed CP/M to IBM, Kildall would probably have become the richest man on the planet. Rebuffed by Kildall, IBM came back to Bill Gates, the cofounder of Microsoft, and asked for help again. Within a short time, Microsoft bought a CP/M clone from a local company, Seattle Computer Products, ported it to the IBM PC, and licensed it to IBM. It was then renamed **MS-DOS 1.0** (**MicroSoft Disk Operating System**) and shipped with the first IBM PC in 1981.

MS-DOS was a 16-bit real-mode, single-user, command-line-oriented operating system consisting of 8 KB of memory resident code. Over the next decade, both the PC and MS-DOS continued to evolve, adding more features and capabilities. By 1986, when IBM built the PC/AT based on the Intel 286, MS-DOS had grown to be 36 KB, but it continued to be a command-line-oriented, one-application-at-a-time, operating system.

11.1.2 1990s: MS-DOS-based Windows

Inspired by the graphical user interface of a system developed by Doug Engelbart at Stanford Research Institute and later improved at Xerox PARC, and their commercial progeny, the Apple Lisa and the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface that it called **Windows**. The first two versions of Windows (1985 and 1987) were not very successful, due in part to the limitations of the PC hardware available at the time. In 1990, Microsoft released **Windows 3.0** for the Intel 386 and sold over one million copies in six months.

Windows 3.0 was not a true operating system, but a graphical environment built on top of MS-DOS, which was still in control of the machine and the file system. All programs ran in the same address space and a bug in any one of them could bring the whole system to a frustrating halt.

In August 1995, **Windows 95** was released. It contained many of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming, and introduced 32-bit programming interfaces. However, it still lacked security, and provided poor isolation between applications and the operating system such that a bug in one program can crash the entire system or cause a system-wide hang. Thus, the problems with instability continued, even with the subsequent releases of **Windows 98** and **Windows Me**, where MS-DOS was still there running 16-bit assembly code in the heart of the Windows operating system.

11.1.3 2000s: NT-based Windows

By the end of the 1980s, Microsoft realized that continuing to evolve an operating system with MS-DOS at its center was not the best way to go. PC hardware was continuing to increase in speed and capability and ultimately the PC market would collide with the desktop, workstation, and enterprise-server computing

markets, where UNIX was the dominant operating system. Microsoft was also concerned that the Intel microprocessor family might not continue to be competitive, as it was already being challenged by RISC architectures. To address these issues, Microsoft recruited a group of engineers from DEC (Digital Equipment Corporation) led by Dave Cutler, one of the key designers of DEC's VMS operating system (among others). Cutler was chartered to develop a brand-new 32-bit operating system that was intended to implement **OS/2**, the operating system API that Microsoft was jointly developing with IBM at the time. The original design documents by Cutler's team called the system **NT OS/2**.

Cutler's system was called **NT** (**New Technology**) (and also because the original target processor was the new Intel 860, code-named the N10). NT was designed to be portable across different processors and emphasized security and reliability, as well as compatibility with the MS-DOS-based versions of Windows. Cutler's background at DEC shows in various places, with there being more than a passing similarity between the design of NT and that of VMS and other operating systems designed by Cutler, shown in Fig. 11-2.

Year	DEC operating system	Characteristics
1973	RSX-11M	16-bit, multiuser, real-time, swapping
1978	VAX/VMS	32-bit, virtual memory
1987	VAXELAN	Real-time
1988	PRISM/Mica	Canceled in favor of MIPS/Ultrix

Figure 11-2. DEC operating systems developed by Dave Cutler.

Programmers familiar only with UNIX find the architecture of NT to be quite different. This is not just because of the influence of VMS, but also because of the differences in the computer systems that were common at the time of design. UNIX was first designed in the 1970s for single-processor, 16-bit, tiny-memory, swapping systems where the process was the unit of concurrency and composition, and fork/exec were inexpensive operations (since swapping systems frequently copy processes to disk anyway). NT was designed in the early 1990s, when multiprocessor, 32-bit, multimegabyte, virtual memory systems were common. In NT, threads are the units of concurrency, dynamic libraries are the units of composition, and fork/exec are implemented by a single operation to create a new process *and* run another program without first making a copy.

The first version of NT-based Windows (Windows NT 3.1) was released in 1993. It was called 3.1 to correspond with the then-current consumer Windows 3.1. The joint project with IBM had foundered, so though the OS/2 interfaces were still supported, the primary interfaces were 32-bit extensions of the Windows APIs, called **Win32**. Between the time NT was started and first shipped, Windows 3.0 had been released and had become extremely successful commercially. It too was able to run Win32 programs, but using the *Win32s* compatibility library.

Like the first version of MS-DOS-based Windows, NT-based Windows was not initially successful. NT required more memory, there were few 32-bit applications available, and incompatibilities with device drivers and applications caused many customers to stick with MS-DOS-based Windows which Microsoft was still improving, releasing Windows 95 in 1995. Windows 95 provided native 32-bit programming interfaces like NT, but better compatibility with existing 16-bit software and applications. Not surprisingly, NT's early success was in the server market, competing with VMS and NetWare.

NT did meet its portability goals, with additional releases in 1994 and 1995 adding support for (little-endian) MIPS and PowerPC architectures. The first major upgrade to NT came with **Windows NT 4.0** in 1996. This system had the power, security, and reliability of NT, but also sported the same user interface as the by-then very popular Windows 95.

Figure 11-3 shows the relationship of the Win32 API to Windows. Having a common API across both the MS-DOS-based and NT-based Windows was important to the success of NT.

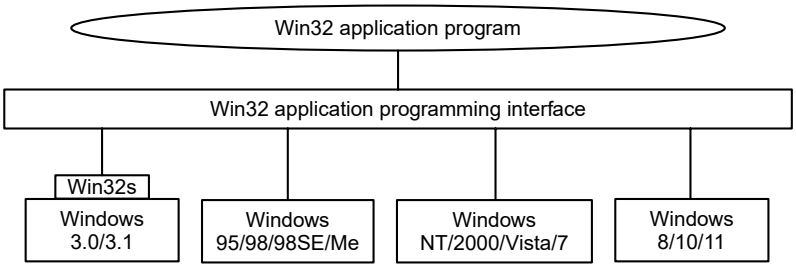

Figure 11-3. The Win32 API allows programs to run on almost all versions of Windows.

This compatibility made it much easier for users to migrate from Windows 95 to NT, and the operating system became a strong player in the high-end desktop market as well as servers. However, customers were not as willing to adopt other processor architectures, and of the four architectures Windows NT 4.0 supported in 1996, only the x86 (i.e., Pentium family) was still actively supported by the time of the next major release, **Windows 2000**.

Windows 2000 represented a significant evolution for NT. The key technologies added were plug-and-play (for consumers who installed a new PCI card, eliminating the need to fiddle with jumpers), network directory services (for enterprise customers), improved power management (for notebook computers), and an improved GUI (for everyone).

The technical success of Windows 2000 led Microsoft to push toward the deprecation of Windows 98 by enhancing the application and device compatibility of the next NT release, **Windows XP**. Windows XP included a friendlier new look-

and-feel to the graphical interface, bolstering Microsoft's strategy of hooking consumers and reaping the benefit as they pressured their employers to adopt systems with which they were already familiar. The strategy was overwhelmingly successful, with Windows XP being installed on hundreds of millions of PCs over its first few years, allowing Microsoft to achieve its goal of effectively ending the era of MS-DOS-based Windows.

Microsoft followed up Windows XP by embarking on an ambitious release to kindle renewed excitement among PC consumers. The result, **Windows Vista**, was completed in late 2006, more than five years after Windows XP shipped. Windows Vista boasted yet another redesign of the graphical interface, and new OS features under the covers. There were many changes in customer-visible experiences and capabilities. The technologies under the covers of the system improved substantially, with much clean-up of the code and many improvements in performance, scalability, and reliability. The server version of Vista (Windows Server 2008) was delivered about a year after the consumer version. It shares, with Vista, the same core system components, such as the kernel, drivers, and low-level libraries and programs.

The human story of the early development of NT is related in the book *Showstopper* (Zachary, 1994). The book tells a lot about the key people involved and the difficulties of undertaking such an ambitious software development project.

11.1.4 Windows Vista

The release of Windows Vista culminated Microsoft's most extensive operating system project to date. The initial plans were so ambitious that a couple of years into its development Vista had to be restarted with a smaller scope. Plans to rely heavily on Microsoft's type-safe, garbage-collected .NET language C# were shelved, as were some significant features such as the WinFS unified storage system for searching and organizing data from many different sources. The size of the full operating system is staggering. The original NT release of 3 million lines of C/C++ had grown to 16 million in NT 4, 30 million in 2000, and 50 million in XP. By Windows Vista, the line count had grown to 70 million and has continued to grow every since.

Much of the size is due to Microsoft's emphasis on adding many new features to its products in every release. In the main *system32* directory, there are 1600 **DLLs** (**Dynamic Link Libraries**) and 400 **EXEs** (**Executables**), and that does not include the other directories containing the myriad of applets included with the operating system that allow users to surf the Web, play music and video, send email, scan documents, organize photos, and even make movies. Because Microsoft wants customers to switch to new versions, it maintains compatibility by generally keeping all the features, APIs, *applets* (small applications), etc., from the previous version. Few things ever get deleted. The result is that Windows was growing dramatically larger from release to release. Windows' distribution media

had moved from floppy, to CD, and with Windows Vista, to DVD. Technology had been keeping up, however, and faster processors and larger memories made it possible for computers to get faster despite all this bloat.

Unfortunately for Microsoft, Windows Vista was released at a time when customers were becoming enthralled with inexpensive computers, such as low-end notebooks and **netbook** computers. These machines used slower processors to save cost and battery life, and in their earlier generations limited memory sizes. At the same time, processor performance ceased to improve at the same rate it had previously due to the difficulties in dissipating the heat created by ever-increasing clock speeds. Moore's law continued to hold, but the additional transistors were going into new features and multiple processors rather than improvements in single-processor performance. The substantial growth in Windows Vista meant that it performed poorly on these computers relative to Windows XP, and the release was never widely accepted.

The issues with Windows Vista were addressed in the subsequent release, **Windows 7**. Microsoft invested heavily in testing and performance automation, new telemetry technology, and extensively strengthened the teams charged with improving performance, reliability, and security. Though Windows 7 had relatively few functional changes compared to Windows Vista, it was better engineered and more efficient. Windows 7 quickly supplanted Vista and ultimately Windows XP to be the most popular version of Windows within a few years after its release.

11.1.5 Windows 8

By the time Windows 7 finally shipped, the computing industry once again began to change dramatically. The success of the Apple iPhone as a portable computing device, and the advent of the Apple iPad, had heralded a sea-change which led to the dominance of lower-cost Android phones and tablets, much as Microsoft had dominated the desktop in the first three decades of personal computing. Small, portable, yet powerful devices and ubiquitous fast networks were creating a world where mobile computing and network-based services were becoming the dominant paradigm. The old world of desktop and notebook computers was replaced by machines with small screens that ran applications readily downloadable from dedicated *app stores*. These applications were not the traditional variety, like word processing, spreadsheets, and connecting to corporate servers. Instead, they provided access to services such as Web search, social networking, games, Wikipedia, streaming music and video, shopping, and personal navigation. The business models for computing were also changing, with user data collection and advertising opportunities becoming the largest economic force behind computing.

Microsoft began a process to redesign itself as a *devices and services* company in order to better compete with Google and Apple. It needed an operating system it could deploy across a wide spectrum of devices: phones, tablets, game consoles, laptops, desktops, servers, and the cloud. Windows thus underwent an even bigger

evolution than with Windows Vista, resulting in **Windows 8**. However, this time Microsoft applied the lessons from Windows 7 to create a well-engineered, performant product with less bloat.

Windows 8 built on the modular **OneCore** operating system composition approach to produce a small operating system core that could be extended onto different devices. The goal was for each of the operating systems for specific devices to be built by extending this core with new user interfaces and features, yet provide as common an experience for users as possible. This approach was successfully applied to Windows Phone 8, which shares most of the core binaries with desktop and server Windows. Support of phones and tablets by Windows required support for the popular ARM architecture (arm32), as well as new Intel processors targeting those devices. What makes Windows 8 part of the Modern Windows era are the fundamental changes in the programming models, as we will examine in the next section.

Windows 8 was not received to universal acclaim. In particular, the lack of the Start Button on the taskbar (and its associated menu) was viewed by many users as a huge mistake. Others objected to using a tablet-like, touch-first interface on a desktop machine with a large monitor and a mouse. Over the following two years, Microsoft responded to this and other criticisms by releasing an update in 2013 called **Windows 8.1** which itself was refreshed again in the spring of 2014. This version made significant progress toward fixing these problems while at the same time introducing a host of new features, such as better cloud integration, improved functionality for apps bundled with Windows, and numerous performance improvements which actually *lowered* the minimum system requirements for Windows for the first time ever.

11.1.6 Windows 10

Windows 10 was the culmination of Microsoft's multi-device OS vision which started with Windows 8. It provided a single, unified operating system and application development platform for desktop/laptop computers, tablets, smartphones, all-in-one devices, Xbox, Hololens, and the Surface Hub collaboration device. Apps written for the new **UWP** (**Universal Windows Platform**) could target multiple device families with the same underlying code and be distributed from the Windows Store. Up until that time, developer interest in Windows 8's modern application platform was low and Microsoft wanted to shift developer mindshare from the competing iOS and Android platforms to Windows.

Internally, teams working on Windows and Windows Phone were merged into a single organization and produced a *converged* OS which unified the application development platform under UWP. Windows Mobile 10 was the mobile *edition* of Windows 10 targeted at smartphones and tablets, built out of a single code base. OneCore-based OS composition allowed each Windows edition to share a common core, but provided its own unique user-interface and features.

Ultimately, Windows 10 was the most successful release of Windows ever with over 1.3 billion devices running it, as of fall 2021. Ironically, this success cannot be attributed to developer enthusiasm over UWP or the popularity of Windows 10 Mobile because neither really happened. Windows 10 Mobile was discontinued in 2017, and while UWP is alive and well, it is not nearly the most popular platform for developing Windows applications.

Windows 10 provided a familiar user interface and numerous usability improvements which worked well on desktop/laptop computers as well as tablets and "convertible" devices. A public beta program called the **Windows Insider Program** was started early in Windows 10 development cycle to regularly share preview builds of the operating system with "Windows Insiders." The program was very successful with several hundred thousand enthusiasts testing and evaluating weekly builds. This arrangement allowed Windows developers access to end-user feedback and telemetry which helped improve the product with every 6-month release.

Windows 10 leveraged virtual machine technology to significantly improve security. **Biometric** and **multi-factor authentication** simplified the user logon experience and made it safer. **Virtualization-based security** helped protect sensitive information from even kernel-mode attacks while providing an isolated runtime environment for certain applications. Taking advantage of the latest hardware features from chip manufacturers (including support for the 64-bit ARM architecture complete with transparent emulation of x86 applications), Windows 10 improved performance and battery life with new devices while keeping its minimum hardware requirements constant and allowed Windows 7 users to upgrade as official support for the OS ended.

11.1.7 Windows 11

Windows 11 is the most recent version of Windows, publicly made available on October 5, 2021. It brings numerous usability improvements such as a fresh, rejuvenated UI, more efficient window management and multitasking capabilities especially on bigger and multiple monitor configurations. Following the remote and hybrid work/learning trend, it provides deeper integration with the Teams collaboration software as well as Microsoft 365 cloud productivity suite.

While the user interface updates are the most talked-about features of any new OS, and most relevant to the typical user, the latest Windows has plenty of advances under the hood. In keeping up with hardware developments, Windows 11 adds various performance, power, and scalability optimizations to take better advantage of increased number of processor cores, with support up to 2048 logical processors and more than 64 processors per socket. Perhaps more importantly, Windows 11 breaks new ground in application compatibility: emulation of x64 applications is now supported on 64-bit ARM devices and it is even possible to run

Android applications. The most significant advance Windows 11 brings, however, is the much higher security baseline. While many of its security features were present on earlier releases, Windows 11 sets its minimum hardware requirements such that all of these security protections (such as Secure Boot, Device Guard, Application Guard, and kernel-mode Control Flow Guard) can be used. All of them are enabled by default. The higher security baseline, along with new security features such as kernel-mode **Hardware Stack Protection**, makes Windows 11 the most secure version of Windows ever.

In the rest of this chapter, we will describe how Windows 11 works, how it is structured, and what these security features do. Although we will use the generic name of "Windows," all subsequent sections in this chapter refer to Windows 11.

11.2 PROGRAMMING WINDOWS

It is now time to start our technical study of Windows. Before getting into the details of the internal structure, however, we will take a look at the native NT API for system calls, the Win32 programming subsystem introduced as part of NT-based Windows, and the WinRT programming environment first introduced with Windows 8.

Figure 11-4 shows the layers of the Windows operating system. Beneath the GUI layers of Windows are the programming interfaces that applications build on. As in most operating systems, these consist largely of code libraries (DLLs) to which programs dynamically link for access to operating system features. Some of these libraries are **client libraries** which use **RPC**s (**Remote Procedure Calls**) to communicate with operating system services running in separate processes.

The core of the NT operating system is the **NTOS** kernel-mode program (*ntoskrnl.exe*), which provides the traditional system-call interfaces upon which the rest of the operating system is built. In Windows, only programmers at Microsoft write to the native system-call layer. The published user-mode interfaces all belong to operating system personalities that are implemented using **subsystems** that run on top of the NTOS layers.

Originally, NT supported three personalities: OS/2, POSIX, and Win32. OS/2 was discarded in Windows XP. Support for POSIX was finally removed in Windows 8.1. Today all Windows applications are written using APIs that are built on top of the Win32 subsystem, such as the **WinRT API** used for building Universal Windows Platform applications or the cross-platform CoreFX API in the .NET (Core) software framework. Furthermore, through the *win32metadata* GitHub project, Microsoft publishes a description of the entire Win32 API surface in a standard format (called ECMA-335) such that **language projections** can be built to allow the API to be called from arbitrary languages like C# and Rust. This allows applications written in languages other than C/C++ to work on Windows.

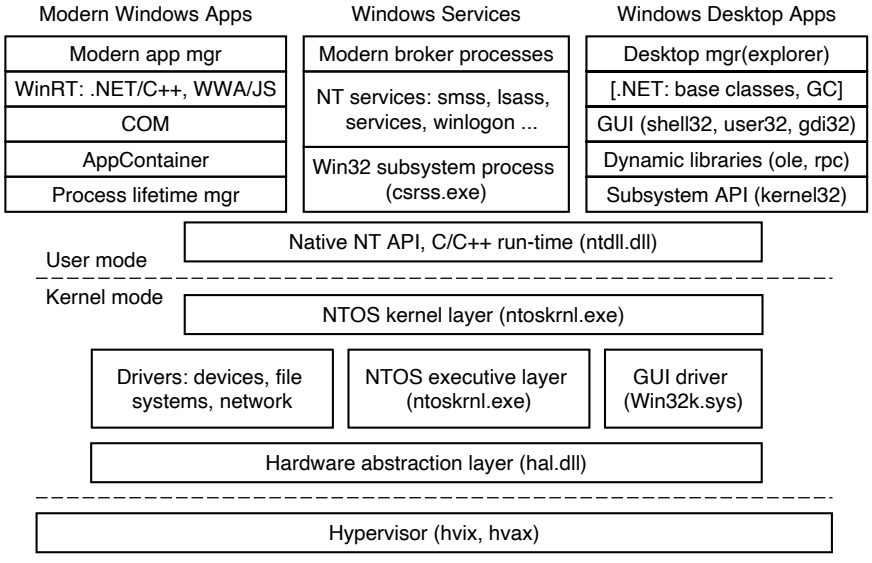

Figure 11-4. The programming layers in Modern Windows.

11.2.1 Universal Windows Platform

The Universal Windows Platform, introduced with Windows 10 based on the modern application platform in Windows 8, represented the first significant change to the application model for Windows programs since Win32. The WinRT API as well as a significant subset of the Win32 API surface is available to UWP applications, allowing them to target multiple device families with the same underlying code while taking advantage of unique device capabilities via device family-specific extensions. UWP is the only supported platform for apps on the Xbox gaming console, the HoloLens mixed reality device, and the Surface Hub collaboration device.

WinRT APIs are carefully curated to avoid various "sharp edges" of the Win32 API to provide more consistent security, user privacy and app isolation properties. They have projections into various languages such as C++, C#, and even JavaScript allowing developer flexibility. In early Windows 10 releases, the subset of the Win32 API available to UWP apps was too limited. For example, various threading or virtual memory APIs were out-of-bounds. This created friction for developers and made it more difficult to port software libraries and frameworks to support UWP. Over time, more and more Win32 APIs were made available to UWP applications.

In addition to the API differences, the *application model* for UWP apps is different from traditional Win32 programs in several ways.

First, unlike traditional Win32 processes, the processes running UWP applications have their lifetimes managed by the operating system. When a user switches away from an application, the system gives it a couple of seconds to save its state and then ceases to give it further processor resources until the user switches back to the application. If the system runs low on resources, the operating system may terminate the application's processes without the application ever running again. When the user switches back to the application at some time in the future, it will be restarted by the operating system. Applications that need to run tasks in the background must specifically arrange to do so using a new set of WinRT APIs. Background activity is carefully managed by the system to improve battery life and prevent interference with the foreground application the user is currently using. These changes were made to make Windows function better on mobile devices, where users frequently switch from app to app and back quickly and often.

Second, in the Win32 desktop world, applications are deployed by running an installer that is part of the application. This scheme leaves clean up in the hands of the application and frequently results in leftover files or settings when the application is uninstalled, leading to "winrot." UWP applications come in an **MSIX package** which is basically a zip file containing application binaries along with a *manifest* that declares the components of the application and how they should integrate with the system. That way, the operating system can install and uninstall the application cleanly and reliably. Typically, UWP applications are distributed and deployed via the Microsoft Store, similar to the model on iOS and Android devices.

Finally, when a modern application is running, it always executes in a *sandbox* called an **AppContainer**. Sandboxing process execution is a security technique for isolating less trusted code so that it cannot freely tamper with the system or user data. The Windows AppContainer treats each application as a distinct user, and uses Windows security facilities to keep the application from accessing arbitrary system resources. When an application does need access to a system resource, there are WinRT APIs that communicate to **broker processes** which do have access to more of the system, such as a user's files.

Despite its many advantages, UWP did not gain widespread traction with developers. This is primarily because the cost of switching existing apps to UWP outweighed the benefits of getting access to the WinRT API and being able to run on multiple Windows device families. Restricted access to the Win32 API and the restructuring necessary to work with the UWP application model meant that apps essentially needed to be rewritten.

To remedy these drawbacks and "bridge the gap" between Win32 desktop app development and UWP, Microsoft is on a path to unify these application models with the Windows App **SDK** (**Software Development Kit**) previously called **Project Reunion**. Windows App SDK is a set of open-source libraries on GitHub, providing a modern, uniform API surface available to all Windows applications. It allows developers to add new functionality previously only exposed to UWP,

without having to rewrite their applications from scratch. Windows App SDK contains the following major components:

1. WinUI, a XAML-based modern UI framework.
2. C++, Rust, C# language projections to expose WinRT API to all apps.
3. MSIX SDK, which allows any application to be packaged and deployed via MSIX.

We briefly covered some of the programming frameworks that developers can use to develop applications for Windows. While these applications built on different frameworks rely on different libraries at higher levels, they ultimately depend on the Win32 subsystem and the native NT API. We will study those shortly.

11.2.2 Windows Subsystems

As shown in Fig. 11-5, NT subsystems are constructed out of four components: a subsystem process, a set of libraries, hooks in CreateProcess, and support in the kernel. A subsystem process is really just a service. The only special property is that it is started by the *smss.exe* (session manager) program—the initial user-mode program started by NT—in response to a request from **CreateProcess** in Win32 or the corresponding API in a different subsystem. Although Win32 is the only remaining subsystem supported, Windows still maintains the subsystem model, including the *csrss.exe* Win32 subsystem process.

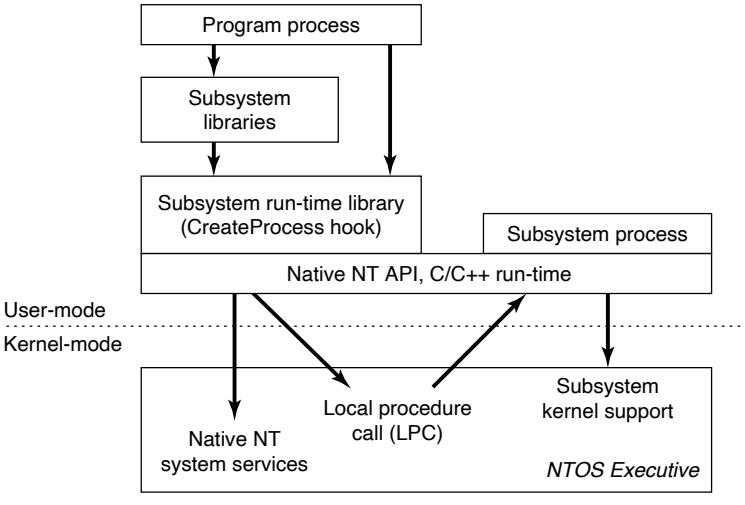

Figure 11-5. The components used to build NT subsystems.

The set of libraries both implements higher-level operating-system functions specific to the subsystem and also contains the stub routines which communicate

between processes using the subsystem (shown on the left) and the subsystem process itself (shown on the right). Calls to the subsystem process normally take place using the kernel-mode **LPC** (**Local Procedure Call**) facilities, which implement cross-process procedure calls.

The hook in Win32 CreateProcess detects which subsystem each program requires by looking at the binary image. It then asks *smss.exe* to start the subsystem process (if it is not already running). The subsystem process then takes over responsibility for loading the program.

The NT kernel was designed to have a lot of general-purpose facilities that can be used for writing operating-system-specific subsystems. But there is also special code that must be added to correctly implement each subsystem. As examples, the native NtCreateProcess system call implements process duplication in support of POSIX fork system call, and the kernel implements a particular kind of string table for Win32 (called *atoms*) which allows read-only strings to be efficiently shared across processes.

The subsystem processes are native NT programs which use the native system calls provided by the NT kernel and core services, such as *smss.exe* and *lsass.exe* (local security administration). The native system calls include cross-process facilities to manage virtual addresses, threads, handles, and exceptions in the processes created to run programs written to use a particular subsystem.

11.2.3 The Native NT Application Programming Interface

Like all other operating systems, Windows has a set of system calls it can perform. In Windows, these are implemented in the NTOS executive layer that runs in kernel mode. Microsoft has published very few of the details of these native system calls. They are used internally by lower-level programs that ship as part of the operating system (mainly services and the subsystems), as well as kernel-mode device drivers. The native NT system calls do not really change very much from release to release, but Microsoft chose not to make them public so that applications written for Windows would be based on Win32 and thus more likely to work with both the MS-DOS-based and NT-based Windows systems, since the Win32 API is common to both.

Most of the native NT system calls operate on kernel-mode objects of one kind or another, including files, processes, threads, pipes, semaphores, and so on. Figure 11-6 gives a list of some of the common categories of kernel-mode objects supported by the kernel in Windows. Later, when we discuss the object manager, we will provide further details on the specific object types.

Sometimes use of the term *object* regarding the data structures manipulated by the operating system can be confusing because it is mistaken for *object-oriented*. Windows operating system objects do provide data hiding and abstraction, but they lack some of the most basic properties of object-oriented systems such as inheritance and polymorphism, so Windows is not object-oriented in the technical sense.

Object category	Examples
Synchronization	Semaphores, mutexes, events, IPC ports, I/O completion queues
I/O	Files, devices, drivers, timers
Program	Jobs, processes, threads, sections, tokens
Win32 GUI	Desktops, application callbacks

Figure 11-6. Common categories of kernel-mode object types.

In the native NT API, calls are available to create new kernel-mode objects or access existing ones. Every call creating or opening an object returns a **handle** to the caller. A handle in Windows is somewhat analogous to a file descriptor in UNIX, except that it can be used for more types of objects than just files. The handle can subsequently be used to perform operations on the object. Handles are specific to the process that created them. In general, handles cannot be passed directly to another process and used to refer to the same object. However, under certain circumstances, it is possible to duplicate a handle into the handle table of other processes in a protected way, allowing processes to share access to objects—even if the objects are not accessible in the namespace. The process duplicating each handle must itself have handles for both the source and target process.

Every object has a **security descriptor** associated with it, telling in detail who may and may not perform what kinds of operations on the object based on the access requested. When handles are duplicated between processes, new access restrictions can be added that are specific to the duplicated handle. Thus, a process can duplicate a read-write handle and turn it into a read-only version in the target process.

Figure 11-7 shows a sampling of the native APIs, all of which use explicit handles to manipulate kernel-mode objects such as processes, threads, IPC ports, and **sections** (which are used to describe memory objects that can be mapped into address spaces). NtCreateProcess returns a handle to a newly created process object, representing an executing instance of the program represented by the SectionHandle. DebugPortHandle is used to communicate with a debugger when giving it control of the process after an exception (e.g., dividing by zero or accessing invalid memory). ExceptPortHandle is used to communicate with a subsystem process when errors occur and are not handled by an attached debugger.

NtCreateThread takes ProcHandle because it can create a thread in any process for which the calling process has a handle (with sufficient access rights). In a similar vein, NtAllocateVirtualMemory, NtMapViewOfSection, NtReadVirtualMemory, and NtWriteVirtualMemory allow one process not only to operate on its own address space, but also to allocate virtual addresses, map sections, and read or write virtual memory in other processes. NtCreateFile is the native API call for creating a new file or opening an existing one. NtDuplicateObject is the API call for duplicating handles from one process to another.

NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)
NtReadVirtualMemory(ProcHandle, Addr, Size, ...)
NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)
NtCreateFile(&FileHandle, FileNameDescriptor, Access, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)

Figure 11-7. Examples of native NT API calls that use handles to manipulate objects across process boundaries.

Kernel-mode objects are, of course, not unique to Windows. UNIX systems also support a variety of kernel-mode objects, such as files, network sockets, pipes, devices, processes, and interprocess communication (IPC) facilities, including shared memory, message ports, semaphores, and I/O devices. In UNIX, there are a variety of ways of naming and accessing objects, such as file descriptors, process IDs, and integer IDs for SystemV IPC objects, and i-nodes for devices. The implementation of each class of UNIX objects is specific to the class. Files and sockets use different facilities than the SystemV IPC mechanisms or processes or devices.

Kernel objects in Windows use a uniform facility based on handles and names in the NT namespace to reference kernel objects, along with a unified implementation in a centralized **object manager**. Handles are per-process but, as described above, can be duplicated into another process. The object manager allows objects to be given names when they are created, and then opened by name to get handles for the objects.

The object manager uses **Unicode** (wide characters) to represent names in the **NT namespace**. Unlike UNIX, NT does not generally distinguish between upper- and lowercase (it is *case preserving* but *case insensitive*). The NT namespace is a hierarchical tree-structured collection of directories, symbolic links, and objects.

The object manager also provides facilities for synchronization, security, and object lifetime management. Whether the general facilities provided by the object manager are made available to users of any particular object is up to the executive components, as they provide the native APIs that manipulate each object type.

It is not only applications that use objects managed by the object manager. The operating system itself can also create and use objects—and does so heavily. Most of these objects are created to allow one component of the system to store some information for a substantial period of time or to pass some data structure to another component, and yet benefit from the naming and lifetime support of the object manager. For example, when a device is discovered, one or more **device objects** are created to represent the device and to logically describe how the device is connected to the rest of the system. To control the device, a device driver is loaded,

and a **driver object** is created holding its properties and providing pointers to the functions it implements for processing the I/O requests. Within the operating system, the driver is then referred to by using its object. The driver can also be accessed directly by name rather than indirectly through the devices it controls (e.g., to set parameters governing its operation from user mode).

Unlike UNIX, which places the root of its namespace in the file system, the root of the NT namespace is maintained in the kernel's virtual memory. This means that NT must recreate its top-level namespace every time the system boots. Using kernel virtual memory allows NT to store information in the namespace without first having to start the file system running. It also makes it much easier for NT to add new types of kernel-mode objects to the system because the formats of the file systems themselves do not have to be modified for each new object type.

A named object can be marked *permanent*, meaning that it continues to exist until explicitly deleted or the system reboots, even if no process currently has a handle for the object. Such objects can even extend the NT namespace by providing *parse* routines that allow the objects to function somewhat like mount points in UNIX. File systems and the registry use this facility to mount volumes and hives (parts of the registry) onto the NT namespace. Accessing the device object for a volume gives access to the raw volume, but the device object also represents an implicit mount of the volume into the NT namespace. The individual files on a volume can be accessed by concatenating the volume-relative file name onto the end of the name of the device object for that volume.

Permanent names are also used to represent synchronization objects and shared memory, so that they can be shared by processes without being continually recreated as processes stop and start. Device objects and often driver objects are given permanent names, giving them some of the persistence properties of the special i-nodes kept in the */dev* directory of UNIX.

We will describe many more of the features in the native NT API in the next section, where we discuss the Win32 APIs that provide wrappers around the NT system calls.

11.2.4 The Win32 Application Programming Interface

The Win32 function calls are collectively called the **Win32 API**. These interfaces are publicly disclosed and fully documented. They are implemented as library procedures that either wrap the native NT system calls used to get the work done or, in some cases, do the work right in user mode. Though the native NT APIs are not published, most of the functionality they provide is accessible through the Win32 API. The existing Win32 API calls do not change with new releases of Windows to maintain application compatibility, though many new functions are added to the API.

Figure 11-8 shows various low-level Win32 API calls and the native NT API calls that they wrap. What is interesting about the figure is how uninteresting the

mapping is. Most low-level Win32 functions have native NT equivalents, which is not surprising as Win32 was designed with NT in mind. In many cases, the Win32 layer must manipulate the Win32 parameters to map them onto NT, for example, canonicalizing path names and mapping onto the appropriate NT path names, including special MS-DOS device names (like *LPT:*). The Win32 APIs for creating processes and threads also must notify the Win32 subsystem process, *csrss.exe*, that there are new processes and threads for it to supervise, as we will describe in Sec. 11.4. It's worth noting that while the Win32 API is built on the NT API, not all of the NT API is exposed through Win32.

Win32 call	Native NT API call
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Figure 11-8. Examples of Win32 API calls and the native NT API calls that they wrap.

Some Win32 calls take path names, whereas the equivalent NT calls use handles. So the wrapper routines have to open the files, call NT, and then close the handle at the end. The wrappers also translate the Win32 APIs from ANSI to Unicode. The Win32 functions shown in Fig. 11-8 that use strings as parameters are actually two APIs, for example, **CreateProcessW** and **CreateProcessA**. The strings passed to the latter API must be translated to Unicode before calling the underlying NT API, since NT works only with Unicode.

Since few changes are made to the existing Win32 interfaces in each release of Windows, in theory, the binary programs that ran correctly on any previous release will continue to run correctly on a new release. In practice, there are often many compatibility problems with new releases. Windows is so complex that a few seemingly inconsequential changes can cause application failures. And applications themselves are often to blame, since they frequently make explicit checks for specific operating system versions or fall victim to their own latent bugs that are exposed when they run on a new release. Nevertheless, Microsoft makes an effort in every release to test a wide variety of applications to find incompatibilities and either correct them or provide application-specific workarounds.

Windows supports two special execution environments both called **WOW** (**Windows-on-Windows**). WoW32 is used on 32-bit x86 systems to run 16-bit Windows 3.x applications by mapping the system calls and parameters between the 16-bit and 32-bit worlds. The last version of Windows to include the WoW32 execution environment was Windows 10. Since Windows 11 requires a 64-bit processor and those processors cannot run 16-bit code, WoW32 is no longer supported. WoW64, which allows 32-bit applications to run on 64-bit systems, continues to be supported on Windows 11. In fact, starting with Windows 10, WoW64 is enhanced to enable running 32-bit x86 applications on arm64 hardware via instruction emulation. Windows 11 further extends emulation capabilities to run 64-bit x64 applications on arm64. Section 11.4.4 describes the WoW64 and emulation infrastructure in more detail.

The Windows API philosophy is very different from the UNIX philosophy. In the latter, the operating system functions are simple, with few parameters and few places where there are multiple ways to perform the same operation. Win32 provides very comprehensive interfaces with many parameters, often with three or four ways of doing the same thing, and mixing together low-level and high-level functions, like CreateFile and CopyFile.

This means Win32 provides a very rich set of interfaces, but it also introduces much complexity due to the poor layering of a system that intermixes both high-level and low-level functions in the same API. For our study of operating systems, only the low-level functions of the Win32 API that wrap the native NT API are relevant, so those are what we will focus on.

Win32 has calls for creating and managing both processes and threads. There are also many calls that relate to interprocess communication, such as creating, destroying, and using mutexes, semaphores, events, communication ports, and other IPC objects.

Although much of the memory-management system is invisible to programmers, one important feature is visible: namely the ability of a process to map a file onto a region of its virtual memory. This allows threads running in a process the ability to read and write parts of the file using pointers without having to explicitly perform read and write operations to transfer data between the disk and memory. With memory-mapped files the memory-management system itself performs the I/Os as needed (demand paging).

Windows implements memory-mapped files using a combination of three facilities. First it provides interfaces which allow processes to manage their own virtual address space, including reserving ranges of addresses for later use. Second, Win32 supports an abstraction called a **file mapping**, which is used to represent addressable objects like files (a file mapping is called a *section* in the NT layer which is a better name because section objects do not have to represent files). Most often, file mappings are created using a file handle to refer to memory backed by files, but they can also be created to refer to memory backed by the system pagefile by using a NULL file handle.

The third facility maps *views* of file mappings into a process' address space. Win32 allows only a view to be created for the current process, but the underlying NT facility is more general, allowing views to be created for any process for which you have a handle with the appropriate permissions. Separating the creation of a file mapping from the operation of mapping the file into the address space is a different approach than used in the mmap function in UNIX.

In Windows, the file mappings are kernel-mode objects represented by a handle. Like most handles, file mappings can be duplicated into other processes. Each of these processes can map the file mapping into its own address space as it sees fit. This is useful for sharing memory between processes without having to create files for sharing. At the NT layer, file mappings (sections) can also be made persistent in the NT namespace and accessed by name.

An important area for many programs is file I/O. In the basic Win32 view, a file is just a linear sequence of bytes. Win32 provides over 70 calls for creating and destroying files and directories, opening and closing files, reading and writing them, requesting and setting file attributes, locking ranges of bytes, and many more fundamental operations on both the organization of the file system and access to individual files.

There are also various advanced facilities for managing data in files. In addition to the primary data stream, files stored on the NTFS file system can have additional data streams. Files (and even entire volumes) can be encrypted. Files can be compressed and/or represented as a sparse stream of bytes where missing regions of data in the middle occupy no storage on disk. File-system volumes can be organized out of multiple separate disk partitions using different levels of RAID storage. Modifications to files or directory subtrees can be detected through a notification mechanism or by reading the **journal** that NTFS maintains for each volume.

Each file-system volume is implicitly mounted in the NT namespace, according to the name given to the volume, so a file *\foo\bar* might be named, for example, *\Device\HarddiskVolume1\foo\bar*. Internal to each NTFS volume, mount points (called reparse points in Windows) and symbolic links are supported to help organize the individual volumes.

The low-level I/O model in Windows is fundamentally asynchronous. Once an I/O operation is begun, the system call can return and allow the thread which initiated the I/O to continue in parallel with the I/O operation. Windows supports cancellation, as well as a number of different mechanisms for threads to synchronize with I/O operations when they complete. Windows also allows programs to specify that I/O should be synchronous when a file is opened, and many library functions, such as the C library and many Win32 calls, specify synchronous I/O for compatibility or to simplify the programming model. In these cases, the executive will explicitly synchronize with I/O completion before returning to user mode.

Another area for which Win32 provides calls is security. Every thread is associated with a kernel-mode object, called a **token**, which provides information about

the identity and privileges associated with the thread. Every object can have an **ACL** (**Access Control List**) telling in great detail precisely which users may access it and which operations they may perform on it. This approach provides for fine-grained security in which specific users can be allowed or denied specific access to every object. The security model is extensible, allowing applications to add new security rules, such as limiting the hours access is permitted.

The Win32 namespace is different than the native NT namespace described in the previous section. Only parts of the NT namespace are visible to Win32 APIs (though the entire NT namespace can be accessed through a Win32 hack that uses special prefix strings, like "\\.\"). In Win32, files are accessed relative to *drive letters*. The NT directory *DosDevices* contains a set of symbolic links from drive letters to the actual device objects. For example, *DosDevices**C:* might be a link to *Device**HarddiskVolume1*. This directory also contains links for other Win32 devices, such as *COM1:*, *LPT:*, and *NUL:* (for the serial and printer ports and the all-important null device). *DosDevices* is really a symbolic link to *??* which was chosen for efficiency. Another NT directory, *BaseNamedObjects*, is used to store miscellaneous named kernel-mode objects accessible through the Win32 API. These include synchronization objects like semaphores, shared memory, timers, communication ports, and device names.

In addition to low-level system interfaces we have described, the Win32 API also supports many calls for GUI operations, including all the calls for managing the graphical interface of the system. There are calls for creating, destroying, managing, and using windows, menus, tool bars, status bars, scroll bars, dialog boxes, icons, and many more items that appear on the screen. There are calls for drawing geometric figures, filling them in, managing the color palettes they use, dealing with fonts, and placing icons on the screen. In contrast, in Linux, none of this is in the kernel. Finally, there are calls for dealing with the keyboard, mouse, and other human-input devices as well as audio, printing, and other output devices.

The GUI operations work directly with the *win32k.sys* driver using special interfaces to access these functions in kernel mode from user-mode libraries. Since these calls do not involve the core system calls in the NTOS executive, we will not say more about them.

11.2.5 The Windows Registry

The root of the NT namespace is maintained in the kernel. Storage, such as file-system volumes, is attached to the NT namespace. Since the NT namespace is constructed afresh every time the system boots, how does the system know about any specific details of the system configuration? The answer is that Windows attaches a special kind of file system (optimized for small files) to the NT namespace. This file system is called the **registry**. The registry is organized into separate volumes called **hives**. Each hive is kept in a separate file (in the directory *C:**Windows**system32**config*\ of the boot volume). When a Windows system

boots, a hive named *SYSTEM* is loaded into memory by the boot program that loads the kernel and other boot files, such as boot drivers, from the boot volume.

Windows keeps much crucial information in the SYSTEM hive, including information about what drivers to use with what devices, what software to run initially, and many parameters governing the operation of the system. This information is used even by the boot program itself to determine which drivers are boot drivers, being needed immediately upon boot. Such drivers include those that understand the file system and disk drivers for the volume containing the operating system itself.

Other configuration hives are used after the system boots to describe information about the software installed on the system, particular users, and the classes of user-mode **COM** (**Component Object-Model**) objects that are installed on the system. Login information for local users is kept in the **SAM** (**Security Access Manager**) hive. Information for network users is maintained by the *lsass* service in the security hive and coordinated with the network directory servers so that users can have a common account name and password across an entire network. A list of the hives used in Windows is shown in Fig. 11-9.

Hive file	Mounted name	Use
SYSTEM	HKLM\SYSTEM	OS configuration information, used by kernel
HARDWARE	HKLM\HARDWARE	In-memory hive recording hardware detected
BCD	HKLM\BCD*	Boot Configuration Database
SAM	HKLM\SAM	Local user account information
SECURITY	HKLM\SECURITY	lsass' account and other security information
DEFAULT	HKEY_USERS\.DEFAULT	Default hive for new users
NTUSER.DAT	HKEY_USERS\<user id>	User-specific hive, kept in home directory
SOFTWARE	HKLM\SOFTWARE	Application classes registered by COM
COMPONENTS	HKLM\COMPONENTS	Manifests and dependencies for sys. components

Figure 11-9. The registry hives in Windows. HKLM is a shorthand for *HKEY_LOCAL_MACHINE*.

Prior to the introduction of the registry, configuration information in Windows was kept in hundreds of *.ini* (initialization) files spread across the disk. The registry gathers these files into a central store, which is available early in the process of booting the system. This is important for implementing Windows plug-and-play functionality. Unfortunately, the registry has become very seriously disorganized over time as Windows has evolved. There are poorly defined conventions about how the configuration information should be arranged, and many applications take an ad hoc approach, leading to interference between them. Also, even though most applications do not, by default, run with administrative privileges, they can escalate to get full privileges and modify system parameters in the registry directly, potentially destabilizing the system. Fixing the registry would break a lot of software.

This is one of the problems UWP application model and more specifically its AppContainer sandbox aims to solve. UWP applications cannot directly access or modify the registry. Rules are somewhat more relaxed for MSIX packaged applications: access to the registry is allowed, but their registry namespace is virtualized such that writes to global or per-user locations are redirected to per-user-per-app locations. This mechanism prevents such applications from potentially destabilizing the system by modifying system settings and eliminates risk of interference between multiple applications.

The registry is accessible to Win32 applications. There are calls to create and delete keys, look up values within keys, and more. Some of the more useful ones are listed in Fig. 11-10.

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key
RegSetValueEx	Modifies data for a value within a key
RegFlushKey	Persist any modifications on the given key to disk

Figure 11-10. Some of the Win32 API calls for using the registry

The registry is a cross between a file system and a database, and yet really unlike either. It's really a key-value store with hierarchical keys. Entire books have been written describing the registry (Hipson, 2002; Halsey and Bettany, 2015; and Ngoie, 2021) and many companies have sprung up to sell special software just to manage the complexity of the registry.

To explore the registry, Windows has a GUI program called **regedit** that allows you to open and explore the directories (called *keys*) and data items (called *values*). Microsoft's **PowerShell** scripting language can also be useful for walking through the keys and values of the registry as if they were directories and files. A more interesting tool to use is *procmon*, which is available from Microsoft's tools' Website: *https://www.microsoft.com/technet/sysinternals*. Procmon watches all the registry accesses that take place in the system and is very illuminating. Some programs will access the same key over and over tens of thousands of times.

Registry APIs are some of the most frequently used Win32 APIs in the system. They need to be fast and reliable. So, the registry implements caching of registry data in memory for fast access, but also persists data on disk to avoid losing too many changes even when RegFlushKey is not called. Because registry integrity is so critical to correct system functioning, the registry uses *write-ahead-logging* similar to database systems to record modifications sequentially into log files before

actually modifying hive files. This approach ensures consistency with minimal overhead and allows recovery of registry data in the face of system crashes or power outages.

11.3 SYSTEM STRUCTURE

In the previous sections, we examined Windows as seen by the programmer writing code for user mode. Now we are going to look under the hood to see how the system is organized internally, what the various components do, and how they interact with each other and with user programs. This is the part of the system seen by the programmer implementing low-level user-mode code, like subsystems and native services, as well as the view of the system provided to device-driver writers.

Although there are many books on how to use Windows, there are fewer on how it works inside. One of the best places to look for additional information on this topic is *Microsoft Windows Internals*, 7th ed. Part 1 (Yosifovich et al, 2017) and *Microsoft Windows Internals*, 7th ed. Part 2. (Allievi et al., 2021).

11.3.1 Operating System Structure

As described earlier, the Windows operating system consists of many layers, as depicted in Fig. 11-4. In the following sections, we will dig into the lowest levels of the operating system: those that run in kernel mode. The central layer is the NTOS kernel itself, which is loaded from *ntoskrnl.exe* when Windows boots. NTOS itself consists of two layers, the **executive**, which containa most of the services, and a smaller layer which is (also) called the **kernel** and implements the underlying thread scheduling and synchronization abstractions (a kernel within the kernel?), as well as implementing trap handlers, interrupts, and other aspects of how the CPU is managed.

The division of NTOS into kernel and executive is a reflection of NT's VAX/VMS roots. The VMS operating system, which was also designed by Cutler, had four hardware-enforced layers: user, supervisor, executive, and kernel corresponding to the four protection modes provided by the VAX processor architecture. The Intel CPUs also support four rings of protection, but some of the early target processors for NT did not, so the kernel and executive layers represent a software-enforced abstraction, and the functions that VMS provides in supervisor mode, such as printer spooling, are provided by NT as user-mode services.

The kernel-mode layers of NT are shown in Fig. 11-11. The kernel layer of NTOS is shown above the executive layer because it implements the trap and interrupt mechanisms used to transition from user mode to kernel mode.

The uppermost layer in Fig. 11-11 is the system library (*ntdll.dll*), which actually runs in user mode. The system library includes a number of support functions

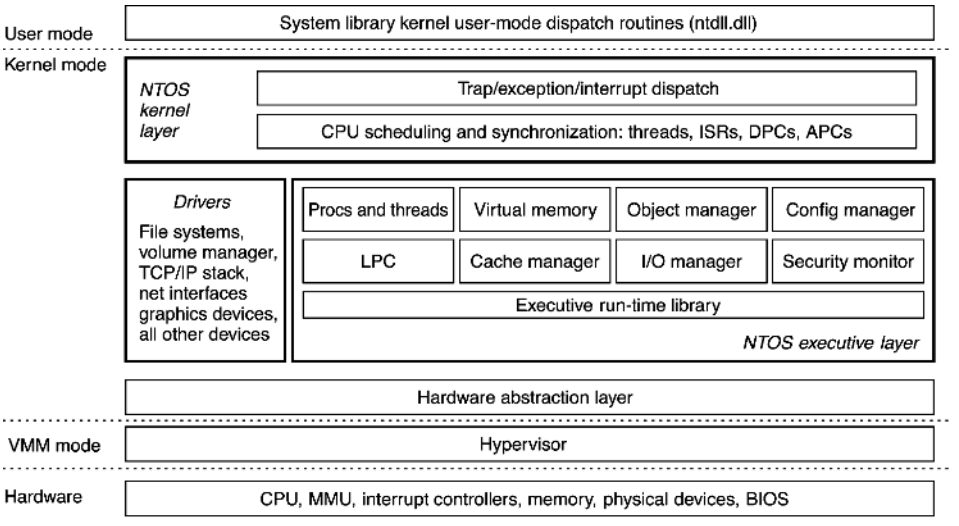

Figure 11-11. Windows kernel-mode organization.

for the compiler runtime and low-level libraries, similar to what is in *libc* in UNIX. *ntdll.dll* also contains special code entry points used by the kernel to initialize threads and dispatch exceptions and user-mode asynchronous procedure calls (described later). Because the system library is so integral to the operation of the kernel, every user-mode process created by NTOS has *ntdll* mapped at the same address (the particular address is randomized in every boot session as a security measure). When NTOS is initializing the system, it creates a section object to use when mapping *ntdll*, and it also records addresses of the *ntdll* entry points used by the kernel.

Below the NTOS kernel and executive layers is a layer of software called the **HAL** (**Hardware Abstraction Layer**) which abstracts low-level hardware details like access to device registers and DMA operations, and the way the motherboard firmware represents configuration information and deals with differences in the CPU support chips, such as various interrupt controllers.

The lowest software layer is the **hypervisor** which is the core of Windows' virtualization stack, called **Hyper-V**. It is a Type-1 (bare metal) hypervisor that runs on top of the hardware and supports concurrently running multiple operating systems. The hypervisor relies on the virtualization stack components running in the *root* operating system to virtualize guest operating systems. The hypervisor was an optional feature in earlier versions of Windows, but Windows 11 enables virtualization by default in order to provide critical security features which we will describe in subsequent sections. Hyper-V requires a 64-bit processor with hardware virtualization support and this is reflected in the minimum hardware requirements of the OS. Consequently, older computers cannot run Windows 11.

The other major components of kernel mode are the device drivers. Windows uses device drivers for any kernel-mode facilities which are not part of NTOS or the HAL. This includes file systems, network protocol stacks, and kernel extensions like antivirus and DRM (Digital Rights Management) software, as well as drivers for managing physical devices, interfacing to hardware buses, and so on.

The I/O and virtual memory components cooperate to load (and unload) device drivers into kernel memory and link them to the NTOS and HAL layers. The I/O manager provides interfaces which allow devices to be discovered, organized, and operated—including arranging to load the appropriate device driver. Much of the configuration information for managing devices and drivers is maintained in the SYSTEM hive of the registry. The plug-and-play subcomponent of the I/O manager maintains information about the hardware detected within the HARDWARE hive, which is a volatile hive maintained in memory rather than on disk, as it is completely recreated every time the system boots.

We will now examine the various components of the operating system in a bit more detail.

The Hypervisor

The Hyper-V hypervisor runs as the lowest software layer underneath Windows. Its job is to virtualize the hardware such that multiple *guest* operating systems can run concurrently, each in their own virtual machine, which Windows calls a **partition**. The hypervisor achieves this by taking advantage of virtualization extensions supported by the CPU (VT-X on Intel, AMD-V on AMD and ARMv8-A on ARM processors) to confine each guest to its assigned memory, CPU, and hardware resources, isolated from other guests. Also, the hypervisor intercepts many of the privileged operations performed by guest operating systems and emulates them to maintain the illusion. An operating system running on top of the hypervisor executes threads and handles interrupts on abstractions of the physical processors called **virtual processors**. The hypervisor schedules the virtual processors on physical processors.

Being a Type-1 hypervisor, the Windows hypervisor runs directly on the underlying hardware, but uses its virtualization stack components in the root operating system to provide device support services to its guests. For example, an emulated disk read request initiated by a guest operating system is handled by the virtual disk controller component running in user-mode by performing the requested read operation using regular Win32 APIs. While the root operating system must be Windows when running Hyper-V, other operating systems, such as Linux, can be run in the guest partitions. A guest operating system may perform very poorly unless it has been modified (i.e., paravirtualized) to work with the hypervisor.

For example, if a guest operating system kernel is using a spinlock to synchronize between two virtual processors and the hypervisor reschedules the virtual processor holding the spinlock, the lock hold time may increase by several orders

of magnitude, leaving other virtual processors running in the partition spinning for very long periods of time. To solve this problem, a guest operating system is *enlightened* to spin only a short time before calling into the hypervisor to yield its physical processor to run another virtual processor.

While the main job of the hypervisor is to run guest operating systems, it also helps improve the security of Windows by exposing a secure execution environment called **VSM** (**Virtual Secure Mode**) in which a security-focused micro-OS called the **SK** (**Secure Kernel**) runs. The Secure Kernel provides a set of security services to Windows, collectively termed **VBS** (**Virtualization-Based Security**). These services help protect code flow and integrity of OS components and maintain consistency of sensitive OS data structures as well as processor registers. In Sec. 11.10 we will discuss the inner workings of Hyper-V virtualization stack and learn how Virtualization-based Security works.

The Hardware Abstraction Layer

One goal of Windows is to make the system portable across hardware platforms. Ideally, to bring up an operating system on a new type of computer system, it should be possible to just recompile the operating system on the new platform. Unfortunately, it is not this simple. While many of the components in some layers of the operating system can be largely portable (because they mostly deal with internal data structures and abstractions that support the programming model), other layers must deal with device registers, interrupts, DMA, and other hardware features that differ significantly from machine to machine.

Most of the source code for the NTOS kernel is written in C rather than assembly language (only 2% is assembly on x86, and less than 1% on x64). However, all this C code cannot just be scooped up from an x86 system, plopped down on, say, an ARM system, recompiled, and rebooted owing to the many hardware differences between processor architectures that have nothing to do with the different instruction sets and which cannot be hidden by the compiler. Languages like C make it difficult to abstract away some hardware data structures and parameters, such as the format of page table entries and the physical memory page sizes and word length, without severe performance penalties. All of these, as well as a slew of hardware-specific optimizations, would have to be manually ported even though they are not written in assembly code.

Hardware details about how memory is organized on large servers, or what hardware synchronization primitives are available, can also have a big impact on higher levels of the system. For example, NT's virtual memory manager and the kernel layer are aware of hardware details related to cache and memory locality. Throughout the system, NT uses compare&swap synchronization primitives, and it would be difficult to port to a system that does not have them. Finally, there are many dependencies in the system on the ordering of bytes within words. On all the systems NT has ever been ported to, the hardware was set to little-endian mode.

Besides these larger issues of portability, there are also minor ones even between different motherboards from different manufacturers. Differences in CPU versions affect how synchronization primitives like spin-locks are implemented. There are several families of support chips that create differences in how hardware interrupts are prioritized, how I/O device registers are accessed, management of DMA transfers, control of the timers and real-time clock, multiprocessor synchronization, working with firmware facilities such as **ACPI** (**Advanced Configuration and Power Interface**), and so on. Microsoft made a serious attempt to hide these types of machine dependencies in a thin layer at the bottom called the HAL, as mentioned earlier. The job of the HAL is to present the rest of the operating system with abstract hardware that hides the specific details of processor version, support chipset, and other configuration variations. These HAL abstractions are presented in the form of machine-independent services (procedure calls and macros) that NTOS and the drivers can use.

By using the HAL services and not addressing the hardware directly, drivers and the kernel require fewer changes when being ported to new processors—and in most cases can run unmodified on systems with the same processor architecture, despite differences in versions and support chips.

The HAL does not provide abstractions or services for specific I/O devices such as keyboards, mice, and disks or for the memory management unit. These facilities are spread throughout the kernel-mode components, and without the HAL the amount of code that would have to be modified when porting would be substantial, even when the actual hardware differences were small. Porting the HAL itself is straightforward because all the machine-dependent code is concentrated in one place and the goals of the port are well defined: implement all of the HAL services. For many releases, Microsoft supported a *HAL Development Kit* allowing system manufacturers to build their own HAL, which would allow other kernel components to work on new systems without modification, provided that the hardware changes were not too great. This practice is no longer active and as such, there's little reason to maintain the HAL layer in a separate binary, *hal.dll*. With Windows 11, the HAL layer has been merged into ntoskrnl.exe. *Hal.dll* is now a forwarder binary kept around to maintain compatibility with drivers that use its interfaces all of which are redirected to the HAL layer in ntoskrnl.exe.

As an example of what the hardware abstraction layer does, consider the issue of memory-mapped I/O vs. I/O ports. Some machines have one and some have the other. How should a driver be programmed: to use memory-mapped I/O or not? Rather than forcing a choice, which would make the driver not portable to a machine that did it the other way, the hardware abstraction layer offers procedures for driver writers to use for reading the device registers others for writing them:

```
uc = READ_PORT_UCHAR(port);          WRITE_PORT_UCHAR(port, uc);
us = READ_PORT_USHORT(port);         WRITE_PORT_USHORT(port, us);
ul = READ_PORT_ULONG(port);          WRITE_PORT_LONG(port, ul);
```

These procedures read and write unsigned 8-, 16-, and 32-bit integers, respectively, to the specified port. It is up to the hardware abstraction layer to decide whether memory-mapped I/O is needed here. In this way, a driver can be moved without modification between machines that differ in the way the device registers are implemented.

Drivers frequently need to access specific I/O devices for various purposes. At the hardware level, a device has one or more addresses on a certain bus. Since modern computers often have multiple buses (PCIe, USB, IEEE 1394, etc.), it can happen that more than one device may have the same address on different buses, so some way is needed to distinguish them. The HAL provides a service for identifying devices by mapping bus-relative device addresses onto systemwide logical addresses. In this way, drivers do not have to keep track of which device is connected to which bus. This mechanism also shields higher layers from properties of alternative bus structures and addressing conventions.

Interrupts have a similar kind of problem—they are also bus dependent. Here, too, the HAL provides services to name interrupts in a systemwide way and also provides ways to allow drivers to attach interrupt service routines to interrupts in a portable way, without having to know anything about which interrupt vector is for which bus. Interrupt request level management is also handled in the HAL.

Another HAL service is setting up and managing DMA transfers in a device-independent way. Both the systemwide DMA engine and DMA engines on specific I/O cards can be handled. Devices are referred to by their logical addresses. The HAL implements software scatter/gather (writing or reading from noncontiguous blocks of physical memory).

The HAL also manages clocks and timers in a portable way. Time is kept track of in units of 100 nanoseconds starting at midnight at the start of Jan. 1, 1601, which is the first date in the previous quadricentury, which simplifies leap-year computations. (Quick Quiz: Was 1800 a leap year? Quick Answer: No. QQ2: Was 2000 a leap year? QA2: Yes. Until 3999, century years are not leap years except 400 years). Under the current rules, 4000 should be a leap year, but in the current model isn't quite right and making 4000 a nonleap year would help. Not everyone agrees however. The time services decouple the drivers from the actual frequencies at which the clocks run.

Kernel components sometimes need to synchronize at a very low level, especially to prevent race conditions in multiprocessor systems. The HAL provides primitives to manage this synchronization, such as spin locks, in which one CPU simply waits for a resource held by another CPU to be released, particularly in situations where the resource is typically held only for a few machine instructions.

Finally, after the system has been booted, the HAL talks to the computer's firmware (BIOS or UEFI) and inspects the system configuration to find out which buses and I/O devices the system contains and how they have been configured. This information is then put into the registry. A brief summary of some of the things the HAL does is given in Fig. 11-12.

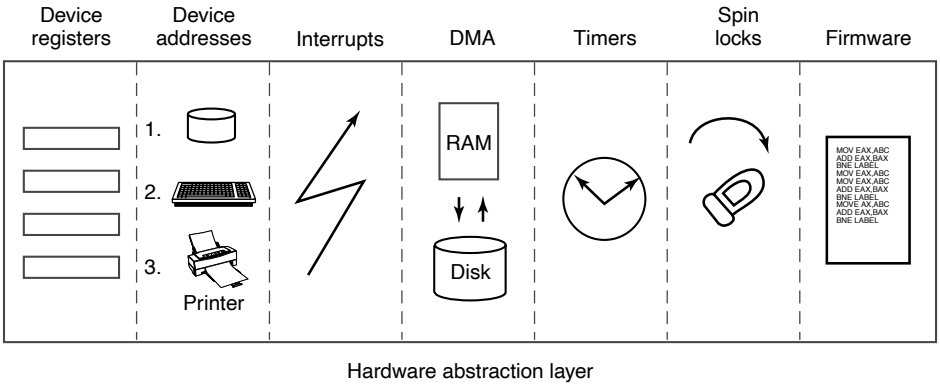

Figure 11-12. Some of the hardware functions the HAL manages.

The Kernel Layer

Above the hardware abstraction layer is NTOS, consisting of two layers: the kernel and the executive. "Kernel" is a confusing term in Windows. It can refer to all the code that runs in the processor's kernel mode. It can also refer to the *ntoskrnl.exe* file which contains NTOS, the core of the Windows operating system. Or it can refer to the kernel layer within NTOS, which is how we use it in this section. It is even used to name the user-mode Win32 library that provides the wrappers for the native system calls: *kernelbase.dll*.

In the Windows operating system, the kernel layer, illustrated above the executive layer in Fig. 11-11, provides a set of abstractions for managing the CPU. The most central abstraction is threads, but the kernel also implements exception handling, traps, and several kinds of interrupts. Creating and destroying the data structures which support threading is implemented in the executive layer. The kernel layer is responsible for scheduling and synchronization of threads. Having support for threads in a separate layer allows the executive layer to be implemented using the same preemptive multithreading model used to write concurrent code in user mode, though the synchronization primitives in the executive are much more specialized.

The kernel's thread scheduler is responsible for determining which thread is executing on each CPU in the system. Each thread executes until a timer interrupt signals that it is time to switch to another thread (quantum expired), or until the thread needs to wait for something to happen, such as an I/O to complete or for a lock to be released, or a higher-priority thread becomes runnable and needs the CPU. When switching from one thread to another, the scheduler runs on the CPU and ensures that the registers and other hardware state have been saved. The scheduler then selects another thread to run on the CPU and restores the state that was previously saved from the last time that thread ran.

If the next thread to be run is in a different address space (i.e., process) than the thread being switched from, the scheduler must also change address spaces. The details of the scheduling algorithm itself will be discussed later in this chapter when we come to processes and threads.

In addition to providing a higher-level abstraction of the hardware and handling thread switches, the kernel layer also has another key function: providing low-level support for two classes of synchronization mechanisms: control objects and dispatcher objects. **Control objects** are the data structures that the kernel layer provides as abstractions to the executive layer for managing the CPU. They are allocated by the executive but they are manipulated with routines provided by the kernel layer. **Dispatcher objects** are the class of ordinary executive objects that use a common data structure for synchronization.

Deferred Procedure Calls

Control objects include primitive objects for threads, interrupts, timers, synchronization, profiling, and two special objects for implementing **DPCs** (**Deferred Procedure Calls**) and APCs (see below). DPC objects are used to reduce the time taken to execute **ISRs** (**Interrupt Service Routines**) in response to an interrupt from a particular device. Limiting time spent in ISRs reduces the chance of losing an interrupt.

The system hardware assigns a hardware priority level to interrupts. The CPU also associates a priority level with the work it is performing. The CPU responds only to interrupts at a higher-priority level than it is currently using. Normal priority level, including the priority level of all user-mode work, is 0. Device interrupts occur at priority 3 or higher, and the ISR for a device interrupt normally executes at the same priority level as the interrupt in order to keep other less important interrupts from occurring while it is processing a more important one.

If an ISR executes too long, the servicing of lower-priority interrupts will be delayed, perhaps causing data to be lost or slowing the I/O throughput of the system. Multiple ISRs can be in progress at any one time, with each successive ISR being due to interrupts at higher and higher-priority levels.

To reduce the time spent processing ISRs, only the critical operations are performed, such as capturing the result of an I/O operation and reinitializing the device. Further processing of the interrupt is deferred until the CPU priority level is lowered and no longer blocking the servicing of other interrupts. The DPC object is used to represent the further work to be done and the ISR calls the kernel layer to queue the DPC to the list of DPCs for a particular processor. If the DPC is the first on the list, the kernel registers a special request with the hardware to interrupt the CPU at priority 2 (which NT calls DISPATCH level). When the last of any executing ISRs completes, the interrupt level of the processor will drop back below 2, and that will unblock the interrupt for DPC processing. The ISR for the DPC interrupt will process each of the DPC objects that the kernel had queued.

The technique of using software interrupts to defer interrupt processing is a well-established method of reducing ISR latency. UNIX and other systems started using deferred processing in the 1970s to deal with the slow hardware and limited buffering of serial connections to terminals. The ISR would deal with fetching characters from the hardware and queuing them. After all higher-level interrupt processing was completed, a software interrupt would run a low-priority ISR to do character processing, such as implementing backspace by sending control characters to the terminal to erase the last character displayed and move the cursor back.

A similar example in Windows today is the keyboard device. After a key is struck, the keyboard ISR reads the key code from a register and then reenables the keyboard interrupt but does not do further processing of the key immediately. Instead, it uses a DPC to queue the processing of the key code until all outstanding device interrupts have been processed.

Because DPCs run at level 2, they do not keep device ISRs from executing, but they do prevent any threads from running on that processor until all the queued DPCs complete and the CPU priority level is lowered below 2. Device drivers and the system itself must take care not to run either ISRs or DPCs for too long. Because threads are not allowed to execute, ISRs and DPCs can make the system appear sluggish and produce glitches when playing music by stalling the threads writing the music buffer to the sound device. Another common use of DPCs is running routines in response to a timer interrupt. To avoid blocking threads, timer events which need to run for an extended time should queue requests to the pool of worker threads the kernel maintains for background activities.

The problem of thread starvation due to excessively long or frequent DPCs (called **DPC Storms**) is common enough that Windows implements a defense mechanism called the **DPC Watchdog**. The DPC Watchdog has time limits for individual DPCs and for back-to-back DPCs. When these limits are exceeded, the watchdog issues a system crash with the DPC_WATCHDOG_VIOLATION code and information about the long DPC (typically a buggy driver) along with a crash dump which can help diagnose the issue.

Even though DPC storms are undesirable, so are system crashes. In environments like the Azure Cloud where DPC storms due to incoming network packets are relatively common and system crashes are catastrophic, DPC watchdog timeouts are typically configured higher to avoid crashes. To improve diagnosability in such situations, the DPC watchdog in Windows 11 supports *soft* and *profiling* thresholds. When the soft threshold is crossed, instead of crashing the system, the watchdog instead logs information which can later be analyzed to determine the source of the DPCs. When the profiling threshold is crossed, the watchdog starts a *profiling timer* and logs a stack trace of DPC execution every millisecond such that much more detailed analysis can be performed to understand the root cause of long or frequent DPCs.

In addition to the improved DPC watchdog, the Windows 11 thread scheduler is also smarter about reducing thread starvation in the face of DPCs. For each

recent DPC, it maintains a short history of DPC runtime which is used to identify *long-running* DPCs. When such long-running DPCs are queued up on a processor, the currently-running thread (which is about to be starved) is rescheduled to another available processor if the thread is high-enough priority. This way, time-critical threads like those feeding media devices are much less likely to be starved due to DPCs.

Asynchronous Procedure Calls

The other special kernel control object is the **APC** (**Asynchronous Procedure Call**) object. APCs are like DPCs in that they defer processing of a system routine, but unlike DPCs, which operate in the context of particular CPUs, APCs execute in the context of a specific thread. When processing a key press, it does not matter which context the DPC runs in because a DPC is simply another part of interrupt processing, and interrupts only need to manage the physical device and perform thread-independent operations such as recording the data in a buffer in kernel space.

The DPC routine runs in the context of whatever thread happened to be running when the original interrupt occurred. It calls into the I/O system to report that the I/O operation has been completed, and the I/O system queues an APC to run in the context of the thread making the original I/O request, where it can access the user-mode address space of the thread that will process the input.

At the next convenient time, the kernel layer delivers the APC to the thread and schedules the thread to run. An APC is designed to look like an unexpected procedure call, somewhat similar to signal handlers in UNIX. The kernel-mode APC for completing I/O executes in the context of the thread that initiated the I/O, but in kernel mode. This gives the APC access to both the kernel-mode buffer as well as all of the user-mode address space belonging to the process containing the thread. *When* an APC is delivered depends on what the thread is already doing, and even what type of system. In a multiprocessor system, the thread receiving the APC may begin executing even before the DPC finishes running.

User-mode APCs can also be used to deliver notification of I/O completion in user mode to the thread that initiated the I/O. User-mode APCs invoke a user-mode procedure designated by the application, but only when the target thread has blocked in the kernel and is marked as willing to accept APCs, a state known as an **alertable wait**. The kernel interrupts the thread from waiting and returns to user mode, but with user-mode stack and registers modified to run the APC dispatch routine in the *ntdll.dll* system library. The APC dispatch routine invokes the user-mode routine that the application has associated with the I/O operation. Besides specifying user-mode APCs as a means of executing code when I/Os complete, the Win32 API QueueUserAPC allows APCs to be used for arbitrary purposes.

Special User-mode APCs are a flavor of APC that were introduced in later Windows 10 releases. These are different from "normal" user-mode APCs in that

they are completely asynchronous: they can execute even when the target thread is not in an alertable wait state. As such, special user APCs are the equivalent of UNIX signals, available to developers via the QueueUserAPC2 API. Prior to the advent of special user APCs, developers who needed to run code in arbitrary threads (e.g., for garbage collection in a managed runtime) had to resort to using more complicated mechanisms like manually changing the context of the target thread using SetThreadContext.

The executive layer also uses APCs for operations other than I/O completion. Because the APC mechanism is carefully designed to deliver APCs only when it is safe to do so, it can be used to safely terminate threads. If it is not a good time to terminate the thread, the thread will have declared that it was entering a critical region and defer deliveries of APCs until it leaves. Kernel threads mark themselves as entering critical regions to defer APCs when acquiring locks or other resources, so that they cannot be terminated while still holding the resource or deadlock due to reentrancy. The thread termination APC is very similar to a special user-mode APC except that it is "extra special" because it runs before any special user APC to terminate the thread immediately.

Dispatcher Objects

Another kind of synchronization object is the **dispatcher object**. This is any ordinary kernel-mode object (the kind that users can refer to with handles) that contains a data structure called a **dispatcher_header**, shown in Fig. 11-13. These objects include semaphores, mutexes, events, waitable timers, and other objects that threads can wait on to synchronize execution with other threads. They also include objects representing open files, processes, threads, and IPC ports. The dispatcher data structure contains a flag representing the signaled state of the object, and a queue of threads waiting for the object to be signaled.

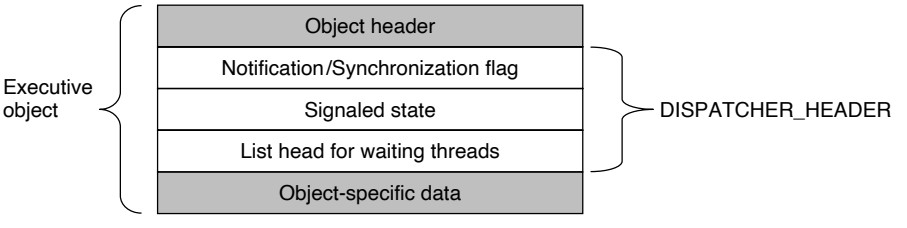

Figure 11-13. *Dispatcher_header* data structure embedded in many executive objects (*dispatcher objects*).

Synchronization primitives, like semaphores, are natural dispatcher objects. Also timers, files, ports, threads, and processes use the dispatcher-object mechanisms in order to do notifications. When a timer goes off, I/O completes on a file,

data are available on a port, or a thread or process terminates, the associated dispatcher object is signaled, waking all threads waiting for that event.

Since Windows uses a single unified mechanism for synchronization with kernel-mode objects, specialized APIs, such as wait3, for waiting for child processes in UNIX, are not needed to wait for events. Often threads want to wait for multiple events at once. In UNIX a process can wait for data to be available on any of 64 network sockets using the select system call. In Windows, there is a similar API **WaitForMultipleObjects**, but it allows for a thread to wait on any type of dispatcher object for which it has a handle. Up to 64 handles can be specified to WaitForMultipleObjects, as well as an optional timeout value. The thread becomes ready to run whenever any of the events associated with the handles is signaled or the timeout occurs.

There are actually two different procedures the kernel uses for making the threads waiting on a dispatcher object runnable. Signaling a **notification object** will make every waiting thread runnable. **Synchronization objects** make only the first waiting thread runnable and are used for dispatcher objects that implement locking primitives, like mutexes. When a thread that is waiting for a lock begins running again, the first thing it does is to retry acquiring the lock. If only one thread can hold the lock at a time, all the other threads made runnable might immediately block, incurring lots of unnecessary context switching. The difference between dispatcher objects using synchronization vs. notification is a flag in the dispatcher_header structure.

As a little aside, mutexes in Windows are called "mutants" in the code because they were required to implement the OS/2 semantics of not automatically unlocking themselves when a thread holding one exited, something Cutler considered bizarre.

The Executive Layer

As shown in Fig. 11-11, below the kernel layer of NTOS there is the executive. The executive layer is written in C, is mostly architecture independent (the memory manager being a notable exception), and has been ported with only modest effort to new processors (MIPS, x86, PowerPC, Alpha, IA64, x64, arm32, and arm64). The executive contains a number of different components, all of which run using the control abstractions provided by the kernel layer.

Each component is divided into internal and external data structures and interfaces. The internal aspects of each component are hidden and used only within the component itself, while the external aspects are available to all the other components within the executive. A subset of the external interfaces are exported from the *ntoskrnl.exe* executable and device drivers can link to them as if the executive were a library. Microsoft calls many of the executive components "managers," because each is charge of managing some aspect of the operating services, such as I/O, memory, processes, and objects.

As with most operating systems, much of the functionality in the Windows executive is like library code, except that it runs in kernel mode so its data structures can be shared and protected from access by user-mode code, and so it can access kernel-mode state, such as the MMU control registers. But otherwise the executive is simply executing operating system functions on behalf of its caller, and thus runs in the thread of its caller. This is the same as in UNIX systems.

When any of the executive functions block waiting to synchronize with other threads, the user-mode thread is blocked, too. This makes sense when working on behalf of a particular user-mode thread, but it can be unfair when doing work related to common housekeeping tasks. To avoid hijacking the current thread when the executive determines that some housekeeping is needed, a number of kernel-mode threads are created when the system boots and dedicated to specific tasks, such as making sure that modified pages get written to disk.

For predictable, low-frequency tasks, there is a thread that runs once a second and has a laundry list of items to handle. For less predictable work, there is the pool of high-priority worker threads mentioned earlier which can be used to run bounded tasks by queuing a request and signaling the synchronization event that the worker threads are waiting on.

The **object manager** manages most of the interesting kernel-mode objects used in the executive layer. These include processes, threads, files, semaphores, I/O devices and drivers, timers, and many others. As described previously, kernel-mode objects are really just data structures allocated and used by the kernel. In Windows, kernel data structures have enough in common that it is very useful to manage many of them in a unified facility.

The facilities provided by the object manager include managing the allocation and freeing of memory for objects, quota accounting, supporting access to objects using handles, maintaining reference counts for kernel-mode pointer references as well as handle references, giving objects names in the NT namespace, and providing an extensible mechanism for managing the lifecycle for each object. Kernel data structures which need some of these facilities are managed by the object manager.

Object-manager objects each have a type which is used to specify exactly how the lifecycle of objects of that type is to be managed. These are not types in the object-oriented sense, but are simply a collection of parameters specified when the object type is created. To create a new type, an executive component calls an object-manager API to create a new type. Objects are so central to the functioning of Windows that the object manager will be discussed in more detail in the next section.

The **I/O manager** provides the framework for implementing I/O device drivers and provides a number of executive services specific to configuring, accessing, and performing operations on devices. In Windows, device drivers not only manage physical devices but they also provide extensibility to the operating system. Many functions that are hard compiled into the kernel on other systems are dynamically

loaded and linked by the kernel on Windows, including network protocol stacks and file systems.

Recent versions of Windows have a lot more support for running device drivers in user mode, and this is the preferred model for new device drivers. There are hundreds of thousands of different device drivers for Windows working with more than a million distinct devices. This represents a lot of code to get correct. It is much better if bugs cause a device to become inaccessible by crashing in a user-mode process rather than causing the system to crash. Bugs in kernel-mode device drivers are the major source of the dreaded **BSOD** (**Blue Screen Of Death**) where Windows detects a fatal error within kernel mode and shuts down or reboots the system. BSOD's are comparable to kernel panics on UNIX systems.

Since device drivers make up something in the vicinity of 70% of the code in the kernel, the more drivers that can be moved into user-mode processes, where a bug will only trigger the failure of a single driver (rather than bringing down the entire system), the better. The trend of moving code from the kernel to user-mode processes for improved system reliability has been accelerating in recent years.

The I/O manager also includes the plug-and-play and device power-management facilities. **Plug-and-play** comes into action when new devices are detected on the system. The plug-and-play subcomponent is first notified. It works with a service, the user-mode plug-and-play manager, to find the appropriate device driver and load it into the system. Getting the right one is not always easy and sometimes depends on sophisticated matching of the specific hardware device version to a particular version of the drivers. Sometimes a single device supports a standard interface which is supported by multiple different drivers, written by different companies.

We will study I/O further in Sec. 11.7 and the most important NT file system, NTFS, in Sec. 11.8.

Device power management reduces power consumption when possible, extending battery life on notebooks, and saving energy on desktops and servers. Getting power management correct can be challenging as there are many subtle dependencies between devices and the buses that connect them to the CPU and memory. Power consumption is not affected just by what devices are powered-on, but also by the clock rate of the CPU, which is also controlled by the device power manager. We will take a more in-depth look at power management in Sec. 11.9.

The **process manager** manages the creation and termination of processes and threads, including establishing the policies and parameters which govern them. But the operational aspects of threads are determined by the kernel layer, which controls scheduling and synchronization of threads, as well as their interaction with the control objects, like APCs. Processes contain threads, an address space, and a handle table containing the handles the process can use to refer to kernel-mode objects. Processes also include information needed by the scheduler for switching between address spaces and managing process-specific hardware information (like segment descriptors). We will study process and thread management in Sec. 11.4.

The executive **memory manager** implements the demand-paged virtual memory architecture. It manages the mapping of virtual pages onto physical page frames, the management of the available physical frames, and management of the pagefile on disk used to back private instances of virtual pages that are no longer loaded in memory. The memory manager also provides special facilities for large server applications such as databases and programming language runtime components such as garbage collectors. We will study memory management later in this chapter, in Sec. 11.5.

The **cache manager** optimizes the performance of I/O to the file system by maintaining a cache of file-system pages in the kernel virtual address space. The cache manager uses virtually addressed caching, that is, organizing cached pages in terms of their location in their files. This differs from physical block caching, as in UNIX, where the system maintains a cache of the physically addressed blocks of the raw disk volume.

Cache management is implemented using mapped files. The actual caching is performed by the memory manager. The cache manager need be concerned only with deciding what parts of what files to cache, ensuring that cached data is flushed to disk in a timely fashion, and managing the kernel virtual addresses used to map the cached file pages. If a page needed for I/O to a file is not available in the cache, the page will be faulted in using the memory manager. We will study the cache manager in Sec. 11.6.

The **security reference monitor** enforces Windows' elaborate security mechanisms, which support the international standards for computer security called **Common Criteria**, an evolution of United States Department of Defense Orange Book security requirements. These standards specify a large number of rules that a conforming system must meet, such as authenticated login, auditing, zeroing of allocated memory, and many more. One rules requires that all access checks be implemented by a single module within the system. In Windows, this module is the security reference monitor in the kernel. We will study the security system in more detail in Sec. 11.10.

The executive contains a number of other components that we will briefly describe. The **configuration manager** is the executive component which implements the registry, as described earlier. The registry contains configuration data for the system in file-system files called hives. The most critical hive is the *SYSTEM* hive which is loaded into memory every time the system is booted from disk. Only after the executive layer has successfully initialized all of its key components, including the I/O drivers that talk to the system disk, is the in-memory copy of the hive reassociated with the copy in the file system. Thus, if something bad happens while trying to boot the system, the on-disk copy very unlikely to be corrupted. If the on-disk copy were to be corrupted, that would be a disaster.

The local procedure call component provides for a highly efficient interprocess communication used between processes running on the same system. It is one of the data transports used by the standards-based remote procedure call facility to

implement the client/server style of computing. RPC also uses named pipes and TCP/IP as transports.

LPC was substantially enhanced in Windows 8 (it is now called **ALPC**, (**Advanced LPC**) to provide support for new features in RPC, including RPC from kernel mode components, like drivers. LPC was a critical component in the original design of NT because it is used by the subsystem layer to implement communication between library stub routines that run in each process and the subsystem process which implements the facilities common to a particular operating system personality, such as Win32 or POSIX.

Windows also provides a publish/subscribe service called **WNF** (**Windows Notification Facility**). WNF notifications are based on changes to an instance of WNF state data. A publisher declares an instance of state data (up to 4 KB) and tells the operating system how long to maintain it (e.g., until the next reboot or permanently). A publisher atomically updates the state as appropriate. Subscribers can arrange to run code whenever an instance of state data is modified by a publisher. Because the WNF state instances contain a fixed amount of preallocated data, there is no queuing of data as in message-based IPC—with all the attendant resource-management problems. Subscribers are guaranteed only that they can see the latest version of a state instance.

This state-based approach gives WNF its principal advantage over other IPC mechanisms: publishers and subscribers are decoupled and can start and stop independently of each other. Publishers need not execute at boot time just to initialize their state instances, as those can be persisted by the operating system across reboots. Subscribers generally need not be concerned about past values of state instances when they start running as all they should need to know about the state's history is encapsulated in the current state. In scenarios where past state values cannot be reasonably encapsulated, the current state can provide metadata for managing historical state, say, in a file or in a persisted section object used as a circular buffer. WNF is part of the native NT APIs and is not (yet) exposed via Win32 interfaces. But it is extensively used internally by the system to implement Win32 and WinRT APIs.

In Windows NT 4.0, much of the code related to the Win32 graphical interface was moved into the kernel because the then-current hardware could not provide the required performance. This code previously resided in the *csrss.exe* subsystem process which implemented the Win32 interfaces. The kernel-based GUI code resides in a special kernel-driver, *win32k.sys*. The move to kernel-mode improved Win32 performance because the extra user-mode/kernel-mode transitions and the cost of switching address spaces to implement communication via LPC was eliminated. However, it has not been without problems because the security requirements on code running in the kernel are very strict, and the complicated API interface exposed by win32k to user-mode has resulted in numerous security vulnerabilities. A future Windows release will hopefully move win32k back into a user-mode process while maintaining acceptable performance for GUI code.

The Device Drivers

The final part of Fig. 11-11 consists of the **device drivers**. Device drivers in Windows are dynamic link libraries which are loaded by the NTOS executive. Though they are primarily used to implement the drivers for specific hardware, such as physical devices and I/O buses, the device-driver mechanism is also used as the general extensibility mechanism for kernel mode. As described earlier, much of the Win32 subsystem is loaded as a driver.

The I/O manager organizes a data flow path for each instance of a device, as shown in Fig. 11-14. This path is called a **device stack** and consists of private instances of kernel device objects allocated for the path. Each device object in the device stack is linked to a particular driver object, which contains the table of routines to use for the I/O request packets that flow through the device stack. In some cases, the devices in the stack represent drivers whose sole purpose is to **filter** I/O operations aimed at a particular device, bus, or network driver. Filtering is used for a number of reasons. Sometimes preprocessing or postprocessing I/O operations results in a cleaner architecture, while other times it is just pragmatic because the sources or rights to modify a driver are not available and so filtering is used to work around the inability to modify those drivers. Filters can also implement completely new functionality, such as turning disks into partitions or multiple disks into RAID volumes.

The file systems are loaded as device drivers. Each instance of a volume for a file system has a device object created as part of the device stack for that volume. This device object will be linked to the driver object for the file system appropriate to the volume's formatting. Special filter drivers, called **file-system filter drivers**, can insert device objects before the file-system device object to apply functionality to the I/O requests being sent to each volume, such as handling encryption.

The network protocols, such as Windows' integrated IPv4/IPv6 TCP/IP implementation, are also loaded as drivers using the I/O model. For compatibility with the older MS-DOS-based Windows, the TCP/IP driver implements a special protocol for talking to network interfaces on top of the Windows I/O model. There are other drivers that also implement such arrangements, which Windows calls **miniports**. The shared functionality is in a **class driver**. For example, common functionality for SCSI or IDE disks or USB devices is supplied by a class driver, which miniport drivers for each particular type of such devices link to as a library.

We will not discuss any particular device driver in this chapter, but will provide more detail about how the I/O manager interacts with device drivers in Sec. 11.7.

11.3.2 Booting Windows

Getting an operating system to run requires several steps. When a computer is turned on, the first processor is initialized by the hardware, and then set to start executing some program in memory. The only available code is in some form of

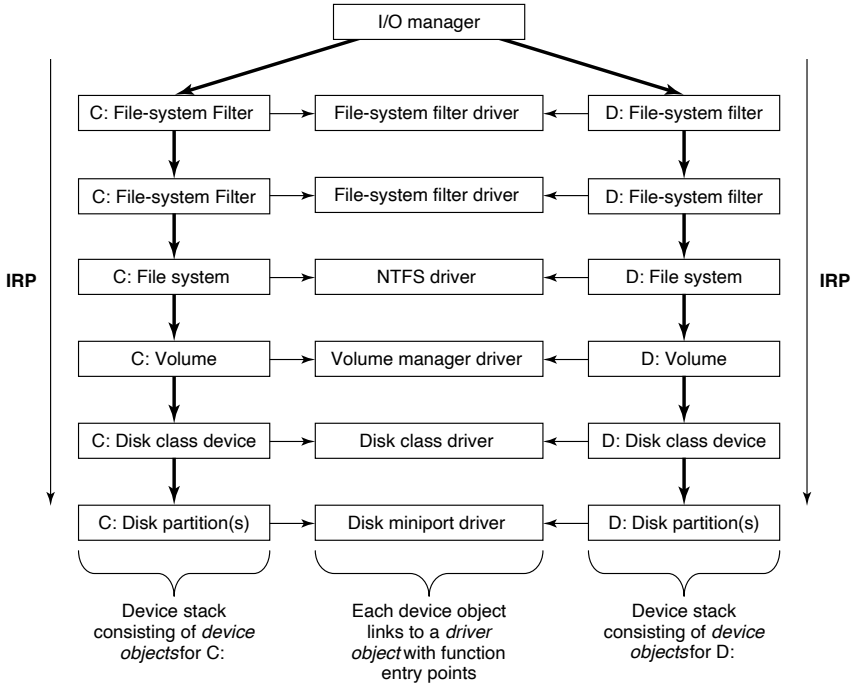

Figure 11-14. Simplified depiction of device stacks for two NTFS file volumes. The I/O request packet is passed from down the stack. The appropriate routines from the associated drivers are called at each level in the stack. The device stacks themselves consist of device objects allocated specifically to each stack.

nonvolatile CMOS memory that is initialized by the computer manufacturer (and sometimes updated by the user, in a process called **flashing**). Because the software persists in (read-only) memory, and is only rarely updated, it is referred to as **firmware**. It is held in a special chip whose contents are not lost when power is turned off. The firmware is loaded on PCs by the manufacturer of either the motherboard or the computer system. Historically, PC firmware was a program called BIOS (Basic Input/Output System), but most new computers use **UEFI** (**Unified Extensible Firmware Interface**). UEFI improves over BIOS by supporting modern hardware, providing a more modular CPU-independent architecture, much improved security mechanisms and supporting an extension model which simplifies booting over networks, provisioning new machines, and running diagnostics. Windows 11 supports only UEFI-based machines.

The main purpose of any firmware is to bring up the operating system by locating and running the bootstrap application. UEFI firmware achieves this by first requiring that the boot disk be formatted in the **GPT** (**GUID partition table**) scheme where each disk partition is identified by a **GUID** (**Globally-Unique**

IDentifier), which, in practice, is a 128-bit number generated to ensure uniqueness. The Windows setup program initializes the boot disk in the GPT format and creates several partitions. The most important are the **EFI system partition** which is formatted with FAT32 and contains the **Windows Boot Manager** UEFI application (bootmgrfw.efi) and the **boot partition** which is formatted with NTFS and contains the actual Windows installation. In addition, the setup program sets some well-known UEFI global variables that indicate to the firmware the location of Windows Boot Manager. These variables are stored in the system's nonvolatile memory and persist across boots.

Given a GPT-partitioned disk, the UEFI firmware locates the Windows Boot Manager in the EFI system partition and transfers control to it. It's able to do this because the firmware supports the FAT32 file system (but not the NTFS file system). The boot manager's job is to select the appropriate OS loader application and execute it. The OS loader's job is to load the actual operating system files into memory and start running the OS. Both the boot manager and the OS loader rely on the UEFI firmware facilities for basic memory management, disk I/O, textual and graphical console I/O. However, once all the required operating system files are loaded into memory and prepared for execution, "ownership" of the platform is transferred to the operating system kernel and these **boot services** provided by the firmware are discarded from memory. The kernel then initializes its own storage and file system drivers to mount the boot partition and load the rest of the files necessary to boot Windows.

Boot security is the foundation of OS security. The boot sequence must be protected from a special type of malware called **rootkits** which are sophisticated malicious software that inject themselves into the boot sequence, take control of the hardware, and hide themselves from the security mechanisms that load afterwards (such as anti-malware applications). As a countermeasure, UEFI supports a feature called **Secure Boot** which validates the integrity of every component loaded during the boot process including the UEFI firmware itself. This verification is performed by checking the digital signature of each component against a database of trusted certificates (or certificates issued by trusted certificates), thereby establishing a **chain of trust** rooted at the **root certificate**. As part of Secure Boot, the firmware validates the Windows Boot Manager before transferring control to it, which, then validates the OS loader, which, then validates the operating system files (hypervisor, secure kernel, kernel, boot drivers, and so on).

Digital signature verification involves calculating a cryptographic hash for the component to be verified. This hash value is also *measured* into the **TPM** (**Trusted Platform Module**) which is a secure cryptographic processor required to be present by Windows 11. The TPM provides various security services such as protection of encryption keys, boot measurements, and attestation. The act of measuring a hash value into the TPM cryptographically combines the hash value with the existing value in a **PCR** (**Platform Configuration Register**) in an operation called **extending** the PCR. The Windows Boot Manager and the OS loader measure not

only the hashes of components to be executed, but also important pieces of boot configuration such as the boot device, code signing requirements, and whether debugging is enabled. The TPM does not allow the PCR values to be manipulated in any way other than extending. As a result, PCRs provide a tamper-proof mechanism to record the OS boot sequence. This is called **Measured Boot**. Injection of a rootkit or a change in boot configuration will result in a different final PCR value. This property allows the TPM to support two important scenarios:

1. **Attestation.** Organizations may want to ensure that a computer is free of rootkits before allowing it access to the enterprise network. A trusted remote attestation server can request from each client a **TPM Quote** which is a signed collection of PCR values that can be checked against a database of acceptable values to determine whether the client is healthy.
2. **Sealing.** The TPM supports storing a secret key using PCR values such that it can be unsealed in a later boot session only if those PCRs have the same values. The BitLocker volume encryption solution uses the boot sequence PCR values to seal its encryption key into the TPM such that the key can only be revealed if the boot sequence is not tampered with.

The Windows Boot Manager orchestrates the steps to boot Windows. It first loads from the EFI system partition the **BCD** (**Boot Configuration Database**) which is registry hive containing descriptors for all boot applications and their parameters. It then checks whether the system had previously been hibernated (a special power-saving mode where the operating system state is saved to disk). If so, the boot manager runs the *winresume.efi* boot application which "resumes" Windows from the saved snapshot. Otherwise, it loads and executes the OS loader boot application, *winload.efi*, to perform a fresh boot. Both of these UEFI applications are generally located on the NTFS-formatted boot volume. The boot manager understands a wide selection of file system formats in order to support booting from various devices. Also, since the boot volume may be encrypted with BitLocker, the boot manager must request the TPM to unseal the BitLocker volume decryption key in order to access winresume or winload.

The Windows OS loader is responsible for loading the remaining boot components into memory: the hypervisor loader (*hvloader.dll*), the secure kernel (securekernel.exe), the NT kernel/executive/HAL (*ntoskrnl.exe*), the stub HAL (*hal.dll*), the SYSTEM hive as well as all boot drivers listed in the SYSTEM hive. It executes the hypervisor loader which picks the appropriate hypervisor binary based on the underlying system and starts it. Then the Secure Kernel is initialized and finally, winload transfers control to the NT Kernel entry point. NT Kernel initialization happens in several phases. Phase 0 initialization runs on the boot processor and initializes the processor structures, locks, kernel address space, and data

structures of kernel components. Phase 1 starts all the remaining processors and completes final initialization of all kernel components. At the end of Phase 1, once the I/O manager is initialized, boot drivers are started and file systems are mounted, the rest of OS boot can proceed to load new binaries from disk.

The first user-mode process to get started during boot is *smss.exe* which is similar to */etc/init* in UNIX systems. Smss first completes the initialization of the subsystem-independent parts of the operating system by creating any configured paging files and finalizing registry initialization by loading the remaining hives. Then it starts acting as a session manager: it launches new instances of itself to initialize Session 0, the non-interactive session, and Session 1, the interactive session. These child smss instances are responsible for enumerating and starting NT subsystems which are listed under the *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Subsystems* registry key. On Windows 11, the only supported subsystem is the Windows subsystem, so the child smss instance starts the Windows subsystem process, *csrss.exe*. Then the Session 0 instance executes the *wininit.exe* process to initialize the rest of the Windows subsystem while the Session 1 instance starts the *winlogon.exe* process to allow the interactive user to log in.

The Windows boot sequence has logic to deal with common problems users encounter when booting the system fails. Sometimes installation of a bad device driver, or incorrectly modifying the SYSTEM hive can prevent the system from booting successfully. To recover from these situations, Windows boot manager allows users to launch the **WinRE** (**Windows Recovery Environment**) WinRE provides an assortment of tools and automated repair mechanisms. These include **System Restore** which allows restoring the boot volume to a previous snapshot. Another is **Startup Repair** which is an automated tool that detects and fixes the most common sources of startup problems. **PC Reset** performs the equivalent of a *factory reset* to bring Windows back to its original state after installation. For cases where manual intervention may be necessary, WinRE can also launch a command prompt where the user has access to any command-line tool. Similarly, the system may be booted in **safe-mode** where only a minimal set of device drivers and services are loaded to minimize the chances of encountering startup failure.

11.3.3 Implementation of the Object Manager

The object manager is probably the single most important component in the Windows executive, which is why we have already introduced many of its concepts. As described earlier, it provides a uniform and consistent interface for managing system resources and data structures, such as open files, processes, threads, memory sections, timers, devices, drivers, and semaphores. Even more specialized objects representing things like kernel transactions, profiles, security tokens, and Win32 desktops are managed by the object manager. Device objects link together the descriptions of the I/O system, including providing the link between the NT

namespace and file-system volumes. The configuration manager uses an object of type **key** to link in the registry hives. The object manager itself has objects it uses to manage the NT namespace and implement objects using a common facility. These are directory, symbolic link, and object-type objects.

The uniformity provided by the object manager has various facets. All these objects use the same mechanism for how they are created, destroyed, and accounted for in the quota system. They can all be accessed from user-mode processes using handles. There is a unified convention for managing pointer references to objects from within the kernel. Objects can be given names in the NT namespace (which is managed by the object manager). Dispatcher objects (objects that begin with the common data structure for signaling events) can use common synchronization and notification interfaces, like WaitForMultipleObjects. There is the common security system with ACLs enforced on objects opened by name, and access checks on each use of a handle. There are even facilities to help kernel-mode developers debug problems by tracing the use of objects.

A key to understanding objects is to realize that an (executive) object is just a data structure in the virtual memory accessible to kernel mode. These data structures are commonly used to represent more abstract concepts. As examples, executive file objects are created for each instance of a file-system file that has been opened. Process objects are created to represent each process. Communication objects (e.g., semaphores) are another example.

A consequence of the fact that objects are just kernel data structures is that when the system is rebooted (or crashes) all objects are lost. When the system boots, there are no objects present at all, not even the object-type descriptors. All object types, and the objects themselves, have to be created dynamically by other components of the executive layer by calling the interfaces provided by the object manager. When objects are created and a name is specified, they can later be referenced through the NT namespace. So building up the objects as the system boots also builds the NT namespace.

Objects have a structure, as shown in Fig. 11-15. Each object contains a header with certain information common to all objects of all types. The fields in this header include the object's name, the object directory in which it lives in the NT namespace, and a pointer to a security descriptor representing the ACL for the object.

The memory allocated for objects comes from one of two heaps (or pools) of memory maintained by the executive layer. There are (malloc-like) utility functions in the executive that allow kernel-mode components to allocate either pageable or non-pageable kernel memory. Non-pageable memory is required for any data structure or kernel-mode object that might need to be accessed from a CPU interrupt request level of 2 or more. This includes ISRs and DPCs (but not APCs) and the thread scheduler itself. The page-fault handler and the paging path through the file system and storage drivers also require their data structures to be allocated from non-pageable kernel memory to avoid recursion.

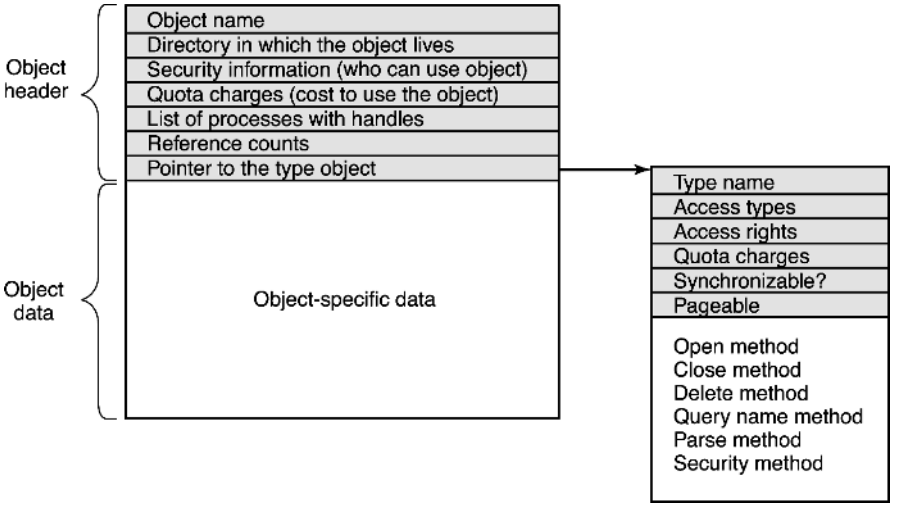

Figure 11-15. Structure of an executive object managed by the object manager.

Most allocations from the kernel heap manager are achieved using per-processor lookaside lists which contain LIFO lists of allocations the same size. These LIFOs are optimized for lock-free operation, improving the performance and scalability of the system.

Each object header contains a quota-charge field, which is the charge levied against a process for opening the object. Quotas are used to keep a user from using too many system resources. On a personal notebook that doesn't matter but on a shared server, it does. There are separate limits for non-pageable kernel memory (which requires allocation of both physical memory and kernel virtual addresses) and pageable kernel memory (which uses up kernel virtual addresses and pagefile space). When the cumulative charges for either memory type hit the quota limit, allocations for that process fail due to insufficient resources. Quotas also are used by the memory manager to control working-set size, and by the thread manager to limit the rate of CPU usage.

Both physical memory and kernel virtual addresses are extremely valuable resources. When an object is no longer needed, it should be deleted and its memory and addresses reclaimed to free up important resources. But it is important that an object should only be deleted when it is no longer in use. In order to correctly track object lifetime, the object manager implements a reference counting mechanism and the concept of a **referenced pointer** which is a pointer to an object whose reference count has been incremented for that pointer. This mechanism prevents premature object deletion when multiple asynchronous operations may be in flight on different threads. Generally, when the last reference to an object is dropped, the object is deleted. It is critical not to delete an object that is in use by some process.

Handles

User-mode references to kernel-mode objects cannot use pointers because they are too difficult to validate and, more importantly, user-mode does not have visibility into kernel-mode address-space layout due to security reasons. Instead, kernel-mode objects must be referred to via an indirection layer. Windows uses **handles** to refer to kernel-mode objects. Handles are opaque values which are converted by the object manager into references to the specific kernel-mode data structure representing an object. Figure 11-16 shows the handle-table data structure used to translate handles into object pointers. The handle table is expandable by adding extra layers of indirection. Each process has its own table, including the system process which contains all the kernel threads not belonging to a user-mode process.

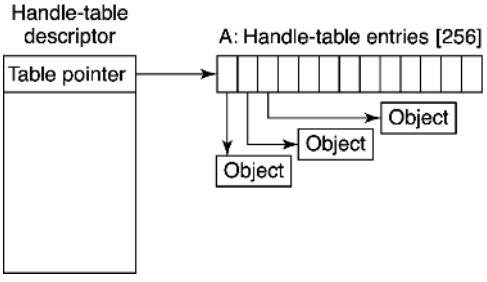

Figure 11-16. Handle table data structures for a minimal table using a single page for up to 512 handles.

Figure 11-17 shows a handle table with two extra levels of indirection, the maximum supported. It is sometimes convenient for code executing in kernel mode to be able to use handles rather than referenced pointers. These are called **kernel handles** and are specially encoded so that they can be distinguished from user-mode handles. Kernel handles are kept in the system processes' handle table and cannot be accessed from user mode. Just as most of the kernel virtual address space is shared across all processes, the system handle table is shared by all kernel components, no matter what the current user-mode process is.

Users can create new objects or open existing objects by making Win32 calls such as CreateSemaphore or OpenSemaphore. These are calls to library procedures that ultimately result in the appropriate system calls being made. The result of any successful call that creates or opens an object is a handle-table entry that is stored in the process' private handle table in kernel memory. The 32-bit index of the handle's logical position in the table is returned to the user to use on subsequent calls. The handle-table entry in the kernel contains a referenced pointer to the object, some flags (e.g., whether the handle should be inherited by child processes), and an access rights mask. The access rights mask is needed because permissions checking is done only at the time the object is created or opened. If a

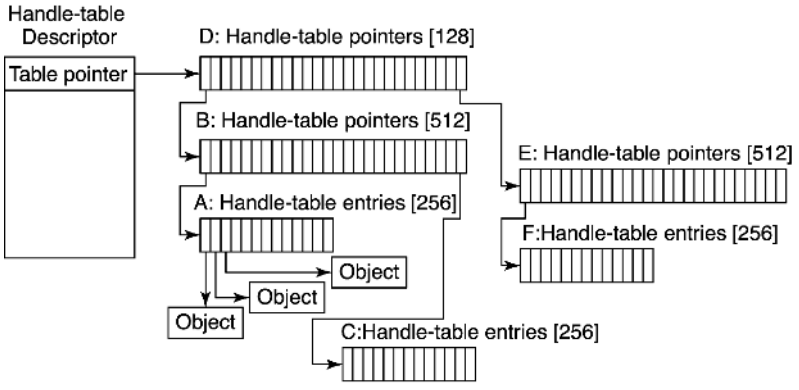

Figure 11-17. Handle-table data structures for a maximal table of up to 16 million handles.

process has only read permission to an object, all the other rights bits in the mask will be 0s, giving the operating system the ability to reject any operation on the object other than reads.

In order to manage lifetime, the object manager keeps a separate handle count in every object. This count is never larger than the referenced pointer count because each valid handle has a referenced pointer to the object in its handle-table entry. The reason for the separate handle count is that many types of objects may need to have their state cleaned up when the last user-mode reference disappears, even though they are not yet ready to have their memory deleted.

One example is file objects, which represent an instance of an opened file. In Windows, files can be opened for exclusive access. When the last handle for a file object is closed, it is important to delete the exclusive access at that point rather than wait for any incidental kernel references to eventually go away (e.g., after the last flush of data from memory). Otherwise closing and reopening a file from user mode may not work as expected because the file still appears to be in use.

Though the object manager has comprehensive mechanisms for managing object lifetimes within the kernel, neither the NT APIs nor the Win32 APIs provide a reference mechanism for dealing with the use of handles across multiple concurrent threads in user mode. Thus, many multithreaded applications have race conditions and bugs where they will close a handle in one thread before they are finished with it in another. Or they may close a handle multiple times, or close a handle that another thread is still using and reopen it to refer to a different object.

Perhaps the Windows APIs should have been designed to require a close API per object type rather than the single generic NtClose operation. That would have at least reduced the frequency of bugs due to user-mode threads closing the wrong handles. Another solution might be to embed a sequence field in each handle in addition to the index into the handle table.

To help application writers find problems like these in their programs, Windows has an **application verifier** that software developers can download from Microsoft. Similar to the verifier for drivers we will describe in Sec. 11.7, the application verifier does extensive rules checking to help programmers find bugs that might not be found by ordinary testing. It can also turn on a FIFO ordering for the handle free list, so that handles are not reused immediately (i.e., turns off the better-performing LIFO ordering normally used for handle tables). Keeping handles from being reused quickly transforms situations where an operation uses the wrong handle into use of a closed handle, which is easy to detect.

The Object Namespace

Processes can share objects by having one process duplicate a handle to the object into the others. But this requires that the duplicating process have handles to the other processes, and is thus impractical in many situations, such as when the processes sharing an object are unrelated, or are protected from each other. In other cases, it is important that objects persist even when they are not being used by any process, such as device objects representing physical devices, or mounted volumes, or the objects used to implement the object manager and the NT namespace itself. To address general sharing and persistence requirements, the object manager allows arbitrary objects to be given names in the NT namespace when they are created. However, it is up to the executive component that manipulates objects of a particular type to provide interfaces that support use of the object manager's naming facilities.

The NT namespace is hierarchical, with the object manager implementing directories and symbolic links. The namespace is also extensible, allowing any object type to specify extensions of the namespace by specifying a **Parse** routine. The *Parse* routine is one of the procedures that can be supplied for each object type when it is created, as shown in Fig. 11-18.

Procedure	When called	Notes
Open	For every new handle	Rarely used
Parse	For object types that extend the namespace	Used for files and registry keys
Close	At last handle close	Clean up visible side effects
Delete	At last pointer dereference	Object is about to be deleted
Security	Get or set object's security descriptor	Protection
QueryName	Get object's name	Rarely used outside kernel

Figure 11-18. Object procedures supplied when specifying a new object type.

The *Open* procedure is rarely used because the default object-manager behavior is usually what is needed and so the procedure is specified as NULL for almost all object types.

The *Close* and *Delete* procedures represent different phases of being done with an object. When the last handle for an object is closed, there may be actions necessary to clean up the state and these are performed by the *Close* procedure. When the final pointer reference is removed from the object, the *Delete* procedure is called so that the object can be prepared to be deleted and have its memory reused. With file objects, both of these procedures are implemented as callbacks into the I/O manager, which is the component that declared the file object type. The object-manager operations result in I/O operations that are sent down the device stack associated with the file object; the file system does most of the work.

The *Parse* procedure is used to open or create objects, like files and registry keys, that extend the NT namespace. When the object manager is attempting to open an object by name and encounters a leaf node in the part of the namespace it manages, it checks to see if the type for the leaf-node object has specified a *Parse* procedure. If so, it invokes the procedure, passing it any unused part of the path name. Again using file objects as an example, the leaf node is a device object representing a particular file-system volume. The *Parse* procedure is implemented by the I/O manager, and results in an I/O operation to the file system to fill in a file object to refer to an open instance of the file that the path name refers to on the volume. We will explore this particular example step-by-step below.

The *QueryName* procedure is used to look up the name associated with an object. The *Security* procedure is used to get, set, or delete the security descriptors on an object. For most object types, this procedure is supplied as a standard entry point in the executive's security reference monitor component.

Note that the procedures in Fig. 11-18 do not perform the most useful operations for each type of object, such as read or write on files (or down and up on semaphores). Rather, the object manager procedures supply the functions needed to correctly set up access to objects and then clean up when the system is finished with them. The objects are made useful by the APIs that operate on the data structures the objects contain. System calls, like NtReadFile and NtWriteFile, use the process' handle table created by the object manager to translate a handle into a referenced pointer on the underlying object, such as a file object, which contains the data that is needed to implement the system calls.

Apart from the object-type callbacks, the object manager also provides a set of generic object routines for operations like creating objects and object types, duplicating handles, getting a referenced pointer from a handle or name, adding and subtracting reference counts to the object header, and NtClose (the generic function that closes all types of handles).

Although the object namespace is crucial to the entire operation of the system, few people know that it even exists because it is not visible to users without special viewing tools. One such viewing tool is *winobj*, available for free at the URL *https://www.microsoft.com/technet/sysinternals*. When run, this tool depicts an object namespace that typically contains the object directories listed in Fig. 11-19 as well as a few others.

Directory	Contents
\GLOBAL??	Starting place for looking up Win32 devices like C:
\Device	All discovered I/O devices
\Driver	Objects corresponding to each loaded device driver
\ObjectTypes	The type objects such as those listed in Fig. 11-21
\Windows	Objects for sending messages to all the Win32 GUI windows
\BaseNamedObjects	User-created Win32 objects such as events, mutexes, etc.
\Sessions	Win32 objects created in the session. Sess. 0 uses \BaseNamedObjects
\Arcname	Partition names discovered by the boot loader
\NLS	National Language Support objects
\FileSystem	File-system driver objects and file system recognizer objects
\Security	Objects belonging to the security system
\KnownDLLs	Key shared libraries that are opened early and held open

Figure 11-19. Some typical directories in the object namespace.

The object manager namespace is not directly exposed through the Win32 API. In fact, Win32 namespace for devices and named objects does not even have a hierarchical structure. This allows the Win32 namespace to be mapped to the object manager namespace in creative ways to provide various application isolation scenarios.

The Win32 namespace for named objects is flat. For example, the CreateEvent function takes an optional object name parameter. This allows multiple applications to open the same underlying Event object and synchronize with one another as long as they agree on the event name, say "MyEvent." The Win32 layer in user-mode (*kernelbase.dll*) determines an object manager directory to place its named objects, called *BaseNamedObjects*. But, where in the object manager namespace should BaseNamedObjects live? If it is stored in a global location, the application sharing scenario is satisfied, but when multiple users are logged onto the machine, application instances in each session may interfere with one another since they expect to be manipulating their own event.

To solve this problem, the Win32 namespace for named objects is *instanced* per user session. Session 0 (where non-interactive OS services run) uses the top-level *\BaseNamedObjects* directory and each interactive session has its own BaseNamedObjects directory underneath the top-level *\Sessions* directory. For example, if a Session 0 service calls CreateEvent with "MyEvent," *kernelbase.dll* redirects it to *\BaseNamedObjects\MyEvent*, but if an application running in interactive Session 2 makes the same call, the event is *\Sessions\2\BaseNamedObjects\MyEvent*.

There may be instances where an application running in an interactive user session needs to share a named event with a Session 0 service. To accommodate that scenario, each session-local BaseNamedObjects directory contains a symbolic link,

called *Global*, pointing to the top-level *BaseNamedObjects* directory. That way, an application can call CreateEvent with "Global\MyEvent" to open *BaseNamedObjects\MyEvent*. Similarly, sometimes a Session 0 service may need to open or create a named object in a particular user session. The BaseNamedObjects directory contains another symbolic link called *Session* which points to *Sessions\BNOLINKS*. That directory, in turn, contains a symbolic link for each active session, pointing to that session's BaseNamedObjects directory. Therefore, a Session 0 process can use the "Session\3\MyEvent" Win32 name to get redirected to *Sessions\3\BaseNamedObjects\MyEvent*.

In the Universal Windows Platform section, we described how UWP apps run in a sandbox called an AppContainer. Namespace isolation for AppContainers is also achieved via BaseNamedObjects mapping. Each session, including Session 0, contains an *AppContainerNamedObjects* directory underneath *Sessions\<ID>*. Each AppContainer has a dedicated directory here for its BaseNamedObjects whose name is derived from the UWP application's package identity. This gives each UWP app its own isolated Win32 namespace. This arrangement also avoids the *namespace squatting* problem where a malicious application creates a named object that it knows its victim will open when it runs. Most Win32 API calls to create named objects will, by default, open the object if it already exists in order to facilitate sharing, but this behavior also allows a squatter to create the object first, even though it might not have had the required permissions to open the object had it been created by the victim application first.

So far we discussed how the Win32 named object namespace is mapped to the global namespace using object manager facilities. The Win32 device namespace also relies on the object manager for proper instancing and isolation. The interestingly named directory *GLOBAL??* shown in Fig. 11-19 contains all Win32 device names, such as *A*: for the floppy disk and *C*: for the first hard disk. These names are actually symbolic links to the *Device* directory where the device objects live. For example, *C*: might be a symbolic link to *Device\HarddiskVolume1*.

Windows allows each user to map Win32 drive letters to devices such as local or remote volumes. Such mappings need to be kept local to that user session to avoid interfering with other users' mappings. This is achieved, again, by instancing the object manager directory containing Win32 devices. Session-local device mappings are stored in the *DosDevices* directory for each session (e.g., *Sessions\1\DosDevices\Z:*). The Win32 layer in user-mode always prepends *??* to paths, indicating that these are Win32 device paths. The object manager has specific handling for items under the *??* directory: it first searches for the item in the session-local *DosDevices* directory associated with the calling process. If the item is not found, then the *GLOBAL??* directory is searched. For example, a CreateFile call for "C:" from a process in Session 2 will result in an NtCreateFile call to *??\C:* and the object manager will check *Sessions\2\DosDevices\CZ:* followed by *GLOBAL??\C:* to find the symbolic link.

Object Types

The device object is one of the most important and versatile kernel-mode objects in the executive. The type is specified by the I/O manager, which, along with the device drivers, are the primary users of device objects. Device objects are closely related to drivers, and each device object usually has a link to a specific driver object, which describes how to access the I/O processing routines for the driver corresponding to the device.

Device objects represent hardware devices, interfaces, and buses, as well as logical disk partitions, disk volumes, and even file systems and kernel extensions like antivirus filters. Many device drivers are given names, so they can be accessed without having to open handles to instances of the devices, as in UNIX. We will use device objects to illustrate how the *Parse* procedure is used, as illustrated in Fig. 11-20:

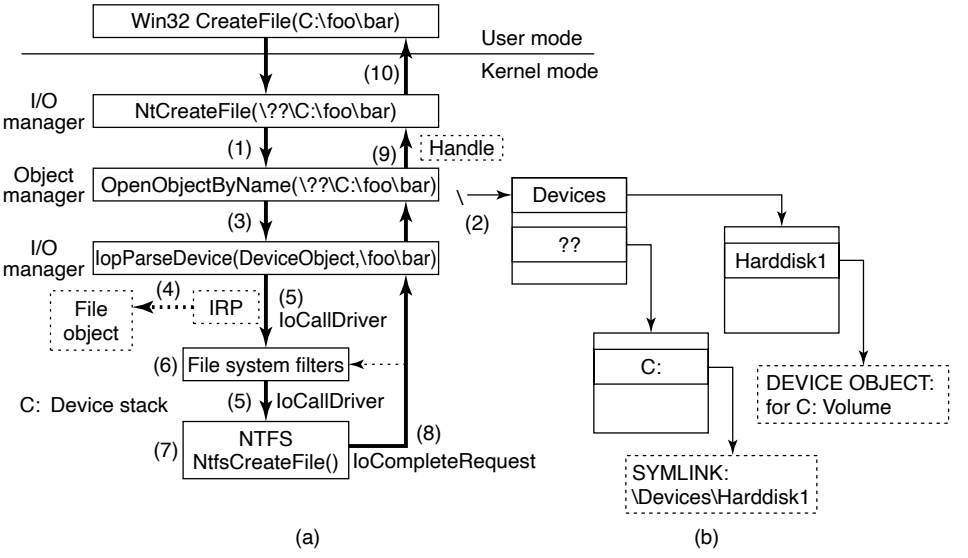

Figure 11-20. I/O and object manager steps for creating/opening a file and getting back a file handle.

1. When an executive component, such as the I/O manager implementing the native system call NtCreateFile, calls ObOpenObjectByName in the object manager, it passes a Unicode path name for the NT namespace, say *\??\C:\foo\bar*.
2. The object manager searches through directories and symbolic links and ultimately finds that *\??\C:* refers to a device object (a type

defined by the I/O manager). The device object is a leaf node in the part of the NT namespace that the object manager manages.

3. The object manager then calls the *Parse* procedure for this object type, which happens to be IopParseDevice implemented by the I/O manager. It passes not only a pointer to the device object it found (for *C:*), but also the remaining string *\foo\bar*.
4. The I/O manager will create an **IRP** (**I/O Request Packet**), allocate a file object, and send the request to the stack of I/O devices determined by the device object found by the object manager.
5. The IRP is passed down the I/O stack until it reaches a device object representing the file-system instance for *C:*. At each stage, control is passed to an entry point into the driver object associated with the device object at that level. The entry point used here is for CREATE operations, since the request is to create or open a file named *\foo\bar* on the volume.
6. The device objects encountered as the IRP heads toward the file system represent file-system filter drivers, which may modify the I/O operation before it reaches the file-system device object. Typically these intermediate devices represent system extensions like antivirus filters.
7. The file-system device object has a link to the file-system driver object, say NTFS. So, the driver object contains the address of the CREATE operation within NTFS.
8. NTFS will fill in the file object and return it to the I/O manager, which returns back up through all the devices on the stack until IopParseDevice returns to the object manager (see Sec. 11.8).
9. The object manager is finished with its namespace lookup. It received back an initialized object from the *Parse* routine (which happens to be a file object—not the original device object it found). So the object manager creates a handle for the file object in the handle table of the current process and returns the handle to its caller.
10. The final step is to return back to the user-mode caller, which in this example is the Win32 API CreateFile, which will return the handle to the application.

Executive components can create new types dynamically, by calling the ObCreateObjectType interface to the object manager. There is no definitive list of object types and they change from release to release. Some of the more common ones in Windows are listed in Fig. 11-21. Let us briefly go over them.

Type	Description
Process	User process
Thread	Thread within a process
Semaphore	Counting semaphore used for interprocess synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
ALPC port	Mechanism for interprocess message passing
Timer	Object allowing a thread to sleep for a fixed time interval
Queue	Object used for completion notification on asynchronous I/O
Open file	Object associated with an open file
Access token	Security descriptor for some object
Profile	Data structure used for profiling CPU usage
Section	Object used for representing mappable files
Key	Registry key, used to attach registry to object-manager namespace
Object directory	Directory for grouping objects within the object manager
Symbolic link	Refers to another object manager object by path name
Device	I/O device object for a physical device, bus, driver, or volume instance
Device driver	Each loaded device driver has its own object

Figure 11-21. Some common executive object types managed by the object manager.

Process and thread are obvious. There is one object for every process and every thread, which holds the main properties needed to manage the process or thread. The next three objects, semaphore, mutex, and event, all deal with interprocess synchronization. Semaphores and mutexes work as expected, but with various extra bells and whistles (e.g., maximum values and timeouts). Events can be in one of two states: signaled or nonsignaled. If a thread waits on an event that is in signaled state, the thread is released immediately. If the event is in nonsignaled state, it blocks until some other thread signals the event, which releases either all blocked threads (notification events) or just the first blocked thread (synchronization events). An event can also be set up so that after a signal has been successfully waited for, it will automatically revert to the nonsignaled state, rather than staying in the signaled state.

Port, timer, and queue objects also relate to communication and synchronization. Ports are channels between processes for exchanging LPC messages. Timers provide a way to block for a specific time interval. Queues (known internally as **KQUEUES**) are used to notify threads that a previously started asynchronous I/O operation has completed or that a port has a message waiting. Queues are designed to manage the level of concurrency in an application, and are also used in high-performance multiprocessor applications, like SQL Server.

Open file objects are created when a file is opened. Files that are not opened do not have objects managed by the object manager. Access tokens are security objects. They identify a user and tell what special privileges the user has, if any. Profiles are structures used for storing periodic samples of the program counter of a running thread to see where the program is spending its time.

Sections are used to represent memory objects backed by files or the pagefile that applications can ask the memory manager to map into their address space. In the Win32 API, these are called *file mapping objects*. Keys represent the mount point for the registry namespace on the object manager namespace. There is usually only one key object, named *\REGISTRY*, which connects the names of the registry keys and values to the NT namespace.

Object directories and symbolic links are entirely local to the part of the NT namespace managed by the object manager. They are similar to their file system counterparts: directories allow related objects to be collected together. Symbolic links allow a name in one part of the object namespace to refer to an object in a different part of the object namespace.

Each device known to the operating system has one or more device objects that contain information about it and are used to refer to the device by the system. Finally, each device driver that has been loaded has a driver object in the object space. The driver objects are shared by all the device objects that represent instances of the devices controlled by those drivers.

Other objects (not shown) have more specialized purposes, such as interacting with kernel transactions, or the Win32 thread pool's worker thread factory.

11.3.4 Subsystems, DLLs, and User-Mode Services

Going back to Fig. 11-4, we see that the Windows operating system consists of components in kernel mode and components in user mode. We have now completed our overview of the kernel-mode components; so it is time to look at the user-mode components, of which three kinds are particularly important to Windows: environment subsystems, DLLs, and service processes.

We have already described the Windows subsystem model; we will not go into more detail now other than to mention that in the original design of NT, subsystems were seen as a way of supporting multiple operating system personalities with the same underlying software running in kernel mode. Perhaps this was an attempt to avoid having operating systems compete for the same platform, as VMS and Berkeley UNIX did on DEC's VAX. Or maybe it was just that nobody at Microsoft knew whether OS/2 would be a success as a programming interface, so they were hedging their bets. In any case, OS/2 became irrelevant, and a latecomer, so the Win32 API designed to be shared with Windows 95, became dominant.

A second key aspect of the user-mode design of Windows is the dynamic link library which is code that is linked to executable programs at run time rather than compile time. Shared libraries are not a new concept, and most modern operating

systems use them. In Windows, almost all libraries are DLLs, from the system library *ntdll.dll* that is loaded into every process to the high-level libraries of common functions that are intended to allow rampant code-reuse by application developers.

DLLs improve the efficiency of the system by allowing common code to be shared among processes, reduce program load times from disk by keeping commonly used code around in memory, and increase the serviceability of the system by allowing operating system library code to be updated without having to recompile or relink all the application programs that use it.

On the other hand, shared libraries introduce the problem of versioning and increase the complexity of the system because changes introduced into a shared library to help one particular program have the potential of exposing latent bugs in other applications, or just breaking them due to changes in the implementation—a problem that in the Windows world is referred to as **DLL hell**.

The implementation of DLLs is simple in concept. Instead of the compiler emitting code that calls directly to subroutines in the same executable image, a level of indirection is introduced: the **IAT** (**Import Address Table**). When an executable is loaded, it is searched for the list of DLLs that must also be loaded (this will be a graph in general, as the listed DLLs will themselves will generally list other DLLs needed in order to run). The required DLLs are loaded and the IAT is filled in for them all.

The reality is more complicated. One problem is that the graphs that represent the relationships between DLLs can contain cycles, or have nondeterministic behaviors, so computing the list of DLLs to load can result in a sequence that does not work. Also, in Windows the DLL libraries are given a chance to run code whenever they are loaded into a process, or when a new thread is created. Generally, this is so they can perform initialization, or allocate per-thread storage, but many DLLs perform a lot of computation in these *attach* routines. If any of the functions called in an *attach* routine needs to examine the list of loaded DLLs, a deadlock can occur, hanging the process. For this reason, these *attach*/*detach* routines must follow strict rules.

DLLs are used for more than just sharing common code. They enable a *hosting* model for extending applications. At the other end of the Internet, Web servers load dynamic code to produce a better Web experience for the pages they display. Applications like Microsoft *Office* link and run DLLs to allow *Office* to be used as a platform for building other applications. The COM (component object model) style of programming allows programs to dynamically find and load code written to provide a particular published interface, which leads to in-process hosting of DLLs by almost all the applications that use COM.

All this dynamic loading of code has resulted in even greater complexity for the operating system, as library version management is not just a matter of matching executables to the right versions of the DLLs, but sometimes loading multiple versions of the same DLL into a process—which Microsoft calls **side-by-side**. A

single program can host two different dynamic code libraries, each of which may want to load the same Windows library—yet have different version requirements for that library.

A better solution would be hosting code in separate processes. But out-of-process hosting of code results has lower performance and makes for a more complicated programming model in many cases. Microsoft has yet to develop a good solution for all of this complexity in user mode. It makes one yearn for the relative simplicity of kernel mode.

One of the reasons that kernel mode has less complexity than user mode is that it supports relatively few extensibility opportunities outside of the device-driver model. In Windows, system functionality is extended by writing user-mode services. This worked well enough for subsystems, and works even better when only a few new services are being provided rather than a complete operating system personality. There are few functional differences between services implemented in the kernel and services implemented in user-mode processes. Both the kernel and process provide private address spaces where data structures can be protected and service requests can be scrutinized.

However, there can be significant performance differences between services in the kernel vs. services in user-mode processes. Entering the kernel from user mode is slow on modern hardware, but not as slow as having to do it twice because you are switching back and forth to another process. Also cross-process communication has lower bandwidth. Unfortunately, the cost of switching between user-mode and kernel-mode has been increasing especially with security mitigations that were implemented against CPU side-channel vulnerabilities like *Spectre* and *Meltdown*, disclosed in 2018.

Kernel-mode code can (carefully) access data at the user-mode addresses passed as parameters to its system calls. With user-mode services, either those data must be copied to the service process, or some games be played by mapping memory back and forth (the ALPC facilities in Windows handle this under the covers).

Windows makes significant use of user-mode service processes to extend the functionality of the system. Some of these services are strongly tied to the operation of kernel-mode components, such as *lsass.exe* which is the local security authentication service which manages the token objects that represent user-identity, as well as managing encryption keys used by the file system. The user-mode plug-and-play manager is responsible for determining the correct driver to use when a new hardware device is encountered, installing it, and telling the kernel to load it. Many facilities provided by third parties, such as antivirus and digital rights management, are implemented as a combination of kernel-mode drivers and user-mode services.

The Windows *taskmgr.exe* has a tab which identifies the services running on the system. Multiple services can be seen to be running in the same process (**svchost.exe**). Windows does this for many of its own boot-time services to reduce the time needed to start up the system and to lower memory usage. Services can be

combined into the same process as long as they can safely operate with the same security credentials.

Within each of the shared service processes, individual services are loaded as DLLs. They normally share a pool of threads using the Win32 thread-pool facility, so that only the minimal number of threads needs to be running across all the resident services.

Services are common sources of security vulnerabilities in the system because they are often accessible remotely (depending on the TCP/IP firewall and IP Security settings) or from unprivileged applications, and not all programmers who write services are as careful as they should be to validate the parameters and buffers that are passed in via RPC. With shared svchosts, a security or a reliability bug, or a memory leak in one service may impact all the other servicing sharing the process as well as making diagnosis more difficult. For these reasons, starting with Windows 10, most Windows services run in their own svchost processes unless the computer is memory-constrained. The few services that still share svchosts either have strong dependencies on being co-located or they make frequent RPC calls to one another which would have significant CPU cost if done across process boundaries.

The number of services running constantly in Windows is staggering. Yet few of those services ever receive a single request, though if they do it is likely to be from an attacker attempting to exploit a vulnerability. As a result more and more services in Windows are turned off by default, particularly on versions of Windows Server.

11.4 PROCESSES AND THREADS IN WINDOWS

Windows has a number of concepts for managing the CPU and grouping resources together. In the following sections, we will examine these, discussing some of the relevant Win32 API calls, and show how they are implemented.

11.4.1 Fundamental Concepts

In Windows, processes are generally containers for programs. They hold the virtual address space, the handles that refer to kernel-mode objects, and threads. In their role as a container for threads, they hold common resources used for thread execution, such as the pointer to the quota structure, the shared token object, and default parameters used to initialize threads—including the priority and scheduling class. Each process has user-mode system data, called the **PEB** (**Process Environment Block**). The PEB includes the list of loaded modules (i.e., the EXE and DLLs), the memory containing environment strings, the current working directory, and data for managing the process' heaps—as well as lots of special-case Win32 cruft that has been added over time.

Threads are the kernel's abstraction for scheduling the CPU in Windows. Priorities are assigned to each thread based on the priority value in the containing process. Threads can also be **affinitized** to run only on certain processors. This helps concurrent programs running on multi-core processors to explicitly spread out work. Each thread has two separate call stacks, one for execution in user mode and one for kernel mode. There is also a **TEB** (**Thread Environment Block**) that keeps user-mode data specific to the thread, including per-thread storage called **TLS** (**Thread Local Storage**), and fields for Win32, language and cultural localization, and other specialized fields that have been added by various facilities.

Besides the PEBs and TEBs, there is another data structure that kernel mode shares with each process, namely, **user shared data**. This is a page that is writable by the kernel, but read-only in every user-mode process. It contains a number of values maintained by the kernel, such as various forms of time, version information, amount of physical memory, and a large number of shared flags used by various user-mode components, such as COM, terminal services, and the debuggers. The use of this read-only shared page is purely a performance optimization, as the values could also be obtained by a system call into kernel mode. But system calls are much more expensive than a single memory access, so for some system-maintained fields, such as the time, this makes a lot of sense. The other fields, such as the current time zone, change infrequently (except on airborne computers), but code that relies on these fields must query them often just to see if they have changed. As with many performance optimizations, it is a bit ugly, but it works.

Processes

The most fundamental component of a process in Windows is its address space. If the process is intended for running a program (and most are), process creation allows a section backed by an executable file on disk to be specified, which gets mapped into the address space and prepared for execution. When a process is created, the creating process receives a handle that allows it to modify the new process by mapping sections, allocating virtual memory, writing parameters and environmental data, duplicating file descriptors into its handle table, and creating threads. This is very different from how processes are created in UNIX and reflects the difference in the target systems for the original designs of UNIX vs. Windows.

As described in Sec. 11.1, UNIX was designed for 16-bit single-processor systems that used swapping to share memory among processes. In such systems, having the process as the unit of concurrency and using an operation like fork to create processes was a brilliant idea. To run a new process with small memory and no virtual memory hardware, processes in memory have to be swapped out to disk to create space. UNIX originally implemented fork simply by swapping out the parent process and handing its physical memory to the child. The operation was almost free. Programmers love things that are free.

In contrast, the hardware environment at the time Cutler's team wrote NT was 32-bit multiprocessor systems with virtual memory hardware to share 1–16 MB of physical memory. Multiprocessors provide the opportunity to run parts of programs concurrently, so NT used processes as containers for sharing memory and object resources, and used threads as the unit of concurrency for scheduling.

Today's systems have 64-bit address spaces, dozens of processing cores and terabytes of RAM. SSDs have displaced rotating magnetic hard disks and virtualization is rampant. So far, Windows' design has held up well as it continued evolving and scaling to keep up with advancing hardware. Future systems are likely to have even more cores, faster and bigger RAM. The difference between memory and storage may start disappearing with **phase-change memories** that retain their contents when powered off, yet very fast to access. Dedicated co-processors are making a comeback to offload operations like memory movement, encryption, and compression to specialized circuits that improve performance and conserve power. Security is more important than ever before and we may start seeing emerging hardware designs based on the **CHERI** (**Capability Hardware Enhanced RISC Instructions**) architecture (Woodruff et al., 2014) with 128-bit capability-based pointers. Windows and UNIX will continue to be adapted to new hardware realities, but what will be really interesting is to see what new operating systems are designed specifically for systems based on these advances.

Jobs and Fibers

Windows can group processes together into jobs. Jobs group processes in order to apply constraints to them and the threads they contain, such as limiting resource use via a shared quota or enforcing a **restricted token** that prevents threads from accessing many system objects. The most significant property of jobs for resource management is that once a process is in a job, all processes' threads in those processes create will also be in the job. There is no escape. As suggested by the name, jobs were designed for situations that are more like batch processing than ordinary interactive computing.

In Windows, jobs are most frequently used to group together the processes that are executing UWP applications. The processes that comprise a running application need to be identified to the operating system so it can manage the entire application on behalf of the user. Management includes setting resource priorities as well as deciding when to suspend, resume, or terminate, all of which happens through job facilities.

Figure 11-22 shows the relationship between jobs, processes, threads, and fibers. Jobs contain processes. Processes contain threads. But threads do not contain fibers. The relationship of threads to fibers is normally many-to-many.

Fibers are cooperatively scheduled user-mode execution contexts which can be switched very quickly without entering kernel mode. As such, they are useful when an application wants to schedule its own execution contexts, minimizing the overhead of thread scheduling by the kernel.

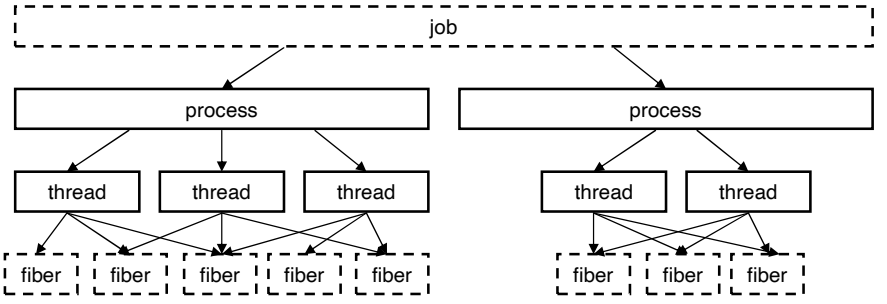

Figure 11-22. The relationship between jobs, processes, threads, and fibers. Jobs and fibers are optional; not all processes are in jobs or contain fibers.

While fibers may sound promising on paper, they face many difficulties in practice. Most of the Win32 libraries are completely unaware of fibers, and applications that attempt to use fibers as if they were threads will encounter various failures. The kernel has no knowledge of fibers, and when a fiber enters the kernel, the thread it is executing on may block and the kernel will schedule an arbitrary thread on the processor, making it unavailable to run other fibers. For these reasons, fibers are rarely used except when porting code from other systems that explicitly need the functionality provided by fibers.

Thread Pools

The Win32 **thread pool** is a facility that builds on top of the Windows thread model to provide a better abstraction for certain types of programs. Thread creation is too expensive to be invoked every time a program wants to execute a small task concurrently with other tasks in order to take advantage of multiple processors. Tasks can be grouped together into larger tasks but this reduces the amount of exploitable concurrency in the program. An alternative approach is for a program to allocate a limited number of threads and maintain a queue of tasks that need to be run. As a thread finishes the execution of a task, it takes another one from the queue. This model separates the resource-management issues (how many processors are available and how many threads should be created) from the programming model (what is a task and how are tasks synchronized). Windows formalizes this solution into the Win32 thread pool, a set of APIs for automatically managing a dynamic pool of threads and dispatching tasks to them.

Thread pools are not a perfect solution, because when a thread blocks for some resource in the middle of a task, the thread cannot switch to a different task. But, the thread pool will inevitably create more threads than there are processors available, so that runnable threads are available to be scheduled even when other threads have blocked. The thread pool is integrated with many of the common synchronization mechanisms, such as awaiting the completion of I/O or blocking until

a kernel event is signaled. Synchronization can be used as triggers for queuing a task so threads are not assigned the task before it is ready to run.

The implementation of the thread pool uses the same queue facility provided for synchronization with I/O completion, together with a kernel-mode thread factory which adds more threads to the process as needed to keep the available number of processors busy. Small tasks exist in many applications, but particularly in those that provide services in the client/server model of computing, where a stream of requests are sent from the clients to the server. Use of a thread pool for these scenarios improves the efficiency of the system by reducing the overhead of creating threads and moving the decisions about how to manage the threads in the pool out of the application and into the operating system.

A summary of CPU execution abstractions is given in Fig. 11-23.

Name	Description	Notes
Job	Collection of processes that share quotas and limits	Used in AppContainers
Process	Container for holding resources	
Thread	Entity scheduled by the kernel	
Fiber	Lightweight thread managed entirely in user space	Rarely used
Thread pool	Task-oriented programming model	Built on top of threads

Figure 11-23. Basic concepts used for CPU and resource management.

Threads

Every process normally starts out with one thread, but new ones can be created dynamically. Threads form the basis of CPU scheduling, as the operating system always selects a thread to run, not a process. Consequently, every thread has a state (ready, running, blocked, etc.), whereas processes do not have scheduling states. Threads can be created dynamically by a Win32 call that specifies the address within the enclosing process' address space at which it is to start running.

Every thread has a thread ID, which is taken from the same space as the process IDs, so a single ID can never be in use for both a process and a thread at the same time. Process and thread IDs are multiples of four because they are actually allocated by the executive using a special handle table set aside for allocating IDs. The system is reusing the scalable handle-management facility illustrated in Figs. 11-16 and 11-17. The handle table does not have references for objects, but does use the pointer field to point at the process or thread so that the lookup of a process or thread by ID is very efficient. FIFO ordering of the list of free handles is turned on for the ID table in recent versions of Windows so that IDs are not immediately reused. The problems with immediate reuse are explored in the problems at the end of this chapter.

A thread normally runs in user mode, but when it makes a system call it switches to kernel mode and continues to run as the same thread with the same properties and limits it had in user mode. Each thread has two stacks, one for use

when it is in user mode and one for use when it is in kernel mode. Whenever a thread enters the kernel, it switches to the kernel-mode stack. The values of the user-mode registers are saved in a **CONTEXT data structure** at the base of the kernel-mode stack. Since the only way for a user-mode thread to not be running is for it to enter the kernel, the CONTEXT for a thread always contains its register state when it is not running. The CONTEXT for each thread can be examined and modified from any process with a handle to the thread.

Threads normally run using the access token of their containing process, but in certain cases related to client/server computing, a thread running in a service process can impersonate its client, using a temporary access token based on the client's token so it can perform operations on the client's behalf. (In general, a service cannot use the client's actual token as the client and server may be running on different systems.)

Threads are also the normal focal point for I/O. Threads block when performing synchronous I/O, and the outstanding I/O request packets for asynchronous I/O are linked to the thread. When a thread is finished executing, it can exit. Any I/O requests pending for the thread will be canceled. When the last thread still active in a process exits, the process terminates.

Please remember that threads are a scheduling concept, not a resource-ownership concept. Any thread is able to access all the objects that belong to its process. All it has to do is use the handle value and make the appropriate Win32 call. There is no restriction on a thread that it cannot access an object because a different thread created or opened it. The system does not even keep track of which thread created which object. Once an object handle has been put in a process' handle table, any thread in the process can use it, even it if is impersonating a different user.

As described previously, in addition to the normal threads that run within user processes Windows has a number of system threads that run only in kernel mode and are not associated with any user process. All such system threads run in a special process called the **system process**. This process has its own user-mode address space which can be used by system threads as necessary. It provides the environment that threads execute in when they are not operating on behalf of a specific user-mode process. We will study some of these threads later when we come to memory management. Some perform administrative tasks, such as writing dirty pages to the disk, while others form the pool of worker threads that are assigned to run specific short-term tasks delegated by executive components or drivers that need to get some work done in the system process.

11.4.2 Job, Process, Thread, and Fiber Management API Calls

New processes are created using the Win32 API function CreateProcess. This function has many parameters and lots of options. It takes the name of the file to be executed, the command-line strings (unparsed), and a pointer to the environment strings. There are also some flags and values that control many details such as how

security is configured for the process and first thread, debugger configuration, and scheduling priorities. A flag also specifies whether open handles in the creator are to be passed to the new process. The function also takes the current working directory for the new process and an optional data structure with information about the GUI Window the process is to use. Rather than returning just a process ID for the new process, Win32 returns both handles and IDs, both for the new process and for its initial thread.

The large number of parameters reveals a number of differences from the design of process creation in UNIX.

1. The actual search path for finding the program to execute is buried in the library code for Win32, but managed more explicitly in UNIX.
2. The current working directory is a kernel-mode concept in UNIX but a user-mode string in Windows. Windows *does* open a handle on the current directory for each process, with the same annoying effect as in UNIX: you cannot delete the directory, unless it happens to be across the network, in which case you *can* delete it.
3. UNIX parses the command line and passes an array of parameters, while Win32 leaves argument parsing up to the individual program. As a consequence, different programs may handle wildcards (e.g., *.txt) and other special symbols in an inconsistent way.
4. Whether file descriptors can be inherited in UNIX is a property of the handle. In Windows, it is a property of both the handle and a parameter to process creation.
5. Win32 is GUI oriented, so new processes are directly passed information about their primary window, while this information is passed as parameters to GUI applications in UNIX.
6. Windows does not have a SETUID bit as a property of the executable, but one process can create a process that runs as a different user, as long as it can obtain a token with that user's credentials.
7. The process and thread handle returned from Windows can be used at any time to modify the new process/thread in many ways, including modifying the virtual memory, injecting threads into the process, and altering the execution of threads. UNIX makes modifications to the new process only between the fork and exec calls, and only in limited ways as exec throws out all the user-mode state of the process.

Some of these differences are historical and philosophical. UNIX was designed to be command-line oriented rather than GUI oriented like Windows. UNIX users are more sophisticated, and they understand concepts like *PATH* variables. Windows inherited a lot of legacy from MS-DOS.

The comparison is also skewed because Win32 is a user-mode wrapper around the native NT process execution, much as the *system* library function wraps fork/exec in UNIX. The actual NT system calls for creating processes and threads, NtCreateProcess and NtCreateThread, are simpler than the Win32 versions. The main parameters to NT process creation are a handle on a section representing the program file to run, a flag specifying whether the new process should, by default, inherit handles from the creator, and parameters related to the security model. All the details of setting up the environment strings and creating the initial thread are left to user-mode code that can use the handle on the new process to manipulate its virtual address space directly.

To support the POSIX subsystem, native process creation has an option to create a new process by copying the virtual address space of another process rather than mapping a section object for a new program. This is used only to implement fork for POSIX, and not exposed by Win32. Since POSIX no longer ships with Windows, process duplication has little use—though sometimes enterprising developers come up with special uses, similar to uses of fork without exec in UNIX. One such interesting usage is process crashdump generation. When a process crashes and a dump needs to be generated, a clone of the address space is created using the native NT process creation API, but without handle duplication. This allows crashdump generation to take its time while the crashing process can be safely restarted without encountering violations, for example due to files still being open by its clone.

Thread creation passes the CPU context to use for the new thread (which includes the stack pointer and initial instruction pointer), a template for the TEB, and a flag saying whether the thread should be immediately run or created in a suspended state (waiting for somebody to call NtResumeThread on its handle). Creation of the user-mode stack and pushing of the *argv*/*argc* parameters is left to user-mode code calling the native NT memory-management APIs on the process handle.

In the Windows Vista release, a new native API for processes, NtCreateUserProcess, was added which moves many of the user-mode steps into the kernel-mode executive and combines process creation with creation of the initial thread. The reason for the change was to support the use of processes as security boundaries. Normally, all processes created by a user are considered to be equally trusted. It is the user, as represented by a token, that determines where the trust boundary is. NtCreateUserProcess allows processes to also provide trust boundaries, but this means that the creating process does not have sufficient rights regarding a new process handle to implement the details of process creation in user mode for processes that are in a different trust environment. The primary use of a process in a different trust boundary (which are called **protected processes**) is to support forms of digital rights management, which protect copyrighted material from being used improperly. Of course, protected processes only target user-mode attacks against protected content and cannot prevent kernel-mode attacks.

Interprocess Communication

Threads can communicate in a wide variety of ways, including pipes, named pipes, mailslots, sockets, remote procedure calls, and shared files. Pipes have two modes: byte and message, selected at creation time. Byte-mode pipes work the same way as in UNIX. Message-mode pipes are somewhat similar but preserve message boundaries, so that four writes of 128 bytes will be read as four 128-byte messages, and not as one 512-byte message, as might happen with byte-mode pipes. Named pipes also exist and have the same two modes as regular pipes. Named pipes can also be used over a network but regular pipes cannot.

Mailslots are a feature of the now-defunct OS/2 operating system implemented in Windows for compatibility. They are similar to pipes in some ways, but not all. For one thing, they are one way, whereas pipes are two way. They could be used over a network but do not provide guaranteed delivery. Finally, they allow the sending process to broadcast a message to many receivers, instead of to just one receiver. Both mailslots and named pipes are implemented as file systems in Windows, rather than executive functions. This allows them to be accessed over the network using the existing remote file-system protocols.

Sockets are like pipes, except that they normally connect processes on different machines. For example, one process writes to a socket and another one on a remote machine reads from it. Sockets can be used on a single machine, but they are less efficient than pipes. Sockets were originally designed for Berkeley UNIX, and the implementation was made widely available. Some of the Berkeley code and data structures are still present in Windows today, as acknowledged in the release notes for the system.

RPCs are a way for process *A* to have process *B* call a procedure in *B*'s address space on *A*'s behalf and return the result to *A*. Various restrictions on the parameters exist. For example, it makes no sense to pass a pointer to a different process, so data structures have to be packaged up and transmitted in a nonprocess-specific way. RPC is normally implemented as an abstraction layer on top of a transport layer. In the case of Windows, the transport can be TCP/IP sockets, named pipes, or ALPC. ALPC is a message-passing facility in the kernel-mode executive. It is optimized for communicating between processes on the local machine and does not operate across the network. The basic design is for sending messages that generate replies, implementing a lightweight version of remote procedure call which the RPC package can build on top of to provide a richer set of features than available in ALPC. ALPC is implemented using a combination of copying parameters and temporary allocation of shared memory, based on the size of the messages.

Finally, processes can share objects. Among these are section objects, which can be mapped into the virtual address space of different processes at the same time. All writes done by one process then appear in the address spaces of the other processes. Using this mechanism, the shared buffer used in producer-consumer problems can easily be implemented.

Synchronization

Processes can also use various types of synchronization objects. Just as Windows provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including events, semaphores, mutexes, and various user-mode primitives. All of these mechanisms work with threads, not processes, so that when a thread blocks on a semaphore, other threads in that process (if any) are not affected and can continue to run.

One of the most fundamental synchronization primitives exposed by the kernel is the **Event**. Events are kernel-mode objects and thus have security descriptors and handles. Event handles can be duplicated using DuplicateHandle and passed to another process so that multiple processes can synchronize on the same event. An event can also be given a name in the Win32 namespace and have an ACL set to protect it. Sometimes sharing an event by name is more appropriate than duplicating the handle.

As we have described previously, there are two kinds of events: **notification events** and **synchronization events**. An event can be in one of two states: signaled or not-signaled. A thread can wait for an event to be signaled with WaitForSingleObject. If another thread signals an event with SetEvent, what happens depends on the type of event. With a notification event, all waiting threads are released and the event stays set until manually cleared with ResetEvent. With a synchronization event, if one or more threads are waiting, exactly one thread is released and the event is cleared. An alternative operation is PulseEvent, which is like SetEvent except that if nobody is waiting, the pulse is lost and the event is cleared. In contrast, a SetEvent that occurs with no waiting threads is remembered by leaving the event in the signaled state so a subsequent thread that calls a wait API for the event will not actually wait.

Semaphores can be created using the CreateSemaphore Win32 API function, which can also initialize it to a given value and define a maximum value as well. Like events, semaphores are also kernel-mode objects. Calls for up and down exist, although they have the somewhat odd names of ReleaseSemaphore (up) and WaitForSingleObject (down). It is also possible to give WaitForSingleObject a timeout, so the calling thread can be released eventually, even if the semaphore remains at 0. WaitForSingleObject and WaitForMultipleObjects are the common interfaces used for waiting on the dispatcher objects discussed in Sec. 11.3. While it would have been possible to wrap the single-object version of these APIs in a wrapper with a somewhat more semaphore-friendly name, many threads use the multiple-object version which may include waiting for multiple flavors of synchronization objects as well as other events like process or thread termination, I/O completion, and messages being available on sockets and ports.

Mutexes are also kernel-mode objects used for synchronization, but simpler than semaphores because they do not have counters. They are essentially locks, with API functions for locking WaitForSingleObject and unlocking ReleaseMutex.

Like semaphore handles, mutex handles can be duplicated and passed between processes so that threads in different processes can access the same mutex.

Another synchronization mechanism is called **Critical Sections**, which implement the concept of critical regions. These are similar to mutexes in Windows, except local to the address space of the creating thread. Because critical sections are not kernel-mode objects, they do not have explicit handles or security descriptors and cannot be passed between processes. Locking and unlocking are done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed initially in user space and make kernel calls only when blocking is needed, they are much faster than mutexes. Critical sections are optimized to combine spin locks (on multiprocessors) with the use of kernel synchronization only when necessary. In many applications, most critical sections are so rarely contended or have such short hold times that it is never necessary to allocate a kernel synchronization object. This results in a very significant saving in kernel memory.

SRW locks (**Slim Reader-Writer locks**) are another type of process-local lock implemented in user-mode like critical sections, but they support both exclusive and shared acquisition via the AcquireSRWLockExclusive and AcquireSRWLockShared APIs and the corresponding release functions. When the lock is held shared, if an exclusive acquire arrives (and starts waiting), subsequent shared acquire attempts block to avoid starving exclusive waiters. A big advantage of SRW locks is that they are the size of a pointer which allows them to be used for granular synchronization of small data structures. Unlike critical sections, SRW locks do not support recursive acquisition which is generally not a good idea anyway.

Sometimes applications need to check some state protected by a lock and wait until a condition is satisfied in a synchronized way. Examples are producer-consumer or bounded buffer problems. Windows provides **Condition variables** for these situations. They allow the caller to atomically release a lock, either a critical section or an SRW lock, and enter a sleeping state using SleepConditionVariableCS and SleepConditionVariableSRW APIs. A thread changing the state can wake any waiters via WakeConditionVariable or WakeAllConditionVariable.

Two other useful user-mode synchronization primitives provided by Windows are WaitOnAddress and InitOnceExecuteOnce. WaitOnAddress is called to wait for the value at the specified address to be modified. The application must call either WakeByAddressSingle (or WakeByAddressAll) after modifying the location to wake either the first (or all) of the threads that called WaitOnAddress on that location. The advantage of this API over using events is that it is not necessary to allocate an explicit event for synchronization. Instead, the system hashes the address of the location to find a list of all the waiters for changes to a given address. WaitOnAddress functions similar to the sleep/wakeup mechanism found in the UNIX kernel. Critical sections mentioned earlier actually use the WaitOnAddress primitive for its implementation. InitOnceExecuteOnce can be used to ensure that an initialization routine is run exactly one time in a program. Correct

initialization of data structures is surprisingly hard in multithreaded programs and this primitive provides a very simple way to ensure correctness and high-performance.

So far, we discussed the most popular synchronization mechanisms provided by Windows to user-mode programs. There are many more primitives exposed to kernel-mode callers. Some examples are **EResources** which are reader-writer locks typically used by the file system stack which support unusual scenarios such as cross-thread lock ownership transfer. **FastMutex** is an exclusive lock similar to a critical section and **PushLocks** are the kernel-mode analogue of SRW locks. A high-performance variant of pushlocks, called the **Cache-aware PushLock**, is implemented to provide scalability even on machines with hundreds of processor cores. A cache-aware pushlock is composed of many pushlocks, one for each processor (or small groups of processors). It is targeted at scenarios where exclusive acquires are rare. Shared acquires only acquire the local pushlock associated with the processor while exclusive acquires must acquire every pushlock. Only acquiring a local lock in the common case results in much more efficient processor cache behavior especially on multi-NUMA machines. While the cache-aware pushlock is great for scalability, it does have a large memory cost and is therefore not always appropriate to use for small, multiplicative data structures. The **Auto-expand PushLock** provides a good compromise: it starts out as a single pushlock, taking up only two pointers worth of space, but automatically "expands" to become a cache-aware pushlock when it detects a high degree of cache contention due to concurrent shared acquires.

A summary of these synchronization primitives is given in Fig. 11-24.

Primitive	Kernel object	Kernel/User	Shared/Exclusive
Event	Yes	Both	N/A
Semaphore	Yes	Both	N/A
Mutex	Yes	Both	Exclusive
Critical Section	No	User-mode	Exclusive
SRW Lock	No	User-mode	Shared
Condition Variable	No	User-mode	N/A
InitOnce	No	User-mode	N/A
WaitOnAddress	No	User-mode	N/A
EResource	No	Kernel-mode	Shared
FastMutex	No	Kernel-mode	Exclusive
PushLock	No	Kernel-mode	Shared
Cache-aware PushLock	No	Kernel-mode	Shared
Auto-expand PushLock	No	Kernel-mode	Shared

Figure 11-24. Summary of synchronization primitives provided by Windows.

11.4.3 Implementation of Processes and Threads

In this section, we will get into more detail about how Windows creates a process (and the initial thread). Because Win32 is the most documented interface, we will start there. But we will quickly work our way down into the kernel and understand the implementation of the native API call for creating a new process. We will focus on the main code paths that get executed whenever processes are created, as well as look at a few of the details that fill in gaps in what we have covered so far.

A process is created when another process makes the Win32 CreateProcess call. This call invokes a user-mode procedure in *kernelbase.dll* that makes a call to NtCreateUserProcess in the kernel to create the process in several steps.

1. Convert the executable file name given as a parameter from a Win32 path name to an NT path name. If the executable has just a name without a directory path name, it is searched for in the directories listed in the default directories (which include, but are not limited to, those in the PATH variable in the environment).
2. Bundle up the process-creation parameters and pass them, along with the full path name of the executable program, to the native API NtCreateUserProcess.
3. Running in kernel mode, NtCreateUserProcess processes the parameters, then opens the program image and creates a section object that can be used to map the program into the new process' virtual address space.
4. The process manager allocates and initializes the process object (the kernel data structure representing a process to both the kernel and executive layers).
5. The memory manager creates the address space for the new process by allocating and initializing the page directories and the virtual address descriptors which describe the kernel-mode portion, including the process-specific regions, such as the **self-map** page-directory entries that gives each process kernel-mode access to the physical pages in its entire page table using kernel virtual addresses. (We will describe the self map in more detail in Sec. 11.5.)
6. A handle table is created for the new process, and all the handles from the caller that are allowed to be inherited are duplicated into it.
7. The shared user page is mapped, and the memory manager initializes the working-set data structures used for deciding what pages to trim from a process when physical memory is low. The executable image

represented by the section object are mapped into the new process' user-mode address space.

8. The executive creates and initializes the user-mode PEB, which is used by both user mode processes and the kernel to maintain processwide state information, such as the user-mode heap pointers and the list of loaded libraries (DLLs).
9. Virtual memory is allocated in the new process and used to pass parameters, including the environment strings and command line.
10. A process ID is allocated from the special handle table (ID table) the kernel maintains for efficiently allocating locally unique IDs for processes and threads.
11. A thread object is allocated and initialized. A user-mode stack is allocated along with the Thread Environment Block. The *CONTEXT* record which contains the thread's initial values for the CPU registers (including the instruction and stack pointers) is initialized.
12. The process object is added to the global list of processes. Handles for the process and thread objects are allocated in the caller's handle table. An ID for the initial thread is allocated from the ID table.
13. NtCreateUserProcess returns to user mode with the new process created, containing a single thread that is ready to run but suspended.
14. If the NT API fails, the Win32 code checks to see if this might be a process belonging to another subsystem like WoW64. Or perhaps the program is marked that it should be run under the debugger. These special cases are handled with special code in the user-mode CreateProcess code.
15. If NtCreateUserProcess was successful, there is still some work to be done. Win32 processes have to be registered with the Win32 subsystem process, *csrss.exe*. *Kernelbase.dll* sends a message to *csrss* telling it about the new process along with the process and thread handles so it can duplicate itself. The process and threads are entered into the subsystems' tables so that they have a complete list of all Win32 processes and threads. The subsystem then displays a cursor containing a pointer with an hourglass to tell the user that something is going on but that the cursor can be used in the meanwhile. When the process makes its first GUI call, usually to create a window, the cursor is removed (it times out after 2 seconds if no call is forthcoming).
16. If the process is restricted, such as low-rights Internet browser, the token is modified to restrict what objects the new process can access.

17. If the application program was marked as needing to be shimmed to run compatibly with the current version of Windows, the specified *shims* are applied. **Shims** usually wrap library calls to slightly modify their behavior, such as returning a fake version number or delaying the freeing of memory to work around bugs in applications.

18. Finally, call NtResumeThread to unsuspend the thread and return the structure to the caller containing the IDs and handles for the process and thread that were just created.

In earlier versions of Windows, much of the algorithm for process creation was implemented in the user-mode procedure which would create a new process in using multiple system calls and by performing other work using the NT native APIs that support implementation of subsystems. These steps were moved into the kernel to reduce the ability of the parent process to manipulate the child process in the cases where the child is running a protected program, such as one that implements DRM to protect movies from piracy.

The original native API, NtCreateProcess, is still supported by the system, so much of process creation could still be done within user mode of the parent process—as long as the process being created is not a protected process.

Generally, when kernel-mode component need to map files or allocate memory in a user-mode address space, they can use the system process. However, sometimes a dedicated address space is desired for better isolation since the system process user-mode address space is accessible to all kernel-mode entities. For such needs, Windows supports the concept of a **Minimal Process**. A minimal process is just an address space; its creation skips over most of the steps described above since it is not intended for execution. It has no shared user page, or a PEB, or any user-mode threads. No DLLs are mapped in its address space; it is entirely empty at creation. And it certainly does not register with the Win32 subsystem. In fact, minimal processes are only exposed to operating system kernel components; not even drivers. Some examples of kernel components that use minimal processes are listed below:

1. *Registry*: The registry creates a minimal process called "Registry" and maps its registry hives into the user-mode address space of the process. This protects the hive data from potential corruption due to bugs in drivers.

2. *Memory Compression*: The memory compression component uses a minimal process called "Memory Compression" to hold its compressed data. Just like the registry, the goal is to avoid corruption. Also, having its own process allows setting of per-process policies like working set limits. We will discuss memory compression in more detail in Sec. 11.5.

3. *Memory Partitions*: A memory partition represents a subset of memory with its own isolated instance of memory management. It is used for subdividing memory for dedicated purposes and to run isolated workloads which should not interfere with one another due to memory management mechanisms. Each memory partition comes with its minimal system process, called "PartitionSystem," into which the memory manager can map executables that are being loaded in that partition. We will cover memory partitions in Sec. 11.5.

Scheduling

The Windows kernel does not use a central scheduling thread. Instead, when a thread cannot run any more, the thread is directed into the scheduler to see which thread to switch to. The following conditions invoke scheduling.

1. A running thread blocks on an I/O, lock, event, semaphore, etc.
2. The thread signals an object (e.g., calls SetEvent on an event).
3. The quantum expires.

In case 1, the thread is already in the kernel to carry out the operation on the dispatcher or I/O object. It cannot possibly continue, so it calls the scheduler code to pick its successor and load that thread's CONTEXT record to resume running it.

In case 2, the running thread is in the kernel, too. However, after signaling some object, it can definitely continue because signaling an object never blocks. Still, the thread is required to call the scheduler to see if the result of its action has readied a thread with a higher scheduling priority that is now ready to run. If so, a thread switch occurs since Windows is fully preemptive (i.e., thread switches can occur at any moment, not just at the end of the current thread's quantum). However, if multiple CPUs are present, a thread that was made ready may be scheduled on a different CPU and the original thread can continue to execute on the current CPU even though its scheduling priority is lower.

In case 3, an interrupt to kernel mode occurs, at which point the thread executes the scheduler code to see who runs next. Depending on what other threads are waiting, the same thread may be selected, in which case it gets a new quantum and continues running. Otherwise a thread switch happens.

The scheduler is also called under two other conditions:

1. An I/O operation completes.
2. A timed wait expires.

In the first case, a thread may have been waiting on this I/O and is now released to run. A check has to be made to see if it should preempt the running thread since there is no guaranteed minimum run time. The scheduler is not run in the interrupt

handler itself (since that may keep interrupts turned off too long). Instead, a DPC is queued for slightly later, after the interrupt handler is done. In the second case, a thread has done a down on a semaphore or blocked on some other object, but with a timeout that has now expired. Again it is necessary for the interrupt handler to queue a DPC to avoid having it run during the clock interrupt handler. If a thread has been made ready by this timeout, the scheduler will be run. If the newly runnable thread has higher priority, the current thread is preempted as in case 1.

Now we come to the actual scheduling algorithm. The Win32 API provides two APIs to influence thread scheduling. First, there is a call SetPriorityClass that sets the priority class of all the threads in the caller's process. The allowed values are real-time, high, above normal, normal, below normal, and idle. The priority class determines the relative priorities of processes. The process priority class can also be used by a process to temporarily mark itself as being *background*, meaning that it should not interfere with any other activity in the system. Note that the priority class is established for the process, but it affects the actual priority of every thread in the process by setting a base priority that each thread starts with when created.

The second Win32 API is SetThreadPriority. It sets the relative priority of a thread (possibly, but not necessarily, the calling thread) with respect to the priority class of its process. The allowed values are time critical, highest, above normal, normal, below normal, lowest, and idle. Time-critical threads get the highest nonreal-time scheduling priority, while idle threads get the lowest, irrespective of the priority class. The other priority values adjust the base priority of a thread with respect to the normal value determined by the priority class (+2, +1, 0, −1, −2, respectively). The use of priority classes and relative thread priorities makes it easier for applications to decide what priorities to specify.

The scheduler works as follows. The system has 32 priorities, numbered from 0 to 31. The combinations of priority class and relative priority are mapped onto 32 absolute thread priorities according to the table of Fig. 11-25. The number in the table determines the thread's **base priority**. In addition, every thread has a **current priority**, which may be higher (but not lower) than the base priority and which we will discuss shortly.

To use these priorities for scheduling, the system maintains an array of 32 lists of threads, corresponding to priorities 0 through 31 derived from the table of Fig. 11-25. Each list contains ready threads at the corresponding priority. The basic scheduling algorithm consists of searching the array from priority 31 down to priority 0. As soon as a nonempty list is found, the thread at the head of the queue is selected and run for one quantum. If the quantum expires, the thread goes to the end of the queue at its priority level and the thread at the front is chosen next. In other words, when there are multiple threads ready at the highest priority level, they run round robin for one quantum each. If no thread is ready, the idle thread is selected for execution in order to idle the processor—that is, set it to a low power state waiting for an interrupt to occur.

		Win32 process class priorities					
		Real-time	High	Above normal	Normal	Below normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Figure 11-25. Mapping of Win32 priorities to Windows priorities.

It should be noted that scheduling is done by picking a thread without regard to which process that thread belongs. Thus, the scheduler does *not* first pick a process and then pick a thread in that process. It only looks at the threads. It does not consider which thread belongs to which process except to determine if it also needs to switch address spaces when switching threads.

To improve the scalability of the scheduling algorithm for multiprocessors with a high number of processors, the scheduler partitions the global set of ready threads into multiple separate ready queues each with its own array of 32 lists. These ready queues exist in two forms, processor local ready queues that are associated with a single processor and shared ready queues that are associated with groups of processors. A thread is only eligible to be placed into a shared ready queue if it is capable of running on all processors associated with the queue. When a processor needs to select a new thread to run due to a thread blocking, it will first consult the ready queues to which it is associated and only consult ready queues associated with other processors if no candidate threads could be found locally.

As an additional improvement, the scheduler tries hard not to have to take the locks that protect access to the ready queue lists. Instead, it sees if it can directly dispatch a thread that is ready to run to the processor where it should run rather than add it to a ready queue.

Some multiprocessor systems have complex memory topologies where CPUs have their own local memory and while they can execute programs and access data out of other processors memory, this comes at a performance cost. These systems are called NUMA (NonUniform Memory Access) machines. Additionally, some multiprocessor systems have complex cache hierarchies where only some of the processor cores in a physical CPU share a last-level cache. The scheduler is aware of these complex topologies and tries to optimize thread placement by assigning each thread an ideal processor. The scheduler then tries to schedule each thread to a processor that is as close *topologically* to its ideal processor as possible. If a thread cannot be scheduled to a processor immediately, then it will be placed in a

ready queue associated with its ideal processor, preferably the shared ready queue. However, if the thread is incapable of running on some processors associated with that queue, for example due to an affinity restriction, it will be placed in the ideal processor's local ready queue. The memory manager also uses the ideal processor to determine which physical pages should be allocated to satisfy page faults, preferring to choose pages from the NUMA node belonging to the faulting thread's ideal processor.

The array of queue headers is shown in Fig. 11-26. The figure shows that there are actually four categories of priorities: real-time, user, zero, and idle, which is effectively −1. These deserve some comment. Priorities 16–31 are called system, and are intended to build systems that satisfy real-time constraints, such as deadlines needed for multimedia presentations. Threads with real-time priorities run before any of the threads with dynamic priorities, but not before DPCs and ISRs. If a real-time application wants to run on the system, it may require device drivers that are careful not to run DPCs or ISRs for any extended time as they might cause the real-time threads to miss their deadlines.

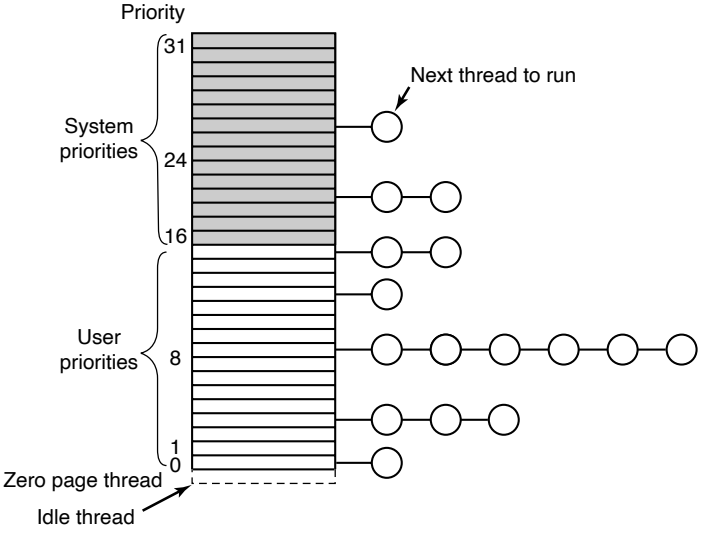

Figure 11-26. Windows supports 32 priorities for threads.

Ordinary users may not create real-time threads. If a user thread ran at a higher priority than, say, the keyboard or mouse thread and got into a loop, the keyboard or mouse thread would never run, effectively hanging the system. The right to set the priority class to real-time requires a special privilege to be enabled in the process' token. Normal users do not have this privilege.

Application threads normally run at priorities 1–15. By setting the process and thread priorities, an application can determine which threads get preference. The

ZeroPage system threads run at priority 0 and convert free pages into pages of all zeroes. There is a separate ZeroPage thread for each real processor.

Each thread has a base priority based on the priority class of the process and the relative priority of the thread. But the priority used for determining which of the 32 lists a ready thread is queued on is determined by its current priority, which is normally the same as the base priority—but not always. Under certain conditions, the current priority of a thread is adjusted by the kernel above its base priority. Since the array of Fig. 11-26 is based on the current priority, changing this priority affects scheduling. These priority adjustments can be classified into two types: priority boost and priority floors.

First let us discuss **priority boosts**. Boosts are temporary adjustments to thread priority and are generally applied when a thread enters the ready state. For example, when an I/O operation completes and releases a waiting thread, the priority is boosted to give it a chance to run again quickly and start more I/O. The idea here is to keep the I/O devices busy. The amount of boost depends on the I/O device, typically 1 for a disk, 2 for a serial line, 6 for the keyboard, and 8 for the sound card.

Similarly, if a thread was waiting on a semaphore, mutex, or other event, when it is released, it gets boosted by two levels if it is in the foreground process (the process controlling the window to which keyboard input is sent) and one level otherwise. This fix tends to raise interactive processes above the big crowd at level 8. Finally, if a GUI thread wakes up because window input is now available, it gets a boost for the same reason.

These boosts are not forever. They take effect immediately and can cause rescheduling of the CPU. But if a thread uses all of its next quantum, it loses one priority level and moves down one queue in the priority array. If it uses up another full quantum, it moves down another level, and so on until it hits its base level, where it remains until it is boosted again. A thread cannot be boosted into or within the real-time priority range, non-realtime threads can be boosted to at most a priority of 15 and realtime threads cannot be boosted at all.

The second class of priority adjustment is a **priority floor**. Unlike boosts which apply an adjustment relative to a thread's base priority, priority floors apply a constraint that a thread's absolute current priority must never fall below a given minimum priority. This constraint is not linked to the thread quantum and persists until explicitly removed.

One case in which priority floors are used is illustrated in Fig. 11-27. Imagine that on a single processor machine, a thread T1 running in kernel-mode at priority 4 gets preempted by a priority 8 thread T2 after acquiring a pushlock. Then, a priority 12 thread T3 arrives, preempts T2 and blocks trying to acquire the pushlock held by T1. At this point, both T1 and T2 are runnable, but T2 has higher priority, so it continues running even though it is effectively preventing T3, a higher priority thread, from making progress because T1 is not able to run to release the pushlock. This situation is a very well-known problem called **priority inversion**. Windows

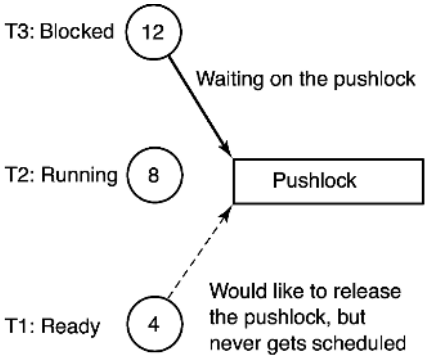

Figure 11-27. An example of priority inversion.

addresses priority inversion between kernel threads through a facility in the thread scheduler called **Autoboost.** Autoboost automatically tracks resource dependencies between threads and applies priority floors to raise the scheduling priority of threads that hold resources needed by higher-priority threads. In this case, Autoboost would determine that the owner of the pushlock needs to be raised to the maximum priority of the waiters, so it would apply a priority floor of 12 to T1 until it releases the lock, thus resolving the inversion.

In some multiprocessor systems, there are multiple types of processors with varying performance and efficiency characteristics. On these systems with *heterogeneous processors*, the scheduler must take these varying performance and efficiency characteristics into account in order to make optimal scheduling decisions. The Windows kernel does this through a thread scheduling property called the **QoS class** (**Quality-of-Service class**), which classifies threads based on their importance to the user and their performance requirements. Windows defines six QoS classes: high, medium, low, eco, multimedia, and deadline. In general, threads with a higher QoS class are threads that are more important to the user and thus require higher performance, for example threads that belong to a process that is in the foreground. Threads with a lower QoS class are threads that are less important and favor efficiency over performance, for example threads performing background maintenance work. Classification of threads into QoS levels is done by the scheduler through a number of heuristics considering attributes such as whether a thread belongs to a process with a foreground window or belongs to a process that is playing audio. Applications can also provide explicit process and thread level hints about their importance via the SetProcessInformation and SetThreadInformation Win32 APIs. From the thread's QoS class, the scheduler derives several more specific scheduling properties based on the system's power policy.

Firstly, the system's power policy can be configured to restrict work to a particular type of processor. For example, the system can be configured to allow low QoS work to run only on the most efficient processors in the system in order to

achieve maximum efficiency for this work at the expense of performance. These restrictions are considered by the scheduler when deciding which processor a thread should be scheduled to.

Secondly, a thread's QoS determines whether it prefers scheduling for performance or efficiency. The scheduler maintains two rankings of the system's processors: one in order of performance and another in order of efficiency. System power policy determines which of these orderings should be used by the scheduler for each QoS class when it is searching for an idle processor upon which to run a thread.

Finally, a thread's QoS determines how important a thread's desire for performance or efficiency is relative to other threads of differing QoS. This importance ordering is used to determine which threads get access to the more performant processors in the system when the more performant cores are over utilized. Note that this is different from a thread's priority in that thread priority determines the set of threads that will run at a given point in time whereas importance controls which of the threads out of that set will be given their preferred placement. This is accomplished via a scheduler policy referred to as core trading. If a thread that prefers performance is being scheduled and the scheduler is unable to find an idle high-performance processor, but is able to locate a low-performance processor, the scheduler will check whether one of the high-performance processors is running a lower importance thread. If so, it will swap the processor assignments to place the higher importance thread on the more performant processor and place the lower importance thread on the less performant processor.

Windows runs on PCs, which usually have only a single interactive session active at a time. However, Windows also supports a **terminal server** mode which supports multiple interactive sessions over the network using the remote desktop protocol. When running multiple user sessions, it is easy for one user to interfere with another by consuming too much processor resources. Windows implements a fair-share algorithm, **DFSS** (**Dynamic Fair-Share Scheduling**), which keeps sessions from running excessively. DFSS uses **scheduling groups** to organize the threads in each session. Within each group, the threads are scheduled according to normal Windows scheduling policies, but each group is given more or less access to the processors based on how much the group has been running in aggregate. The relative priorities of the groups are adjusted slowly to allow ignore short bursts of activity and reduce the amount a group is allowed to run only if it uses excessive processor time over long periods.

11.4.4 WoW64 and Emulation

Application compatibility has always been the hallmark of Windows to maintain and grow its user and developer base. As hardware evolves and Windows gets ported to new processor architectures, retaining the ability to run existing software has consistently been important for customers (and therefore Microsoft). For this

reason, the 64-bit version of Windows XP, released in 2001, included WoW64 (Windows-on-Windows), an emulation layer for running unmodified 32-bit applications on 64-bit Windows. OriginalO, WoW64 only ran 32-bit x86 applications on IA-64 and then x64, but Windows 10 further expanded the scope of WoW64 to run 32-bit ARM applications as well as x86 applications on arm64.

WoW64 Design

At its heart, WoW64 is a paravirtualization layer which makes the 32-bit application believe that it is running on a 32-bit system. In this context, the 32-bit architecture is called the *guest* and the 64-bit OS is the *host*. Such virtualization could have been done by using a virtual machine with full 32-bit Windows running in it. In fact, Windows 7 had a feature called *XP Mode* which did exactly that. However, virtual machine-based approaches are much more expensive due to the memory and CPU overhead of running two operating systems. Also hiding all the seams between the operating systems and making the user feel like she's using a single operating system is difficult. Instead, WoW64 emulates a 32-bit system at the system call layer, in user-mode. The application and all of its 32-bit dependencies load and run normally. Their system calls are redirected to the WoW64 layer which converts them to 64-bit and makes the actual system call through the host *ntdll.dll*. This essentially eliminates all overhead and the 64-bit kernel-mode code is largely unaware of the 32-bit emulation; it runs just like any other process.

Figure 11-28 shows the composition of a WoW64 process and the WoW64 layers compared to a native 64-bit process. WoW64 processes contain both 32-bit code for the guest (composed of application and 32-bit OS binaries) and 64-bit native code for the WoW64 layer and *ntdll.dll*. At process creation time, the kernel prepares the address space similar to what a 32-bit OS would. 32-bit versions of data structures such as PEB and TEB are created and the 32-bit WoW64-aware *ntdll.dll* is mapped into the process along with the 32-bit application executable. Each thread has a 32-bit stack and a 64-bit stack which are switched when transitioning between the two layers (much like how entering kernel-mode switches to the thread's kernel stack and back). All 32-bit components and data structures use the low 4 GB of the process address space so all addresses fit within guest pointers.

Native layer sits underneath the guest code and is composed of WoW64 DLLs as well as the native *ntdll.dll* and the normal 64-bit PEB and TEBs. This layer effectively acts as the 32-bit kernel for the guest. There are two categories of WoW64 DLLs: the **WoW64 abstraction layer** (wow64.dll, *wow64base.dll* and *wow64win.dll*) and the **CPU emulation layer**. The WoW64 abstraction layer is largely platform-independent and acts as the **thunk layer**, which receives 32-bit system calls and converts them to 64-bit calls, accounting for differences in types and structure layout. Some of the simpler system calls which do not need extensive type conversion go through an optimized path called **Turbo Thunks** in the CPU emulation layer to make direct system calls into the kernel. Otherwise, *wow64.dll*

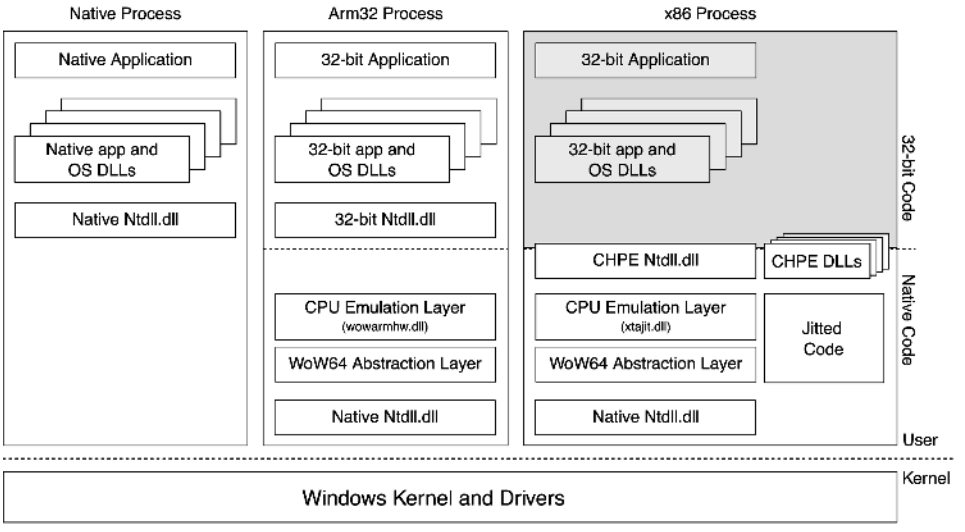

Figure 11-28. Native vs. WoW64 processes on an arm64 machine. Shaded areas indicate emulated code.

handles NT system calls and *wow64win.dll* handles system calls that land in win32k.sys. Exception dispatching is also conducted by this layer which translates the 64-bit exception record generated by the kernel to 32-bit and dispatches to the guest *ntdll*.dll. Finally, the WoW64 abstraction layer performs the **namespace redirection** necessary for 32-bit applications. For example, when a 32-bit application accesses *c:\Windows\System32*, it is redirected to *c:\Windows\SysWoW64* or *c:\Windows\SysArm32* as appropriate. Similarly, some registry paths, for example those under the SOFTWARE hive, are redirected to a subkey called *WoW6432Node* or *WoWAA32Node*, for x64 or arm64, respectively. That way, if the 32-bit and the 64-bit versions of the same component run, they do not overwrite each other's registry state.

The WoW64 CPU emulation layer is very much architecture dependent. Its job is to execute the machine code for the guest architecture. In many cases, the host CPU can actually execute guest instructions after going through a mode switch. So, when running x86 code on x64 or arm32 code on arm64, the CPU emulation layer only needs to switch the CPU mode and start running guest code. That's what *wow64cpu.dll* and wowarmhw.dll do. However, that's not possible when running an x86 guest on arm64. In that case, the CPU emulation layer (*xtajit.dll*) needs to perform *binary translation* to parse and emulate x86 instructions. While many emulation strategies exist, *xtajit.dll* performs **jitting**, that is, just-in-time generation of native code from guest instructions. In addition, *xtajit.dll* communicates with an NT service called **XtaCache** to persist jitted code on disk such that it can prevent re-jitting the same code when the guest binary runs again.

As mentioned earlier, WoW64 guests run with guest versions of OS binaries that live in SysWoW64 for (x86) or SysArm32 (arm32) directories under c:\Windows. From a performance perspective, that's OK if the host CPU can execute guest instructions, but when jitting is necessary, having to jit and cache OS binaries is not ideal. A better approach could have been to *pre-jit* these OS binaries and ship them with the OS. That's still not ideal because jitting arm64 instructions from x86 instructions misses a lot of the context that exists in source code and results in suboptimal code due to the architectural differences between x86 and arm64. For example, the strongly ordered memory model of the x86 vs. the weak memory model or arm64 forces the jitter to pessimistically add expensive memory barrier instructions.

A much better option is to enhance the compiler toolchain to pre-compile the OS binaries from source code to arm64 directly, but in an x86-compatible way. That means the compiler uses x86 types and structures, but generates arm64 instructions along with thunks to perform calling convention adjustments for calls from and to x86 code. For example, x86 function calls generally pass parameters on the stack whereas the arm64 calling convention expects them in registers. Any x86 assembly code is linked into the binary as is. These types of binaries containing both x86-compatible arm64 code as well as x86 code are called **CHPE** (**Compiled Hybrid Portable Executable**) binaries. They are stored under c:\Windows\SyChpe32 and are loaded whenever the x86 application tries to load a DLL from SysWoW64, providing improved performance by almost completely eliminating emulation for OS code. Figure 11-28 shows CHPE DLLs in the address space of the emulated x86 process on an arm64 machine.

x64 Emulation on arm64

The first arm64 release of Windows 10 in 2017 only supported emulating 32-bit x86 programs. While most Windows software has a 32-bit version, an increasing number of popular applications, especially games, are only available as x64. For that reason, Microsoft added support for x64-on-arm64 emulation in Windows 11. It's pretty remarkable that one can run x86, x64, arm32, and arm64 applications on the arm64 version of Windows 11.

There are many similarities between how emulation is implemented for x86 and x64 guest architectures as shown in Fig. 11-29. Instruction emulation still happens via a jitter, *xtajit64.dll*, which has been ported to support x64 machine code. Since a given process cannot have both x86 and x64 code, either *xtajit.dll* or *xtajit64.dll* is loaded, as appropriate. Jitted code is persisted via the XtaCache NT service, as before. User-mode OS binaries intended to load into x64 processes are built using a hybrid binary interface similar to CHPE, called **ARM64EC ARM 64 Emulation Compatible**. ARM64EC binaries contain arm64 machine code, compiled using x64 types and behaviors with thunks to perform calling convention

adjustments. As such, other than x64 assembly code which may be linked into these binaries, there's no need for any instruction emulation and they run at native speed.

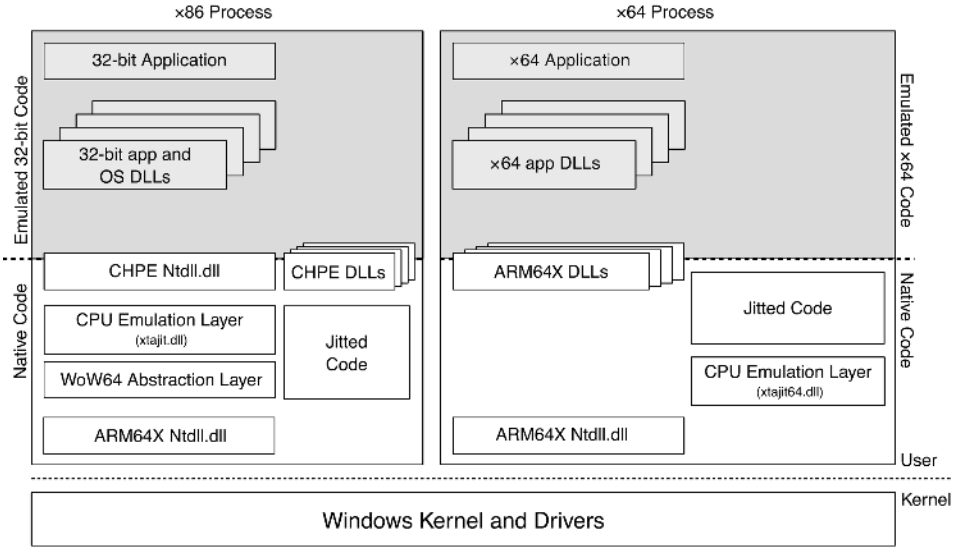

Figure 11-29. Comparison of x86 and x64 emulation infrastructure on an arm64 machine. Shaded areas indicate emulated code.

There are also some big differences between x86 and x64 emulation. First of all, x64 emulation does not rely at all on the WoW64 infrastructure because no 32-bit to 64-bit thunking or redirection of file system or registry paths is necessary; these are already 64-bit applications and use 64-bit types and data structures. In fact, ARM64EC binaries which do not contain any x64 code can run just like native arm64 binaries with no intervention by the emulator; ARM64EC is effectively a second native architecture supported on arm64. The remaining role of the WoW64 abstraction layer has been moved into the ARM64EC *ntdll.dll* which loads in x64 processes. This *ntdll* is enlightened to allow loading x64 binaries and summon the xtajit64 jitter to emulate x64 machine code.

At this point, careful readers might be asking themselves: given that no file system redirection exists for x64 applications on arm64, would not an x64 process end up loading the arm64 native DLL if, for example, it tries to load c:\windows\system32\kernelbase.dll? The answer is yes and no. Yes, the x64 process will load the *kernelbase.dll* under the system32 directory (which normally contains native binaries), but the DLL will be transformed in memory depending on whether it gets loaded into an x64 process or an arm64 process. This is possible because arm64 uses a new type of portable executable (PE) binary called **ARM64X** for user-mode OS binaries. ARM64X binaries contain both native arm64 code as well

as x64 compatible code (ARM64EC or x64 machine code) and the necessary metadata to switch between the two personalities. On disk, these files look like regular native arm64 binaries: the machine type field in the PE header indicates arm64 and export tables point to native arm64 code. However, when this binary is loaded into an x64 process, the kernel memory manager transforms the process' view of the binary by applying modifications described by the metadata similar to how relocation fixups are performed. The PE header machine type field, the export and import table pointers are adjusted to make the binary appear as an ARM64EC binary to the process.

In addition to helping eliminate file system redirection, ARM64X binaries provide another significant benefit. For most functions compiled into the binary, the native arm64 compiler and the ARM64EC compiler will generate identical arm64 machine instructions. Such code can be single-instanced in the ARM64X binary rather than being stored as two copies, thus reducing binary size as well as allowing the same code pages to be shared in memory between arm64 and x64 processes.

11.5 MEMORY MANAGEMENT

Windows has an extremely sophisticated and complex virtual memory system. It has a number of Win32 functions for using it, implemented by the memory manager—the largest component of NTOS. In the following sections, we will look at the fundamental concepts, the Win32 API calls, and finally the implementation.

11.5.1 Fundamental Concepts

Since Windows 11 supports only 64-bit machines, this chapter is only going to consider 64-bit processes on 64-bit machines. 32-bit emulation on 64-bit machines was described in the WoW64 section earlier.

In Windows, every user process has its own virtual address space, split equally between kernel-mode and user-mode. Today's 64-bit processors generally implement 48-bits of virtual addresses resulting in a 256 TB total address space. When the full 64-bit addresses are not implemented, hardware requires that all the unimplemented bits be the same as the highest implemented bit. Addresses in this format are called **canonical**. This approach helps ensure that applications and operating systems do not rely on storing information in these bits to make future expansion possible. Out of the 256 TB address space, user-mode takes the lower 128 TB, kernel-mode takes the upper 128 TB. Even though this may sound, pretty big, a nontrivial portion is actually already reserved for various categories of data, security mitigations as well as for performance optimizations.

On today's 64-bit processors, the 48-bits of virtual addresses are mapped using a 4-level page table scheme where each page table is 4 KB in size and each **PTE**

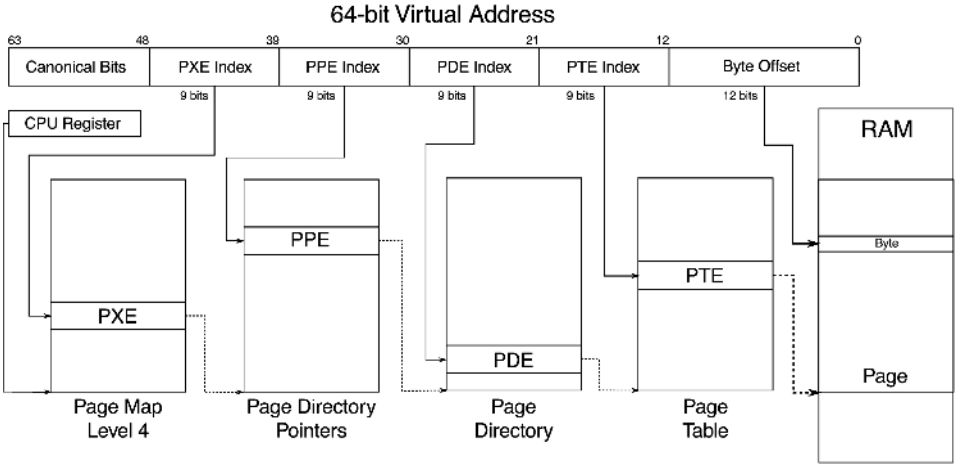

Figure 11-30. Virtual to physical address translation with a 4-level page table scheme implementing 48-bits of virtual address.

(**Page Table Entry**) is 8 bytes with 512 PTEs per page table. As a result, each page table is indexed by 9 bits of the virtual address and the remaining 12 bits of the 48-bit virtual address is the byte index into the 4 KB page. The physical address of the topmost level table is contained in a special processor register, and this register is updated during context switches between processes. This virtual to physical address translation is shown in Fig. 11-30. Windows also takes advantage of the hardware support for larger page sizes (where available), where a page directory entry can map a 2-MB **large page** or a page directory parent entry can map a 1-GB **huge page**.

Emerging hardware implements 57-bits of virtual addresses using a 5-level page table. Windows 11 supports these processors and provides 128 PB of address space on such machines. In our discussion, we will generally stick to the more common 48-bit implementations.

The virtual address space layouts for two 64-bit processes are shown in Fig. 11-31 in simplified form. The bottom and top 64 KB of each process' virtual address space is normally unmapped. This choice was made intentionally to help catch programming errors and mitigate the exploitability of certain types of vulnerabilities.

Starting at 64 KB comes the user's private code and data. This extends up to 128 TB – 64 KB. The upper 128 TB of the address space is called the **kernel address space** and contains the operating system, including the code, data, paged and nonpaged pools, and numerous other OS data structures. The kernel address space is shared among all user processes, except for per-process and per-session data like page tables and session pool. Of course, this part of the address space is

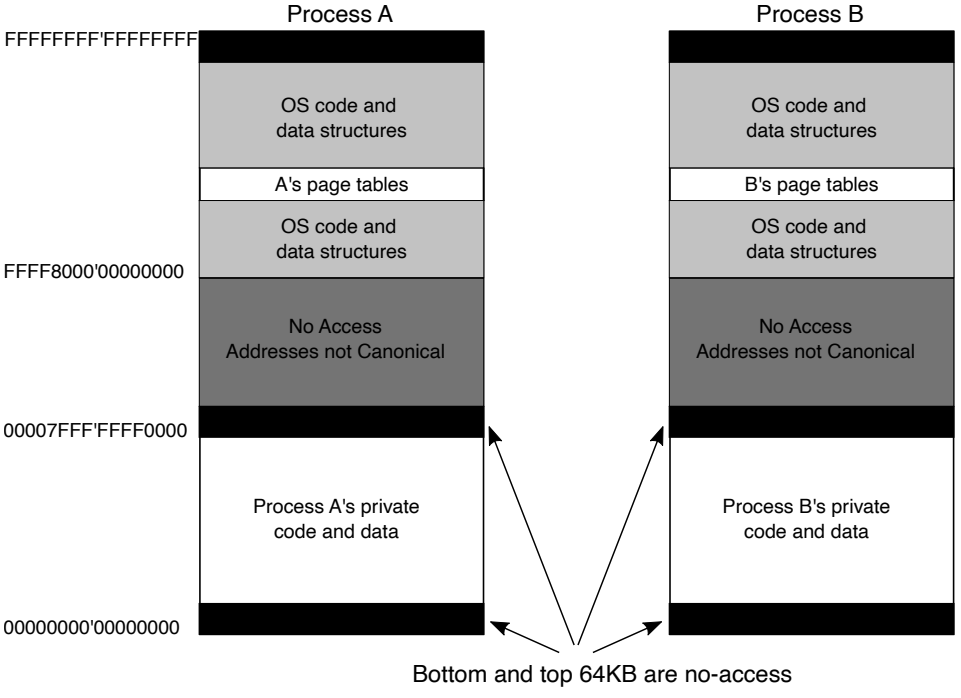

Figure 11-31. Virtual address space layout for three 64-bit user processes. The white areas are private per process. The shaded areas are shared among all processes.

only accessible while running in kernel-mode, so any access attempt from user-mode will result in an access violation. The reason for sharing the process' virtual memory with the kernel is that when a thread makes a system call, it traps into kernel mode and can continue running without changing the memory map by updating the special processor register. All that has to be done is switch to the thread's kernel stack. From a performance point of view, this is a big win, and something UNIX does as well. Because the process' user-mode pages are still accessible, the kernel-mode code can read parameters and access buffers without having to switch back and forth between address spaces or temporarily double-mapping pages into both. The trade-off being made here is less private address space per process in return for faster system calls.

Windows allows threads to attach themselves to other address spaces while running in the kernel. Attachment to an address space allows the thread to access all of the user-mode address space, as well as the portions of the kernel address space that are specific to a process, such as the self-map for the page tables. However, threads must switch back to their original address space before returning to user mode, of course.

Virtual Address Allocation

Each page of virtual addresses can be in one of three states: invalid, reserved, or committed. An **invalid page** is not currently mapped to a memory section object and a reference to it causes a page fault that results in an access violation. Once code or data is mapped onto a virtual page, the page is said to be **committed**. A committed page does not necessarily have a physical page allocated for it, but the operating system has ensured that a physical page *is guaranteed to be available* when necessary. A page fault on a committed page results in mapping the page containing the virtual address that caused the fault onto one of the pages represented by the section object or stored in the pagefile. Often this will require allocating a physical page and performing I/O on the file represented by the section object to read in the data from disk. But page faults can also occur simply because the page table entry needs to be updated, as the physical page referenced is still cached in memory, in which case I/O is not required. These are called **soft faults**. We will discuss them in more detail shortly.

A virtual page can also be in the **reserved** state. A reserved virtual page is invalid but has the property that those virtual addresses will never be allocated by the memory manager for another purpose. As an example, when a new thread is created, many pages of user-mode stack space are reserved in the process' virtual address space, but only one page is committed. As the stack grows, the virtual memory manager will automatically commit additional pages under the covers, until the reservation is almost exhausted. The reserved pages function as guard pages to keep the stack from growing too far and overwriting other process data. Reserving all the virtual pages means that the stack can eventually grow to its maximum size without the risk that some of the contiguous pages of virtual address space needed for the stack might be given away for another purpose. In addition to the invalid, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable.

Pagefiles

An interesting trade-off occurs with assignment of backing store to committed pages that are not being mapped to specific files. These pages use the **pagefile**. The question is *how* and *when* to map the virtual page to a specific location in the pagefile. A simple strategy would be to assign each virtual page to a page in one of the paging files on disk at the time the virtual page was committed. This would guarantee that there was always a known place to write out each committed page should it be necessary to evict it from memory, but it would require a much larger pagefile than necessary and would not be able to support small pagefiles.

Windows uses a *just-in-time* strategy. Committed pages that are backed by the pagefile are not assigned space in the pagefile until the time that they have to be paged out. The memory manager maintains a system-wide **commit limit** which is

the sum of RAM size and the total size of all pagefiles. As non-paged or pagefile-backed virtual memory is committed, system-wide **commit charge** is increased until it reaches the commit limit at which point commit requests start failing. This strict commit tracking ensures that pagefile space will be available when a committed page needs to be paged out. No disk space is allocated for pages that are never paged out. If the total virtual memory is less than the available physical memory, a pagefile is not needed at all. This is convenient for embedded systems based on Windows. It is also the way the system is booted, since pagefiles are not initialized until the first user-mode process, *smss.exe*, begins running.

With demand-paging, requests to read pages from disk need to be initiated right away, as the thread that encountered the missing page cannot continue until this *page-in* operation completes. The possible optimizations for faulting pages into memory involve attempting to prepage additional pages in the same I/O operation, called **page fault clustering**. However, operations that write modified pages to disk are not normally synchronous with the execution of threads. The just-in-time strategy for allocating pagefile space takes advantage of this to boost the performance of writing modified pages to the pagefile. Modified pages are grouped together and written in big chunks. Since the allocation of space in the pagefile does not happen until the pages are being written, the number of seeks required to write a batch of pages can be optimized by allocating the pagefile pages to be near each other, or even making them contiguous.

While grouping modified pages into bigger chunks before writing to pagefile is beneficial for disk write efficiency, it does not necessarily help make in-page operations any faser. In fact, if pages from different processes or discontiguous virtual addresses are combined together, it becomes impossible to cluster pagefile reads during in-page operations since subsequent virtual addresses belonging to the faulting process may be scattered across the pagefile. In order to optimize pagefile read efficiency for groups of virtual pages that are expected to be used together, Windows supports the concept of **pagefile reservations**. Ranges of pagefile can be *soft-reserved* for process virtual memory pages such that when those pages are about to be written to the pagefile, they are written to their reserved locations instead. While this can make pagefile writing less efficient compared to not having such reservations, subsequent page-in operations proceed much quicker because of improved clustering and sequential disk reads. Since in-page operations directly block application progress, they are generally more important for system performance than pagefile write efficiency. These are soft reservations, so if the pagefile is full and no other space is available, the memory manager will overwrite unoccupied reserved space.

When pages stored in the pagefile are read into memory, they keep their allocation in the pagefile until the first time they are modified. If a page is never modified, it will go onto a list of cached physical pages, called the **standby list**, where it can be reused without having to be written back to disk. If it *is* modified, the memory manager will free the pagefile space and the only copy of the page will be in

memory. The memory manager implements this by marking the page as read-only after it is loaded. The first time a thread attempts to write the page the memory manager will detect this situation and free the pagefile page, grant write access to the page, and then have the thread try again.

Windows supports up to 16 pagefiles, normally spread out over separate disks to achieve higher I/O bandwidth. Each one has an initial size and a maximum size it can grow to later if needed, but it is better to create these files to be the maximum size at system installation time. If it becomes necessary to grow a pagefile when the file system is much fuller, it is likely that the new space in the pagefile will be highly fragmented, reducing performance.

The operating system keeps track of which virtual page maps onto which part of which paging file by writing this information into the page table entries for the process for private pages, or into prototype page table entries associated with the section object for shared pages. In addition to the pages that are backed by the pagefile, many pages in a process are mapped to regular files in the file system.

The executable code and read-only data in a program file (e.g., an EXE or DLL) can be mapped into the address space of whatever process is using it. Since these pages cannot be modified, they never need to be paged out and end up on the standby list as cached pages when they are no longer in use and can immediately be reused. When the page is needed again in the future, the memory manager will read the page in from the program file.

Sometimes pages that start out as read-only end up being modified, for example, setting a breakpoint in the code when debugging a process, or fixing up code to relocate it to different addresses within a process, or making modifications to data pages that started out shared. In cases like these, Windows, like most modern operating systems, supports a type of page called **copy-on-write**. These pages start out as ordinary mapped pages, but when an attempt is made to modify any part of the page the memory manager makes a private, writable copy. It then updates the page table for the virtual page so that it points at the private copy and has the thread retry the write—which will succeed the second time. If that copy later needs to be paged out, it will be written to the pagefile rather than the original file,

Besides mapping program code and data from EXE and DLL files, ordinary files can be mapped into memory, allowing programs to reference data from files without doing read and write operations. I/O operations are still needed, but they are provided implicitly by the memory manager using the section object to represent the mapping between pages in memory and the blocks in the files on disk.

Section objects do not have to refer to a file. They can refer to anonymous regions of memory, called **pagefile-backed sections**. By mapping pagefile-backed section objects into multiple processes, memory can be shared without having to allocate a file on disk. Since sections can be given names in the NT namespace, processes can rendezvous by opening sections by name, as well as by duplicating and passing handles between processes.

11.5.2 Memory-Management System Calls

The Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. The most important of these functions are listed in Fig. 11-32. All of them operate on a region consisting of either a single page or a sequence of two or more pages that are consecutive in the virtual address space. Of course, processes do not have to manage their memory; paging happens automatically, but these calls give processes additional power and flexibility. Most applications use higher-level heap APIs to allocate and free dynamic memory. Heap implementations build on top of these lower-level memory management calls to manage smaller blocks of memory.

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file-mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file-mapping object

Figure 11-32. The principal Win32 API functions for managing virtual memory in Windows.

The first four API functions are used to allocate, free, protect, and query regions of virtual address space. Allocated regions always begin on 64-KB boundaries to minimize porting problems to future architectures with pages larger than current ones as well as reducing virtual address space fragmentation. The actual amount of address space allocated can be less than 64 KB, but must be a multiple of the page size. The next two APIs give a process the ability to hardwire pages in memory so they will not be paged out and to undo this property. A real-time program might need pages with this property to avoid page faults to disk during critical operations, for example. A limit is enforced by the operating system to prevent processes from getting too greedy. The pages actually can be removed from memory, but only if the entire process is swapped out. When it is brought back, all the locked pages are reloaded before any thread can start running again. Although not shown in Fig. 11-32, Windows also has native API functions to allow a process to read/write the virtual memory of a different process over which it has been given control, that is, for which it has a handle (see Fig. 11-7).

The last four API functions listed are for managing sections (i.e., file-backed or pagefile-backed sections). To map a file, a file-mapping object must first be created with CreateFileMapping (see Fig. 11-8). This function returns a handle to the file-mapping object (i.e., a section object) and optionally enters a name for it into the Win32 namespace so that other processes can use it, too. The next two functions map and unmap views on section objects from a process' virtual address space. The last API can be used by a process to map share a mapping that another process created with CreateFileMapping, usually one created to map anonymous memory. In this way, two or more processes can share regions of their address spaces. This technique allows them to write in limited regions of each other's virtual memory.

11.5.3 Implementation of Memory Management

Windows supports a single linear 256 TB demand-paged address space per process. Segmentation is not supported in any form. As noted earlier, page size is 4 KB on all processor architectures supported by Windows today. In addition, the memory manager can use 2-MB large pages or even 1-GB huge pages to improve the effectiveness of the **TLB** (**Translation Lookaside Buffer**) in the processor's memory management unit. Use of large and huge pages by the kernel and large applications significantly improves performance by improving the hit rate for the TLB as well as enabling a shallower and thus faster hardware page table walk when a TLB miss does occur. Large and huge pages are simply composed of aligned, contiguous runs of 4 KB pages. As such, these pages are considered non-pageable since paging and reusing them for single pages would make it very difficult, if not impossible, for the memory manager to construct a large or huge page when the application accesses it again.

Unlike the scheduler, which selects individual threads to run and does not care much about processes, the memory manager deals entirely with processes and does not care much about threads. After all, processes, not threads, own the address space and that is what the memory manager is concerned with. When a region of virtual address space is allocated, as four of them have been for process *A* in Fig. 11-33, the memory manager creates a **VAD** (**Virtual Address Descriptor**) for it, listing the range of addresses mapped, the section representing the backing store file and offset where it is mapped, and the permissions. When the first page is touched, the necessary page table hierarchy is created and corresponding page table entries are filled in as physical pages are allocated to back the VAD. An address space is completely defined by the list of its VADs. The VADs are organized into a balanced tree, so that the descriptor for a particular address can be found efficiently. This scheme supports sparse address spaces. Unused areas between the mapped regions use no memory or disk so they are essentially free.

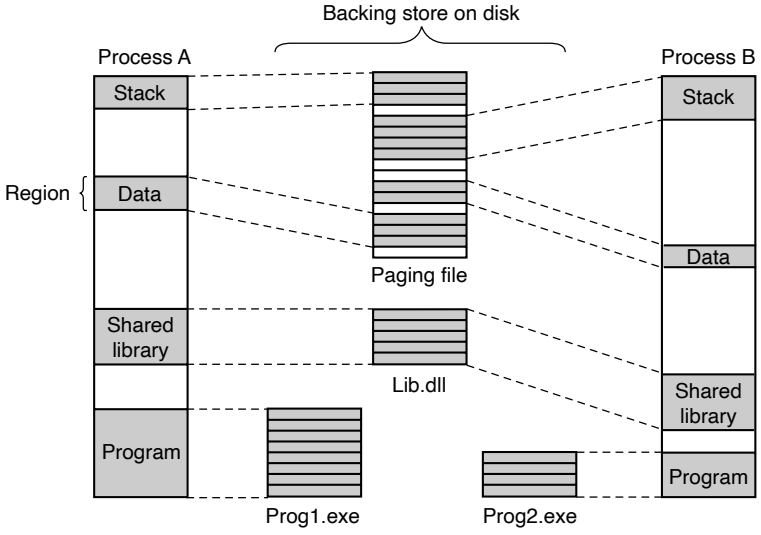

Figure 11-33. Mapped regions with their shadow pages on disk. The *lib.dll* file is mapped into two address spaces at the same time.

Page-Fault Handling

Windows is a demand-paged operating system, meaning that physical pages are generally not allocated and mapped into a process address space until they are actually accessed by some process (although there is also prepaging the background for performance reasons), Demand paging in the memory manager is driven by page faults. On each page fault, a trap to the kernel occurs and the CPU enters kernel mode. The kernel then builds a machine-independent descriptor telling what happened and passes this to the memory-manager part of the executive. The memory manager then checks the access for validity. If the faulted page falls within a committed region and access is allowed, it looks up the address in the VAD tree and finds (or creates) the process page table entry.

Generally, pageable memory falls into one of two categories: *private pages* and *shareable pages*. Private pages only have meaning within the owning process; they are not shareable with other processes. As such, these pages become free pages when the process terminates. For example, VirtualAlloc calls allocate private memory for the process. Shareable pages represent memory that can be shared with other processes. Mapped files and pagefile-backed sections fall into this category. Since these pages have relevance outside of the process, they get cached in memory (on the standby or modified lists) even after the process terminates because some other process might need them. Each page fault can be considered as being in one of five categories:

1. The page referenced is not committed.
2. Access to a page has been attempted in violation of the permissions.
3. A shared copy-on-write page was about to be modified.
4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

The first and second cases are due to programming errors. If a program attempts to use an address which is not supposed to have a valid mapping, or attempts an invalid operation (like attempting to write a read-only page), this is called an **access violation** and causes the memory manager to raise an exception, which, if not handled, results in termination of the process. Access violations are often the result of bad pointers, including accessing memory that was freed and unmapped from the process.

The third case has the same symptoms as the second one (an attempt to write to a read-only page), but the treatment is different. Because the page has been marked as *copy-on-write*, the memory manager does not report an access violation, but instead makes a private copy of the page for the current process and then returns control to the thread that attempted to write the page. The thread will retry the write, which will now complete without causing a fault.

The fourth case occurs when a thread pushes a value onto its stack and crosses onto a page which has not been allocated yet. The memory manager is programmed to recognize this as a special case. As long as there is still room in the virtual pages reserved for the stack, the memory manager will supply a new physical page, zero it, and map it into the process. When the thread resumes running, it will retry the access and succeed this time around.

Finally, the fifth case is a normal page fault. However, it has several subcases. If the page is mapped by a file, the memory manager must search its data structures, such as the prototype page table associated with the section object to be sure that there is not already a copy in memory. If there is, say in another process or on the standby or modified page lists, it will just share it—perhaps marking it as copy-on-write if changes are not supposed to be shared. If there is not already a copy, the memory manager will allocate a free physical page and arrange for the file page to be copied in from disk, unless another page is already transitioning in from disk, in which case it is only necessary to wait for the transition to complete.

When the memory manager can satisfy a page fault by finding the needed page in memory rather than reading it in from disk, the fault is classified as a **soft fault**. If the copy from disk is needed, it is a **hard fault**. Soft faults are much cheaper and have little impact on application performance compared to hard faults. Soft faults can occur because a shared page has already been mapped into another process, or the needed page was trimmed from the process' working set but is being requested again before it has had a chance to be reused. A common sub-category

of soft faults is *demand-zero* faults. These indicate that a zeroed page should be allocated and mapped in, for example, when the first access to a VirtualAlloc'd address occurs. When trimming private pages from process working sets, Windows checks if the page is entirely zero. If so, instead of putting the page on the modified list and writing it out to the pagefile, the memory manager frees the page and encodes the PTE to indicate a demand-zero fault on next access. Soft faults can also occur because pages have been compressed to effectively increase the size of physical memory. For most configurations of CPU, memory, and I/O in current systems, it is more efficient to use compression rather than incur the I/O expense (performance and energy) required to read a page from disk. We will cover memory compression in more detail later in this section.

When a physical page is no longer mapped by the page table in any process, it goes onto one of three lists: free, modified, or standby. Pages that will never be needed again, such as stack pages of a terminating process, are freed immediately. Pages that may be faulted again go to either the modified list or the standby list, depending on whether or not the dirty bit was set for any of the page table entries that mapped the page since it was last read from disk. Pages in the modified list will be eventually written to disk, then moved to the standby list.

Since soft faults are much quicker to satisfy than hard faults, a big performance improvement opportunity is to **prepage** or **prefetch** into the standby list, data that is expected to be used soon. Windows makes heavy use of prefetch in several ways:

1. **Page fault clustering**: When satisfying hard page faults from files or from the pagefile, the memory manager reads additional pages, up to a total of 64 KB, as long as the next page in the file corresponds to the next virtual page. That is almost always the case for regular files so mechanisms like pagefile reservations we described earlier in the section help clustering efficiency for pagefiles.

2. **Application-launch prefetching**: Application launches are generally very consistent from launch to launch: the same application and DLL pages are accessed. Windows takes advantage of this behavior by tracing the set of file pages accessed during an application launch, persisting this history on disk, identifying those pages that are indeed consistently accessed and prefetching them during the next launch potentially seconds before the application actually needs them. When the pages to prefetch are already resident in memory, no disk I/Os are issued, but when they are not, application-launch prefetch routinely issues hundreds of I/O requests to disk which significantly improves disk read efficiency on both rotational and solid state disks.

3. **Working set in-swap**: The working set of a process in Windows is composed of the set of user-mode virtual addresses that are mapped by valid PTEs, that is, addresses that can be accessed without a page

fault. Normally, when the memory manager detects memory pressure, it trims pages from process working sets in order to generate more available memory. The UWP application model provides an opportunity for a more optimized approach due to its lifetime management. UWP applications are suspended via their job objects when they are no longer visible and resumed when the user switches back to them. This reduces CPU consumption and power usage.

4. **Working set out-swap**. Working set out-swap involves reserving preferably sequential space in the pagefile for each page in the process working set and remembering the set of pages that are in the working set. In order to improve the chances of finding sequential space, Windows actually creates and uses a separate pagefile called *swapfile.sys* exclusively for working set out-swap. When under memory pressure, the entire working set of the UWP application is emptied at once and since each page is reserved sequential space in the swapfile, pages removed from the working set can be written out very efficiently with large, sequential I/Os. When the UWP application is about to be resumed, the memory manager performs a working set in-swap operation where it prefetches the out-swapped pages from the swapfile, using large, sequential reads, directly into the working set. In addition to maximizing disk read bandwidth, this also avoids any subsequent soft-faults because the working set is fully restored to its state before suspension.

5. **Superfetch**: Today's desktop systems generally have 8, 16, 32, 64, or even more GBs of memory installed, and this memory is largely empty after a system boot. Similarly, memory contents can experience significant disruptions, for example, when the user runs a big game, which pushes out everything else to disk, and then exits the game. Having GBs of empty memory is lost opportunity because the next application launch or switching to an old browser tab will need to page in lots of data from disk. Would it not be better if there was a mechanism to populate empty memory pages with useful data in the background, and cache them on the standby list? That's what Superfetch does. It's a user-mode service for proactive memory management. It tracks frequently-used file pages and prefetches them into the standby list when free memory is available. Superfetch also tracks paged out private pages of important applications and brings these into memory as well. As opposed to the earlier forms of prefetch, which are just-in-time, Superfetch employs background prefetch, using low-priority I/O requests in order to avoid creating disk contention with higher-priority disk reads.

Page Tables

The format of the page table entries differs depending on the processor architecture. For the x64 architecture, the entries for a mapped page are shown in Fig. 11-34. If an entry is marked valid, its contents are interpreted by the hardware so that the virtual address can be translated into the correct physical page. Unmapped pages also have entries, but they are marked *invalid* and the hardware ignores the rest of the entry. The software format is somewhat different from the hardware format and is determined by the memory manager. For example, for an unmapped page that must be allocated and zeroed before it may be used, that fact is noted in the page table entry.

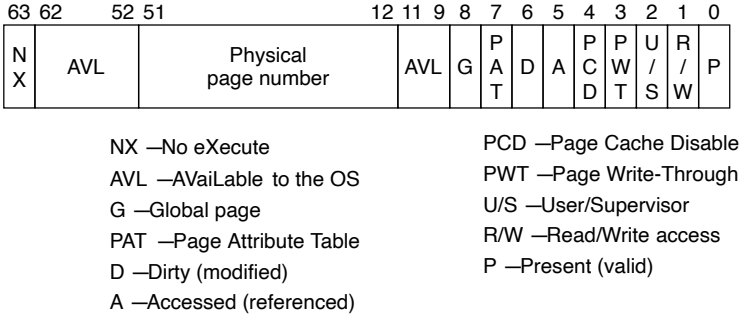

Figure 11-34. A page table entry (PTE) for a mapped page on the Intel x86 and AMD x64 architectures.

Two important bits in the page table entry are updated by the hardware directly. These are the access (A) and dirty (D) bits. These bits keep track of when a particular page mapping has been used to access the page and whether that access could have modified the page by writing it. This really helps the performance of the system because the memory manager can use the access bit to implement the **LRU** (**Least-Recently Used**) style of paging. The LRU principle says that pages that have not been used the longest are the least likely to be used again soon. The access bit allows the memory manager to determine that a page has been accessed. The dirty bit lets the memory manager know that a page may have been modified, or more significantly, that a page has *not* been modified. If a page has not been modified since being read from disk, the memory manager does not have to write the contents of the page to disk before using it for something else.

The page table entries in Fig. 11-34 refer to physical page numbers, not virtual page numbers. To update entries in the page table hierarchy, the kernel needs to use virtual addresses. Windows maps the page table hierarchy for the current process into kernel virtual address space using a clever self-map technique, as shown in Fig. 11-35. By making an entry (the self-map PXE entry) in the top-level page table point to the top-level page table, the Windows memory manager creates virtual

addresses that can be used to refer to the entire page table hierarchy. Figure 11-35 shows two example virtual address translations (a) for the self-map entry and (b) for a page table entry. The self-map occupies the same 512 GB of kernel virtual address space for every process because a top-level PXE entry maps 512 GB.

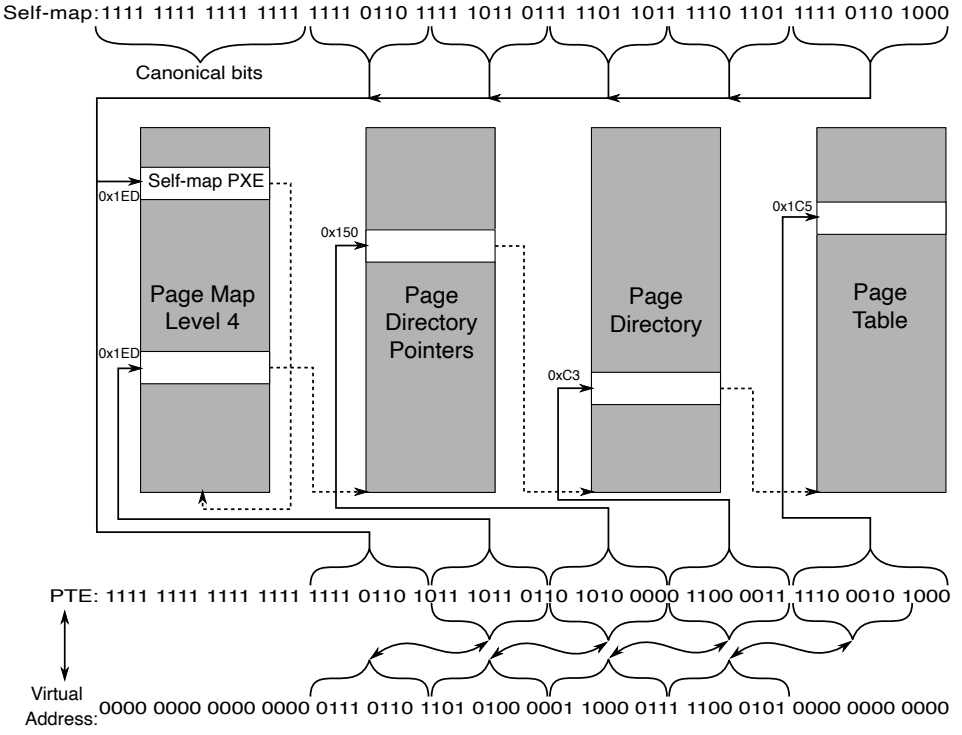

Figure 11-35. The Windows self-map entries are used to map the physical pages of the page table hierarchy into kernel virtual addresses. This makes conversion between a virtual address and its PTE address very easy.

The Page Replacement Algorithm

When the number of free physical memory pages starts to get low, the memory manager starts working to make more physical pages available by removing them from user-mode processes as well as the system process, which represents kernel-mode use of pages. The goal is to have the most important virtual pages present in memory and the others on disk. The trick is in determining what *important* means. In Windows this is answered by making heavy use of the working-set concept. Each process (*not* each thread) has a working set. This set consists of the mapped-in pages that are in memory and thus can be referenced without a page fault. The size and composition of the working set fluctuates in unpredictable ways as the process' threads run, of course.

Working sets come into play only when the available physical memory is getting low in the system. Otherwise processes are allowed to consume memory as they choose, often far exceeding the working-set maximum. But when the system comes under **memory pressure**, the memory manager starts to squeeze processes back into their working sets, starting with processes that are over their maximum by the most. There are three levels of activity by the working-set manager, all of which is periodic based on a timer. New activity is added at each level:

1. **Lots of memory available:** Scan pages resetting access bits and using their values to represent the *age* of each page. Keep an estimate of the unused pages in each working set.
2. **Memory getting tight:** For any process with a significant proportion of unused pages, stop adding pages to the working set and start replacing the oldest pages whenever a new page is needed. The replaced pages go to the standby or modified list.
3. **Memory is tight:** Trim (i.e., reduce) working sets by removing the oldest pages.

The working set manager runs every second, called from the **balance set manager** thread. The working-set manager throttles the amount of work it does to keep from overloading the system. It also monitors the writing of pages on the modified list to disk to be sure that the list does not grow too large, waking the ModifiedPageWriter thread as needed.

Physical Memory Management

Above we mentioned three different lists of physical pages, the free list, the standby list, and the modified list. There is a fourth list which contains free pages that have been zeroed. The system frequently needs pages that contain all zeros. When new pages are given to processes, or the final partial page at the end of a file is read, a zero page is needed. It is time consuming to fill a page with zeros on demand, so it is better to create zero pages in the background using a low-priority thread. There is also a fifth list used to hold pages that have been detected as having hardware errors (i.e., through hardware error detection).

All pages in the system are managed using a data structure called the **PFN database** (**Page Frame Number database**), as shown in Fig. 11-36. The PFN database is a table indexed by physical page frame number where each entry represents the state of the corresponding physical page, using different formats for different page types (e.g., sharable vs. private). For pages that are in use, the PFN entry contains information about how many references exist to the page and how many page table entries reference it such that the system can track when the page is no longer in use. There is also a pointer to the PTE which references the physical page. For private pages, this is the address of the hardware PTE but for shareable

pages, it is the address of the prototype PTE. To be able to edit the PTE when in a different process address space, the PFN entry also contains the page frame index of the page that contains the PTE.

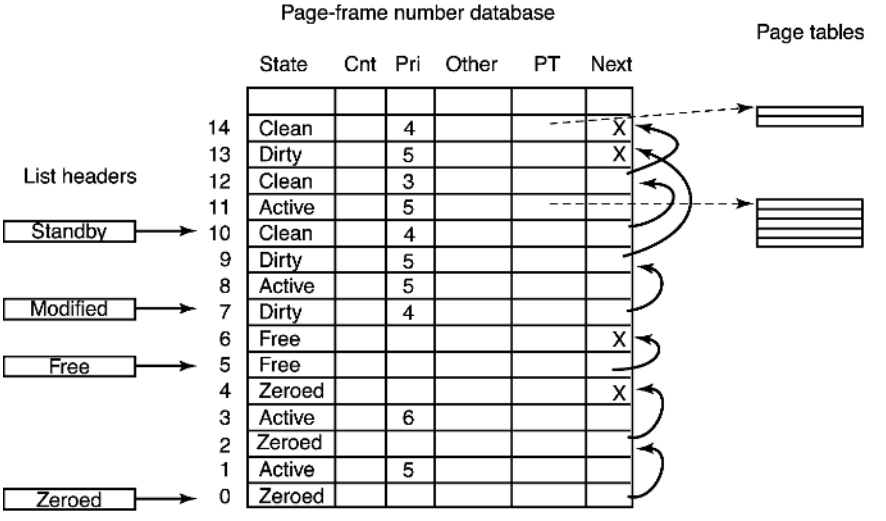

Figure 11-36. Some of the fields in the page-frame database for a valid page.

Additionally the PFN entry contains forward and backward links for the aforementioned page lists and various flags, such as *read in progress*, *write in progress*, and so on. To save space, the lists are linked together with fields referring to the next element by its index within the table rather than pointers. The table entries for the physical pages are also used to summarize the dirty bits found in the various page table entries that point to the physical page (i.e., because of shared pages). There is also information used to represent differences in memory pages on larger server systems which have memory that is faster from some processors than from others, namely NUMA machines.

One important PFN entry field is priority. The memory manager maintains *page priority* for every physical page. Page priorities range from 0 to 7 and reflect how "important" a page is or how likely it is to get re-accessed. The memory manager ensures that higher-priority pages are more likely to remain in memory rather than getting paged out and reused. Working set trimming policy takes page priority into account by trimming lower-priority pages before higher-priority ones even if they are more recently accessed. Even though we generally talk about the standby list as if it is a single list, it is actually composed of eight lists, one for each priority. When a page is inserted into the standby list, it is linked to the appropriate sublist based on its priority. When the memory manager is repurposing pages off the standby list, it does so starting with the lowest-priority sublists. That way, higher-priority pages are more likely to avoid getting repurposed.

Pages are moved between the working sets and the various lists based on actions taken by the processes themselves as well as the working-set manager and other system threads. Let us examine the transitions. When the working-set manager removes pages from a working set, or when a process unmaps a file from its address space, the removed pages goes on the bottom of the standby or modified list, depending on its cleanliness. This transition is shown as (1) in Fig. 11-37.

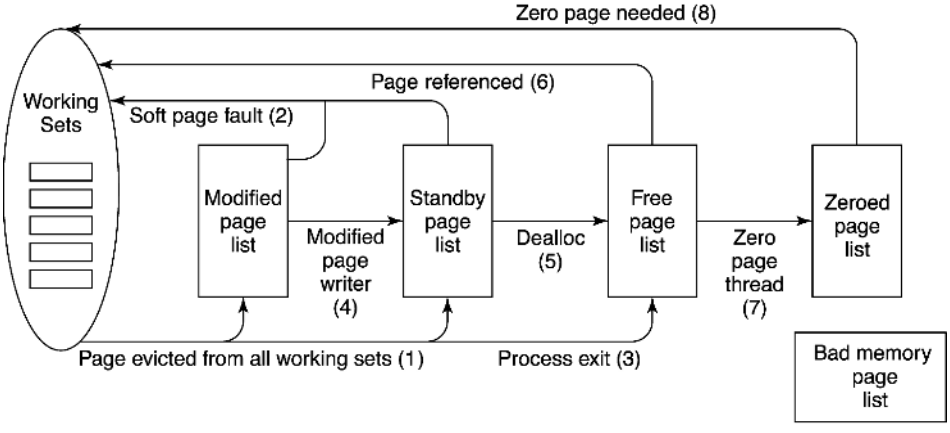

Figure 11-37. The various page lists and the transitions between them.

Pages on both lists are still live pages, so if a page fault occurs and one of these pages is needed, it is removed from the list and faulted back into the working set without any disk I/O (2). When a process exits, its private pages are not live anymore, so they move to the free list regardless of whether they were in the working set or on the modified or standby lists (3). Any pagefile space in use by the process is also freed.

Other transitions are caused by other system threads. Every 4 seconds the balance set manager thread runs and looks for processes all of whose threads have been idle for a certain number of seconds. If it finds any such processes, their kernel stacks are unpinned from physical memory and their pages are moved to the standby or modified lists, also shown as (1).

Two other system threads, the **mapped page writer** and the **modified page writer**, wake up periodically to see if there are enough clean pages. If not, they take pages from the top of the modified list, write them back to disk, and then move them to the standby list (4). The former handles writes to mapped files and the latter handles writes to the pagefiles. The result of these writes is to transform modified (dirty) pages into standby (clean) pages.

The reason for having two threads is that a mapped file might have to grow as a result of the write, and growing it requires access to on-disk data structures to allocate a free disk block. If there is no room in memory to bring them in when a

page has to be written, a deadlock could result. The other thread can solve the problem by writing out pages to a paging file.

The other transitions in Fig. 11-37 are as follows. If a process takes an action to end the lifetime of a group of pages, for example, by decommitting private pages or closing the last handle on a pagefile-backed section or deleting a file, the associated pages become free (5). When a page fault requires a page frame to hold the page about to be read in, the page frame is taken from the free list (6), if possible. It does not matter that the page may still contain confidential information because it is about to be overwritten in its entirety.

The situation is different during demand-zero faults, for example, when a stack grows or when a process takes a page fault on a newly committed private page. In that case, an empty page frame is needed and the security rules require the page to contain all zeros. For this reason, another kernel system thread, the **ZeroPage thread**, runs at the lowest priority (see Fig. 11-26), erasing pages that are on the free list and putting them on the zeroed page list (7). Whenever the CPU is idle and there are free pages, they might as well be zeroed since a zeroed page is potentially more useful than a free page and it costs nothing to zero the page when the CPU is idle. On big servers with terabytes of memory distributed across multiple processor sockets, it can take a long time to zero all that memory. Even though zeroing memory might be thought of as a background activity, when a cloud provider needs to start a new VM and give it terabytes of memory, zeroing pages can easily be the bottleneck. For this reason, the ZeroPage thread is actually composed of multiple threads assigned to each processor and carefully managed to maximize throughput.

The existence of all these lists leads to some subtle policy choices. For example, suppose that a page has to be brought in from disk and the free list is empty. The system is now forced to choose between taking a clean page from the standby list (which might otherwise have been faulted back in later) or an empty page from the zeroed page list (throwing away the work done in zeroing it). Which is better?

The memory manager has to decide how aggressively the system threads should move pages from the modified list to the standby list. Having clean pages around is better than having dirty pages around (since clean ones can be reused instantly), but an aggressive cleaning policy means more disk I/O and there is some chance that a newly cleaned page may be faulted back into a working set and dirtied again anyway. In general, Windows resolves these kinds of trade-offs through algorithms, heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter settings.

Page Combining

One of the interesting optimizations the memory manager performs to optimize system memory usage is called **page combining**. UNIX systems do this, too, but they call it "deduplication," as discussed in Chap. 3. Page combining is the act of single-instancing identical pages in memory and freeing the redundant ones.

Periodically, the memory manager scans process private pages and identifies identical ones by computing hashes to pick candidates and then performing a byte-by-byte comparison after blocking any modification to candidate pages. Once identical pages are found, these private pages are converted to shareable pages transparently to the process. Each PTE is marked copy-on-write such that if any of the sharing processes writes to a combined page, they get their own copy.

In practice, page combining results in fairly significant memory savings because many processes load the same system DLLs at the same addresses which result in many identical pages due to copy-on-written import address table pages, writable data sections and even heap allocations with identical contents. Interestingly, the most common combined page is entirely composed of zeroes, indicating that a lot of code allocates and zeroes memory, but does not write to it afterwards.

While page combining sounds like a broadly applicable optimization, it has various security implications that must be considered. Even though page combining happens without application involvement and is hidden from applications—for example, when they call Win32 APIs to query whether a certain virtual address range is private or shareable—it is possible for an attacker to determine whether a virtual page is combined with others by timing how long it takes to write to the page (and other clever tricks). This can allow the attacker to infer contents of pages in other, potentially more privileged, processes leading to information disclosure. For this reason, Windows does not combine pages across different security domains, except for "well-known" page contents like all-zeroes.

11.5.4 Memory Compression

Another significant performance optimization in Windows memory management is memory compression. It's a feature enabled by default on client systems, but off by default on server systems. Memory compression aims to fit more data into physical memory by compressing currently unused pages such that they take up less space. As a result, it reduces hard page faults and replaces them with soft faults involving a decompression step. Finally, it reduces the volume of pagefile writes as well since all data written to the pagefile is now compressed. Memory compression is implemented in an executive component called the **store manager** which closely integrates with the memory manager and exposes to it a simple key-value interface to add, retrieve, and remove pages.

Let us follow the journey of a private page in a process working set as it goes through the compression pipeline, illustrated in Fig. 11-38. When the memory manager decides to trim the page from the working set based on its normal policies, the private page ends up on the modified list. At some point, the memory manager decides, again based on usual policies, to gather pages from the modified list to write to the pagefile.

Since our page is not compressed, the memory manager calls the store manager's SmPageWrite routine to add the page to a store. The store manager chooses

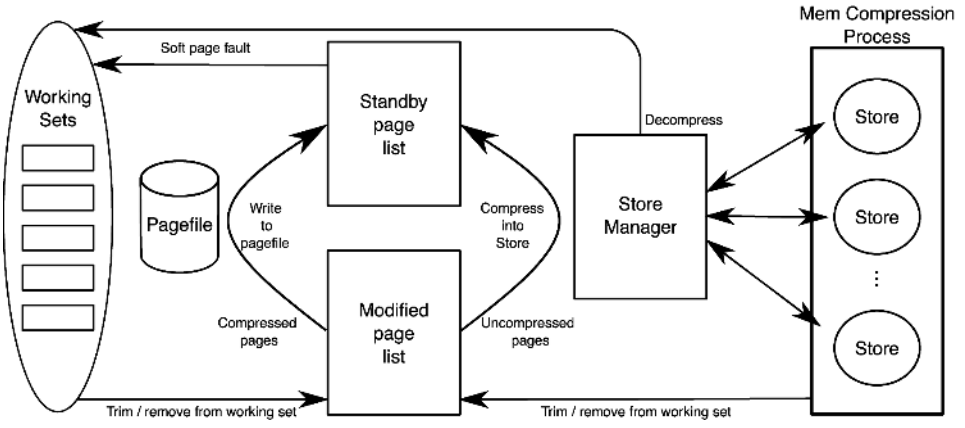

Figure 11-38. Page transitions with memory compression (free/zero lists and mapped files omitted for clarity).

an appropriate store (more on this later), compresses the page into it and returns to the memory manager. Since the page contents have been safely compressed into a store, the memory manager sets its page priority to the lowest (zero) and inserts into the standby list. It could have freed the page, but caching it at low priority is generally a better option because it avoids decompression in case the page may get soft-faulted from the standby list. Let us assume the page has been repurposed off the priority 0 standby sublist and now the process decides to write to the page. That access will result in a page fault and the memory manager will determine that the page is saved to the store manager (rather than the pagefile), so it will allocate a new physical page and call SmPageRead to retrieve page contents into the new physical page. The store manager will route the request to the appropriate store which will find and decompress the data into the target page.

Astute readers may notice that the store manager behaves almost exactly like a regular pagefile, albeit a compressed one. In fact, the memory manager treats the store manager just like another pagefile. During system initialization, if memory compression is enabled, the memory manager creates a **virtual pagefile** to represent the store manager. The size of the virtual pagefile is largely arbitrary, but it limits how many pages can be saved in the store manager at one time, so an appropriate size based on the system commit limit is picked. For most intents and purposes, the virtual pagefile is a real pagefile: it uses one of the 16 pagefile slots and has the same underlying bitmap data structures to manage available space. However, it does not have a backing file and, instead, uses the store manager SmPageRead and SmPageWrite interface to perform I/O. So, during modified page writing, a virtual pagefile offset is allocated for the uncompressed page and the pagefile offset combined with the pagefile number is used as the *key* to identify the page when handing it over to the store manager. After the page is compressed, the PFN entry

and the PTE associated with the page is updated with pagefile index and offset exactly as it is done for a regular pagefile write. When pages in the virtual pagefile are modified or freed and corresponding pagefile space marked free, a system thread called the **store eviction thread** is notified to evict the corresponding keys from store manager via SmPageEvict. One difference between regular pagefiles and the store manager virtual pagefile is that whereas clean pages faulted into working sets are not removed from regular pagefiles, they are evicted from the store manager to avoid keeping both the uncompressed and the compressed copy of the page in memory.

As indicated in Fig. 11-38, the store manager can manage multiple stores. A **system store** is created at boot time as the default store for modified pages. Additional, per-process stores can also be created for individual processes. In practice, this is done for UWP applications. The store manager picks the appropriate store for an incoming modified page based on the owning process.

When the store manager initializes at boot time, it creates the MemCompression system process which provides the user-mode address space for all stores to allocate their backing memory into which incoming pages are compressed. This backing store is regular private pageable memory, allocated with a variant of VirtualAlloc. As such, the memory manager may choose to trim these pages from the MemCompression process working set or a store may decide to explicitly remove them. Once removed, these pages go to the modified list as usual, but since they are coming from the MemCompression process, and thus, are already compressed, the memory manager writes them directly to the pagefile. That's why, when memory compression is enabled, all writes to the pagefile contain compressed data from the MemCompression process.

We mentioned above how UWP applications get their own stores rather than using the system store. This is done to optimize the working set in-swap operation we described earlier. When a per-process store is present, the out-swap proceeds normally at UWP application suspend time except that no pagefile reservation is made. This is because the pages will go to the store manager virtual pagefile and sequentiality is not important since the allocated offsets are only used to construct keys to associate with pages. Later, when the UWP process working set is emptied due to memory pressure, all pages are compressed into the per-process store.

At this point, the compressed pages of the per-process store are out-swapped, reserving sequential space in the swapfile. If memory pressure continues, these compressed pages may be explicitly emptied or trimmed from the MemCompression process working set, get written out to the pagefile and remain cached on the standby list or leave memory. When the UWP application is about to be resumed, during working set in-swap, the system carefully choreographs disk read and decompression operations to maximize parallelism and efficiency. First, a store in-swap is kicked off to bring the compressed pages belonging to the store into the MemCompression process working set from the swapfile using large, sequential I/Os. Of course, if the compressed pages never left memory (which is very likely),

no actual I/Os need to be issued. In parallel, the working set in-swap for the UWP process is initiated, which uses multiple threads to decompress pages from the per-process store. The precise ordering of pages for these two operations ensures that they make progress in parallel with no unnecessary delays to reconstruct the UWP process working set quickly.

11.5.5 Memory Partitions

A memory partition is an instantiation of the memory manager with its own isolated slice of RAM to manage. Being kernel objects, they support naming and security. There are NT APIs for creating and managing them as well as allocating memory from them using partition handles. Memory can be hot-added into a partition or moved between partitions. At boot time, the system creates the initial memory partition called the **system partition** which owns all memory on the machine and houses the default instance of memory management. The system partition is actually named and can be seen in the object manager namespace at *\KernelObjects\MemoryPartition0*.

Memory partitions are mainly targeted at two scenarios: memory isolation and workload isolation. *Memory isolation* is when memory needs to be set aside for later allocation. For such scenarios, a memory partition can be created and appropriate memory can be added to it (e.g., a mix of 4 KB/2 MB/1 GB pages from select NUMA nodes). Later, pages can be allocated from the partition using regular physical memory allocation APIs which have variants that accept memory partition handles or object pointers. Azure servers which host customer VMs utilize this approach to set aside memory for VMs and ensure that other activity on the server is not going to interfere with that memory. It's important to understand that this is very different from simply pre-allocating these pages because the full set of memory management interfaces to allocate, free, and efficient zeroing of memory is available within the partition.

Workload isolation is necessary in situations where multiple separate workloads need to run concurrently without interfering with one another. In such scenarios, isolating the workloads' CPU usage (e.g., by affinitizing workloads to different processor cores) is not sufficient. Memory is another resource that needs isolation. Otherwise, one workload can easily interfere with others by repurposing all pages on the standby list (causing others to take more hard faults) or by dirtying lots of pagefile- and file-backed memory (depleting available memory and causing new memory allocations to block until dirty pages are written out) or by fragmenting physical memory and slowing down large or huge page allocations.

Memory partitions can provide the necessary workload isolation. By associating a memory partition with a job object, it is possible to confine a process tree to a memory partition and use the job object interfaces to set the desired CPU and disk I/O restrictions for complete resource isolation.

Being an instance of memory management, a memory partition includes the following major components as shown in Fig. 11-39:

1. **Page lists**: Each partition owns its isolated slice of physical memory, so it maintains its own free, zero, standby, and modified page lists.
2. **System process**: Each partition creates its own minimal system process called "PartitionSystem." This process provides the address space to map executables during load as well as housing per-partition system threads.
3. **System threads**: Fundamental memory management threads such as the zero page thread, the working set manager thread, the modified and mapped page writer threads are all created per-partition. In addition, other components such as the cache manager we will discuss in Sec. 11.6 also maintain per-partition threads. Finally, each partition has its dedicated system thread pool such that kernel components can queue work to it without worrying about contention from other workloads.
4. **Pagefiles**: Each partition has its own set of pagefiles and associated modified page writer thread. This is critical for maintaining its own commit.
5. **Resource tracking**: Each partition holds its own memory management resources such as commit and available memory to independently drive policies such as working set trimming and pagefile writing.

Notably, a memory partition does not include its own PFN database. Instead, it maintains a data structure describing the memory ranges it is responsible for and uses the system global PFN database entries. Also, most threads and data structures are initialized on demand. For example, the modified page writer thread is not necessary until a pagefile is created in the partition.

All in all, memory management is a highly complex executive component with many data structures, algorithms, and heuristics. It attempts to be largely self tuning, but there are also many knobs that administrators can tweak to affect system performance. A number of these knobs and the associated counters can be viewed using tools in the various tool kits mentioned earlier. Probably the most important thing to remember here is that memory management in real systems is a lot more than just one simple page replacement algorithm like clock or aging.

11.6 CACHING IN WINDOWS

The Windows cache improves the performance of file systems by keeping recently and frequently used regions of files in memory. Rather than cache physical addressed blocks from the disk, the cache manager manages virtually addressed

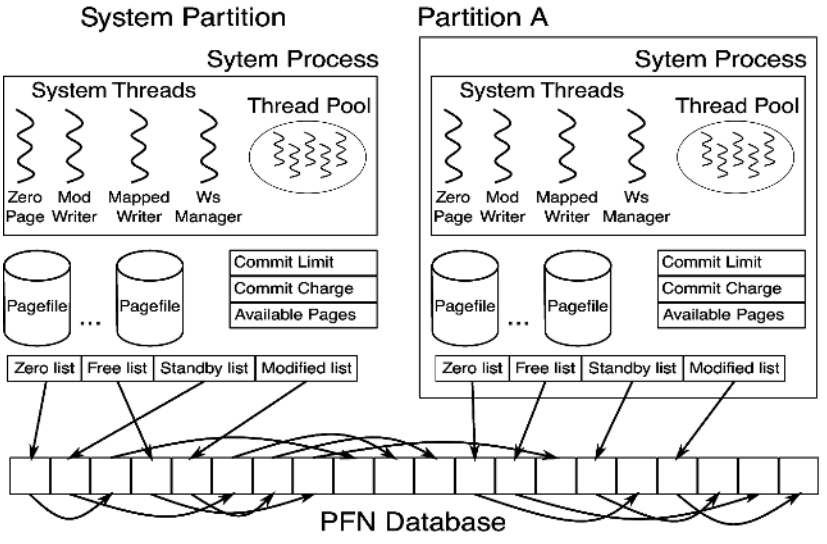

Figure 11-39. Memory partition data structures.

blocks, that is, regions of files. This approach fits well with the structure of the native NT File System (NTFS), as we will see in Sec. 11.8. NTFS stores all of its data as files, including the file-system metadata.

The cached regions of files are called *views* because they represent regions of kernel virtual addresses that are mapped onto file-system files. Thus, the actual management of the physical memory in the cache is provided by the memory manager. The role of the cache manager is to manage the use of kernel virtual addresses for views, arrange with the memory manager to pin pages in physical memory, and provide interfaces for the file systems.

The Windows cache-manager facilities are shared among all the file systems. Because the cache is virtually addressed according to individual files, the cache manager is easily able to perform read-ahead on a per-file basis. Requests to access cached data come from each file system. Virtual caching is convenient because the file systems do not have to first translate file offsets into physical block numbers before requesting a cached file page. Instead, the translation happens later when the memory manager calls the file system to access the page on disk.

Besides management of the kernel virtual address and physical memory resources used for caching, the cache manager also has to coordinate with file systems regarding issues like coherency of views, flushing to disk, and correct maintenance of the end-of-file marks—particularly as files expand. One of the most difficult aspects of a file to manage between the file system, the cache manager, and the memory manager is the offset of the last byte in the file, called the **ValidDataLength**. If a program writes past the end of the file, the blocks that were

skipped have to be filled with zeros, and for security reasons it is critical that the ValidDataLength recorded in the file metadata not allow access to uninitialized blocks, so the zero blocks have to be written to disk before the metadata is updated with the new length. While it is expected that if the system crashes, some of the blocks in the file might not have been updated from memory, it is not acceptable that some of the blocks might contain data previously belonging to other files.

Let us now examine how the cache manager works. When a file is referenced, the cache manager maps a 256-KB chunk of kernel virtual address space onto the file. If the file is larger than 256 KB, only a portion of the file is mapped at a time. If the cache manager runs out of 256-KB chunks of virtual address space, it must unmap an old file before mapping in a new one. Once a file is mapped, the cache manager can satisfy requests for its blocks by just copying from kernel virtual address space to the user buffer. If the block to be copied is not in physical memory, a page fault will occur and the memory manager will satisfy the fault in the usual way. The cache manager is not even aware of whether the block was in memory or not. The copy always succeeds.

The cache manager has various heuristics for detecting file access patterns. For example, when it detects a sequential access pattern, it starts performing **read-ahead** on behalf of the application such that data is ready by the time the application issues its I/O. This is very similar to the prefetching performed by the memory manager and uses the same underlying memory manager APIs.

Another important background operation the cache manager performs is **write-behind.** When dirty data accumulates in the cache manager's virtual address space, it starts proactively writing out the dirty data to disk to minimize the amount of lost data if, for example, the power goes out. Applications can always use the FlushFileBuffers Win32 API to flush all dirty data to disk; write-behind is a secondary measure. Another important benefit of write-behind is that the underlying pages can be more quickly reclaimed by the memory manager if available memory starts running low.

The cache manager also works for pages that are mapped into virtual memory and accessed with pointers rather than being copied between kernel and user-mode buffers. When a thread accesses a virtual address mapped to a file and a page fault occurs, the memory manager may in many cases be able to satisfy the access as a soft fault. It does not need to access the disk, since it finds that the page is already in physical memory because it is mapped by the cache manager.

11.7 INPUT/OUTPUT IN WINDOWS

The goals of the Windows I/O manager are to provide a fundamentally extensive and flexible framework for efficiently handling a very wide variety of I/O devices and services, support automatic device discovery and driver installation (plug and play) and efficient power management for devices and the CPU—all using a

fundamentally asynchronous structure that allows computation to overlap with I/O transfers. There are many hundreds of thousands of devices that work with Windows. For a large number of common devices, it is not even necessary to install a driver, because there is already a driver that shipped with the Windows operating system. But even so, counting all the revisions, there are almost a million distinct driver binaries that run on Windows. In the following sections, we will examine some of the issues relating to I/O.

11.7.1 Fundamental Concepts

The I/O manager is on intimate terms with the **plug-and-play manager**. The basic idea behind plug and play is that of an enumerable bus. Many buses, including PC Card, PCI, PCIe, AGP, USB, IEEE 1394, EIDE, SCSI, and SATA, have been designed so that the plug-and-play manager can send a request to each slot and ask the device there to identify itself. Having discovered what is out there, the plug-and-play manager allocates hardware resources, such as interrupt levels, locates the appropriate drivers, and loads them into memory. As each driver is loaded, a driver object is created for it. And then for each device, at least one device object is allocated. For some buses, such as SCSI, enumeration happens only at boot time, but for other buses, such as USB, it can happen at any time, requiring close cooperation between the plug-and-play manager, the bus drivers (which actually do the enumerating), and the I/O manager.

In Windows, all the file systems, antivirus filters, volume managers, network protocol stacks, and even kernel services that have no associated hardware are implemented using I/O drivers. The system configuration must be set to cause some of these drivers to load, because there is no associated device to enumerate on the bus. Others, like the file systems, are loaded by special code that detects they are needed, such as the file-system recognizer that looks at a raw volume and deciphers what type of file system format it contains.

An interesting feature of Windows is its support for **dynamic disks**. These disks may span multiple partitions and even multiple disks and may be reconfigured on the fly, without even having to reboot. In this way, logical volumes are no longer constrained to a single partition or even a single disk so that a single file system may span multiple drives in a transparent way. This property turned out to be difficult to support for software since a disk typically contains multiple partitions and thus multiple volumes, but with dynamic disks, a volume can span multiple disks and the underlying disks are individually visible to software as well, potentially causing confusion.

Starting with Windows 10, dynamic disks were effectively superseded by **storage spaces**, a new feature that provides virtualization of physical storage hardware. With storage spaces, a user can create **virtual disks** backed by potentially different underlying disk media, called the **storage pool**. The point is that these virtual disks are presented to the system as being actual disk device objects (as opposed to

virtual volumes presented by dynamic disks). This property makes storage spaces much more straightforward to work with.

Since its introduction, numerous features have been added to storage spaces beyond virtual disks. One interesting feature is called **thin provisioning**. This refers to the ability to create a virtual disk that is larger than the total size of the underlying storage pool. Actual physical storage is only allocated as the virtual disk is used. If the available space in the storage pool starts running low, the administrator is alerted and additional disks can be added to the pool at which point storage spaces will automatically redistribute allocated blocks between the new disks.

The I/O to volumes can be filtered by a special Windows driver to produce **volume shadow copies**. The filter driver creates a snapshot of the volume which can be separately mounted and represents a volume at a previous point in time. It does this by keeping track of changes after the snapshot point. This is very convenient for recovering files that were accidentally deleted, or traveling back in time to see the state of a file at periodic snapshots made in the past.

But shadow copies are also valuable for making accurate backups of server systems. The operating system works with server applications to have them reach a convenient point for making a clean backup of their persistent state on the volume. Once all the applications are ready, the system initializes the snapshot of the volume and then tells the applications that they can continue. The backup is made of the volume state at the point of the snapshot. And the applications were only blocked for a very short time rather than having to go offline for the duration of the backup.

Applications participate in the snapshot process, so the backup reflects a state that is easy to recover in case there is a future failure. Otherwise, the backup might still be useful, but the state it captured would look more like the state if the system had crashed. Recovering from a system error at the point of a crash can be more difficult or even impossible since crashes occur at arbitrary times in the execution of the application. *Murphy's Law* says that crashes are most likely to occur at the worst possible time, that is, when the application data is in a state where recovery is impossible.

Another aspect of Windows is its support for asynchronous I/O. It is possible for a thread to start an I/O operation and then continue executing in parallel with the I/O. This feature is especially important on servers. There are various ways the thread can find out that the I/O has completed. One is to specify an event object at the time the call is made and then wait on it eventually. Another is to specify a queue to which a completion event will be posted by the system when the I/O is done. A third is to provide a callback procedure that the system calls when the I/O has completed. A fourth is to poll a location in memory that the I/O manager updates when the I/O completes.

The final aspect that we will mention is prioritized I/O. I/O priority is determined by the priority of the issuing thread, or it can be explicitly set. There are

five priorities specified: *critical, high*, *normal*, *low*, and *very low*. Critical is reserved for the memory manager to avoid deadlocks that could otherwise occur when the system experiences extreme memory pressure. Low and very low priorities are used by background processes, like the disk defragmentation service and spyware scanners and desktop search, which are attempting to avoid interfering with normal operations of the system. Most I/O gets normal priority, but multimedia applications can mark their I/O as high to avoid glitches. Multimedia applications can alternatively use **bandwidth reservation** to request guaranteed bandwidth to access time-critical files, like music or video. The I/O system will provide the application with the optimal transfer size and the number of outstanding I/O operations that should be maintained to allow the I/O system to achieve the requested bandwidth guarantee.

11.7.2 Input/Output API Calls

The system call APIs provided by the I/O manager are not very different from those offered by most other operating systems. The basic operations are open, read, write, ioctl, and close, but there are also plug-and-play and power operations, operations for setting parameters, as well as calls for flushing system buffers, and so on. At the Win32 layer, these APIs are wrapped by interfaces that provide higher-level operations specific to particular devices. At the bottom, though, these wrappers open devices and perform these basic types of operations. Even some metadata operations, such as file rename, are implemented without specific system calls. They just use a special version of the ioctl operations. This will make more sense when we explain the implementation of I/O device stacks and the use of IRPs by the I/O manager.

The native NT I/O system calls, in keeping with the general philosophy of Windows, take numerous parameters and include many variations. Figure 11-40 lists the primary system-call interfaces to the I/O manager. NtCreateFile is used to open existing or new files. It provides security descriptors for new files, a rich description of the access rights requested, and gives the creator of new files some control over how blocks will be allocated. NtReadFile and NtWriteFile take a file handle, buffer, and length. They also take an explicit file offset and allow a key to be specified for accessing locked ranges of bytes in the file. Most of the parameters are related to specifying which of the different methods to use for reporting completion of the (possibly asynchronous) I/O, as described earlier.

NtQueryDirectoryFile is an example of a standard paradigm in the executive where various Query APIs exist to access or modify information about specific types of objects. In this case, it is file objects that refer to directories. A parameter specifies what type of information is being requested, such as a list of the names in the directory or detailed information about each file that is needed for an extended directory listing. Since this is really an I/O operation, all the standard ways of reporting that the I/O completed are supported. NtQueryVolumeInformationFile is

I/O system call	Description
NtCreateFile	Open new or existing files or devices
NtReadFile	Read from a file or device
NtWriteFile	Write to a file or device
NtQueryDirectoryFile	Request information about a directory, including files
NtQueryVolumeInformationFile	Request information about a volume
NtSetVolumeInformationFile	Modify volume information
NtNotifyChangeDirectoryFile	Finishes when any file in the directory or subtree is modified
NtQueryInformationFile	Request information about a file
NtSetInformationFile	Modify file information
NtLockFile	Lock a range of bytes in a file
NtUnlockFile	Remove a range lock
NtFsControlFile	Miscellaneous operations on a file
NtFlushBuffersFile	Flush in-memory file buffers to disk
NtCancelIoFile	Cancel outstanding I/O operations on a file
NtDeviceIoControlFile	Special operations on a device

Figure 11-40. Native NT API calls for performing I/O.

like the directory query operation, but expects a file handle which represents an open volume which may or may not contain a file system. Unlike for directories, there are parameters than can be modified on volumes, and thus there is a separate API NtSetVolumeInformationFile.

NtNotifyChangeDirectoryFile is an example of an interesting NT paradigm. Threads can do I/O to determine whether any changes occur to objects (mainly file-system directories, as in this case, or registry keys). Because the I/O is asynchronous the thread returns and continues, and is only notified later when something is modified. The pending request is queued in the file system as an outstanding I/O operation using an I/O Request Packet. Notifications are problematic if you want to remove a file-system volume from the system, because the I/O operations are pending. So Windows supports facilities for canceling pending I/O operations, including support in the file system for forcibly dismounting a volume with pending I/O.

NtQueryInformationFile is the file-specific version of the system call for directories. It has a companion system call, NtSetInformationFile. These interfaces access and modify all sorts of information about file names, file features like encryption and compression and sparseness, and other file attributes and details, including looking up the internal file id or assigning a unique binary name (object id) to a file.

These system calls are essentially a form of ioctl specific to files. The set operation can be used to rename or delete a file. But note that they take handles, not file

names, so a file first must be opened before being renamed or deleted. They can also be used to rename the alternative data streams on NTFS (see Sec. 11.8).

Separate APIs, NtLockFile and NtUnlockFile, exist to set and remove byte-range locks on files. NtCreateFile allows access to an entire file to be restricted by using a sharing mode. An alternative is these lock APIs, which apply mandatory access restrictions to a range of bytes in the file. Reads and writes must supply a *key* matching the key provided to NtLockFile in order to operate on the locked ranges.

Similar facilities exist in UNIX, but there it is discretionary whether applications heed the range locks. NtFsControlFile is much like the preceding Query and Set operations, but is a more generic operation aimed at handling file-specific operations that do not fit within the other APIs. For example, some operations are specific to a particular file system.

Finally, there are miscellaneous calls such as NtFlushBuffersFile. Like the UNIX sync call, it forces file-system data to be written back to disk. NtCancelIoFile cancels outstanding I/O requests for a particular file, and NtDeviceIoControlFile implements ioctl operations for devices. The list of operations is actually much longer. There are system calls for deleting files by name, and for querying the attributes of a specific file—but these are just wrappers around the other I/O manager operations we have listed and did not really need to be implemented as separate system calls. There are also system calls for dealing with **I/O completion ports**, a queuing facility in Windows that helps multithreaded servers make efficient use of asynchronous I/O operations by readying threads by demand and reducing the number of context switches required to service I/O on dedicated threads.

11.7.3 Implementation of I/O

The Windows I/O system consists of the plug-and-play services, the device power manager, the I/O manager, and the device-driver model. Plug-and-play detects changes in hardware configuration and builds or tears down the device stacks for each device, as well as causing the loading and unloading of device drivers. The device power manager adjusts the power state of the I/O devices to reduce system power consumption when devices are not in use. The I/O manager provides support for manipulating I/O kernel objects, and IRP-based operations like IoCallDrivers and IoCompleteRequest. But most of the work required to support Windows I/O is implemented by the device drivers themselves.

Device Drivers

To make sure that device drivers work well with the rest of Windows, Microsoft has defined the **WDM** (**Windows Driver Model**) that device drivers are expected to conform with. The **WDK** (**Windows Driver Kit**) contains examples and

documentation to help developers produce drivers which conform to the WDM. Most Windows drivers start out as copies of an appropriate sample driver from the WDK, which is then modified by the driver writer.

Microsoft also provides a **driver verifier** which validates many of the actions of drivers to be sure that they conform to the WDM requirements for the structure and protocols for I/O requests, memory management, and so on. The verifier ships with the system, and administrators can control it by running *verifier.exe*, which allows them to configure which drivers are to be checked and how extensive (i.e., expensive) the checks should be.

Even with all the support for driver development and verification, it is still very difficult to write even simple drivers in Windows, so Microsoft has built a system of wrappers called the **WDF** (**Windows Driver Foundation**) that runs on top of WDM and simplifies many of the more common requirements, mostly related to correct interaction with device power management and plug-and-play operations.

To further simplify driver writing, as well as increase the robustness of the system, WDF includes the **UMDF** (**User-Mode Driver Framework**) for writing drivers as services that execute in processes. And there is the **KMDF** (**Kernel-Mode Driver Framework**) for writing drivers as services that execute in the kernel, but with many of the details of WDM made automagical. Since underneath it is the WDM that provides the driver model, that is what we will focus on in this section.

Devices in Windows are represented by device objects. Device objects are also used to represent hardware, such as buses, as well as software abstractions like file systems, network protocol engines, and kernel extensions, such as antivirus filter drivers. All these are organized by producing what Windows calls a device stack, as previously shown in Fig. 11-14.

I/O operations are initiated by the I/O manager calling an executive API IoCallDriver with pointers to the top device object and to the IRP representing the I/O request. This routine finds the driver object associated with the device object. The operation types that are specified in the IRP generally correspond to the I/O manager system calls described earlier, such as create, read, and close.

Figure 11-41 shows the relationships for a single level of the device stack. For each of these operations, a driver must specify an entry point. IoCallDriver takes the operation type out of the IRP, uses the device object at the current level of the device stack to find the driver object, and indexes into the driver dispatch table with the operation type to find the corresponding entry point into the driver. The driver is then called and passed the device object and the IRP.

Once a driver has finished processing the request represented by the IRP, it has three options. It can call IoCallDriver again, passing the IRP and the next device object in the device stack. It can declare the I/O request to be completed and return to its caller. Or it can queue the IRP internally and return to its caller, having declared that the I/O request is still pending. This latter case results in an asynchronous I/O operation, at least if all the drivers above in the stack agree and also return to their callers.

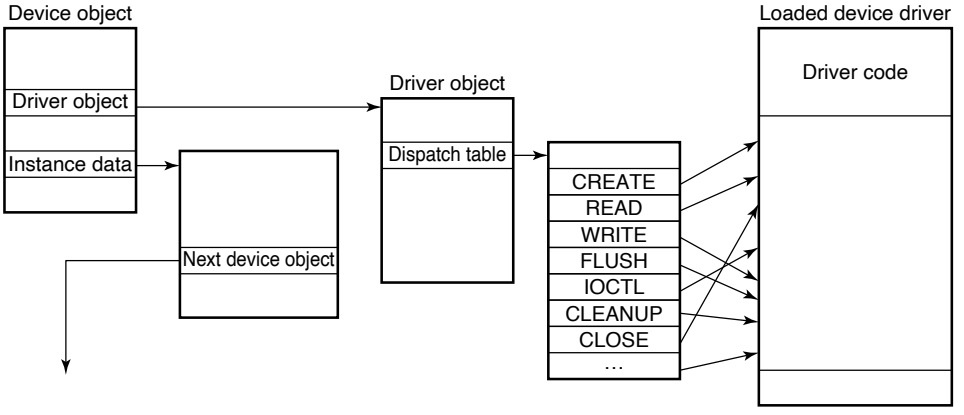

Figure 11-41. A single level in a device stack.

I/O Request Packets

Figure 11-42 shows the major fields in the IRP. The bottom of the IRP is a dynamically sized array containing fields that can be used by each driver for the device stack handling the request. These *stack* fields also allow a driver to specify the routine to call when completing an I/O request. During completion each level of the device stack is visited in reverse order, and the completion routine assigned by each driver is called in turn. At each level, the driver can continue to complete the request or decide there is still more work to do and leave the request pending, suspending the I/O completion for the time being.

When allocating an IRP, the I/O manager has to know how deep the particular of the stack depth in a field in each device object as the device stack is formed. Note that there is no formal definition of what the next device object is in any stack. That information is held in private data structures belonging to the previous driver on the stack. In fact, the stack does not really have to be a stack at all. At any layer a driver is free to allocate new IRPs, continue to use the original IRP, send an I/O operation to a different device stack, or even switch to a system worker thread to continue execution.

The IRP contains flags, an operation code for indexing into the driver dispatch table, buffer pointers for possibly both kernel and user buffers, and a list of **MDLs** (**Memory Descriptor Lists**) which are used to describe the physical pages represented by the buffers, that is, for DMA operations. There are fields used for cancellation and completion operations. The fields in the IRP that are used to queue the IRP to devices while it is being processed are reused when the I/O operation has finally completed to provide memory for the APC control object used to call the I/O manager's completion routine in the context of the original thread. There is also a link field used to link all the outstanding IRPs to the initiating thread.

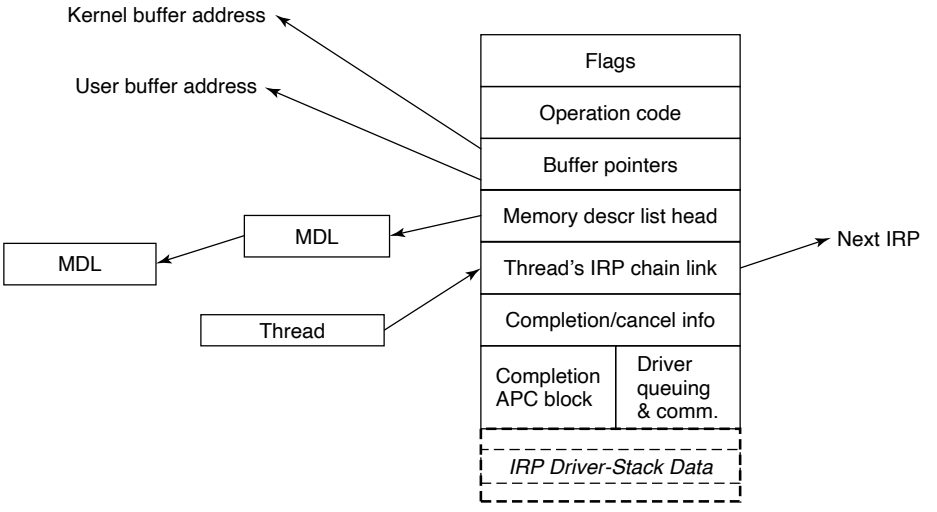

Figure 11-42. The major fields of an I/O Request Packet.

Device Stacks

A driver in Windows may do all the work by itself, or drivers may also be stacked, which means that a request may pass through a sequence of drivers, each doing part of the work. Two stacked drivers are also illustrated in Fig. 11-43.

One common use for stacked drivers is to separate the bus management from the functional work of controlling the device. Bus management on the PCI bus is quite complicated on account of many kinds of modes and bus transactions. By separating this work from the device-specific part, driver writers are freed from learning how to control the bus. They can just use the standard bus driver in their stack. Similarly, USB and SCSI drivers have a device-specific part and a generic part, with common drivers being supplied by Windows for the generic part.

Another use of stacking drivers is to be able to insert **filter drivers** into the stack. We have already looked at the use of file-system filter drivers, which are inserted above the file system. Filter drivers are also used for managing physical hardware. A filter driver performs some transformation on the operations as the IRP flows down the device stack, as well as during the completion operation with the IRP flows back up through the completion routines each driver specified. For example, a filter driver could compress data on the way to the disk or encrypt data on the way to the network. Putting the filter here means that neither the application program nor the true device driver has to be aware of it, and it works automatically for all data going to (or coming from) the device.

Kernel-mode device drivers are a serious problem for the reliability and stability of Windows. Most of the kernel crashes in Windows are due to bugs in device

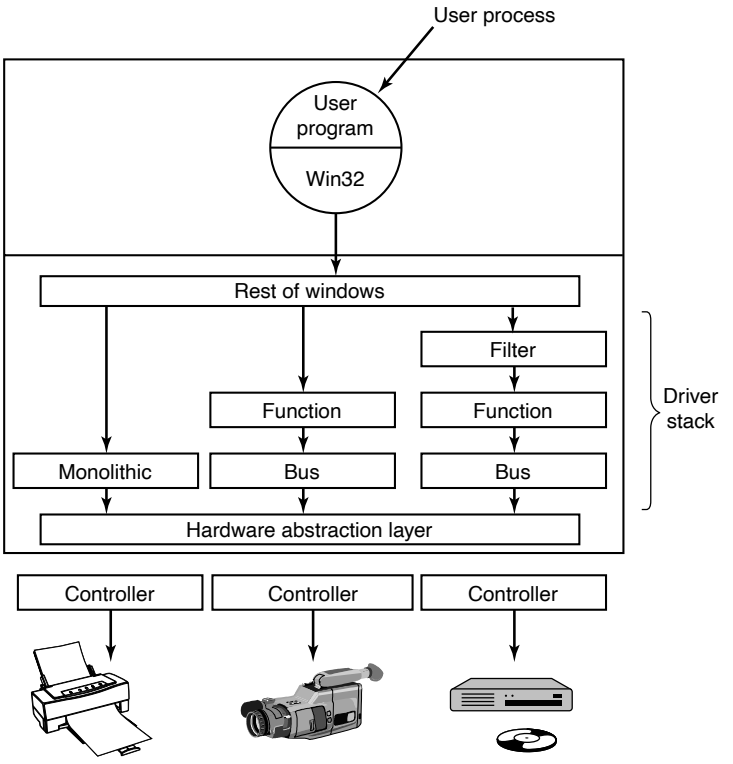

Figure 11-43. Windows allows drivers to be stacked to work with a specific instance of a device. The stacking is represented by device objects.

drivers. Because kernel-mode device drivers all share the same address space with the kernel and executive layers, errors in the drivers can corrupt system data structures, or worse, create security vulnerabilities. Some of these bugs are due to the astonishingly large numbers of device drivers that exist for Windows, or to the development of drivers by less-experienced system programmers. The bugs are also due to the enormous amount of detail involved in writing a correct driver for Windows.

The I/O model is powerful and flexible, but all I/O is fundamentally asynchronous, so race conditions can abound. Windows 2000 added the plug-and-play and device power management facilities from the Win9x systems to the NT-based Windows for the first time. This put a large number of requirements on drivers to deal correctly with devices coming and going while I/O packets are in the middle of being processed. Users of PCs frequently dock/undock devices, close the lid and toss notebooks into briefcases, and generally do not worry at all about whether the little green activity light happens to still be on. Writing device drivers that work

correctly in this environment can be very challenging, which is why WDF was developed to simplify the Windows Driver Model.

Many books are available about the Windows Driver Model and the newer Windows Driver Foundation (Orwick and Smith, 2007; Viscarola et al., 2007; Kanetkar, 2008; Vostokov, 2009; Reeves, 2010; and Yosifovich, 2019).

11.8 THE WINDOWS NT FILE SYSTEM

Windows supports several file systems, the most important of which are **FAT-16**, **FAT-32**, **NTFS** (**NT File System**), and **ReFS** (**Resilient File System**). FAT stands for **File Access Table**. FAT-16 is the old MS-DOS file system. It uses 16-bit disk addresses, which limits it to disk partitions no larger than 2 GB. It was primarily used for floppy disks. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. There is no security in FAT-32 and today it is really used only for transportable media, like flash drives. NTFS is the file system developed specifically for the NT version of Windows. Starting with Windows XP it became the default file system installed by most computer manufacturers, greatly improving the security and functionality of Windows. NTFS uses 64-bit disk addresses and can (theoretically) support disk partitions up to 2^{64} bytes, although other considerations limit it to smaller sizes.

ReFS is the newest file system in this group and initially shipped with Windows Server 2012 R2 which coincides with Windows 8.1. It's called the Resilient File System because one of its design goals is to be *self-healing*. ReFS can verify and automatically repair itself without downtime. This is achieved by maintaining integrity metadata for its on disk structures as well as user data. It's a nonoverwriting file system which means that metadata on disk is never updated in place; instead the new one is written elsewhere and the old version is marked deleted. When paired with storage spaces, ReFS supports the concept of *tiering* of user data and file system metadata meaning that it can keep "hot" data in faster disks and move "cold" data to slower disks automatically. Since ReFS is not used as the default file system for Windows yet, we will not study it in detail.

In this chapter, we will examine the NTFS file system because it is the default file system for Windows and a modern one with many interesting features and design innovations. It is large and complex and space limitations prevent us from covering all of its features, but the material presented below should give a reasonable impression of it.

11.8.1 Fundamental Concepts

Individual file names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Greece, Japan, India, Russia, and Israel) to write

file names in their native language. For example, *φιλε* is a perfectly legal file name. NTFS fully supports case-sensitive names (so *foo* is different from *Foo* and *FOO*). The Win32 API does not support case-sensitivity fully for file names and not at all for directory names. The support for case sensitivity exists when running the POSIX subsystem in order to maintain compatibility with UNIX. Win32 is not case sensitive, but it is case preserving, so file names can have different case letters in them. Though case sensitivity is a feature that is very familiar to users of UNIX, it is largely inconvenient to ordinary users who do not make such distinctions normally. For example, the Internet is largely case-insensitive today.

An NTFS file is not just a linear sequence of bytes, as FAT-32 and UNIX files are. Instead, a file consists of multiple attributes, each represented by a stream of bytes. Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data. However, a file can also have two or more (long) data streams as well. Each stream has a name consisting of the file name, a colon, and the stream name, as in *foo:stream1*. Each stream has its own size and is lockable independently of all the other streams. The idea of multiple streams in a file is not new in NTFS. The file system on the Apple Macintosh used two streams per file, the data fork and the resource fork. The first use of multiple streams for NTFS was to allow an NT file server to serve Macintosh clients. Multiple data streams are also used to represent metadata about files, such as the thumbnail pictures of JPEG images that are available in the Windows GUI. But alas, the multiple data streams are fragile and frequently fall off files when they are transported to other file systems, transported over the network, or even when backed up and later restored, because many utilities ignore them.

NTFS is a hierarchical file system, similar to the UNIX file system. The separator between component names is "\", however, instead of "/", an old fossil inherited from the compatibility requirements with CP/M when MS-DOS was created (CP/M used the slash for flags). Unlike UNIX the concept of the current working directory, hard links to the current directory (.) and the parent directory (..) are implemented as conventions rather than as a fundamental part of the file-system.

Hard links and symbolic links are supported for NTFS. Creation of symbolic links is normally restricted to administrators to avoid security issues like spoofing, as UNIX experienced when symbolic links were first introduced in 4.2BSD. The implementation of symbolic links uses an NTFS feature called reparse points (discussed later in this section). In addition, compression, encryption, fault tolerance, journaling, and sparse files are also supported. These features and their implementations will be discussed shortly.

11.8.2 Implementation of the NT File System

NTFS is a highly complex and sophisticated file system that was developed specifically for NT as an alternative to the HPFS file system that had been developed for OS/2. While most of NT was designed on dry land, NTFS is unique

among the components of the operating system in that much of its original design took place aboard a sailboat out on the Puget Sound (following a strict protocol of work in the morning, beer in the afternoon). Below we will examine a number of features of NTFS, starting with its structure, then moving on to file-name lookup, file compression, journaling, and file encryption.

File System Structure

Each NTFS volume (e.g., disk partition) contains files, directories, bitmaps, and other data structures. Each volume is organized as a linear sequence of blocks (clusters in Microsoft's terminology), with the block size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Most NTFS disks use 4-KB blocks as a compromise between large blocks (for efficient transfers) and small blocks (for low internal fragmentation). Blocks are referred to by their offset from the start of the volume using 64-bit numbers.

The principal data structure in each volume is the **MFT** (**Master File Table**), which is a linear sequence of fixed-size 1-KB records. Each MFT record describes one file or one directory. It contains the file's attributes, such as its name and timestamps, and the list of disk addresses where its blocks are located. If a file is extremely large, it is sometimes necessary to use two or more MFT records to contain the list of all the blocks, in which case the first MFT record, called the **base record**, points to the additional MFT records. This overflow scheme dates back to CP/M, where each directory entry was called an extent. A bitmap keeps track of which MFT entries are free.

The MFT is itself a file and as such can be placed anywhere within the volume, thus eliminating the problem with defective sectors in the first track. Furthermore, the file can grow as needed, up to a maximum size of 2^{48} records.

The MFT is shown in Fig. 11-44. Each MFT record consists of a sequence of (attribute header, value) pairs. Each attribute begins with a header telling which attribute this is and how long the value is. Some attribute values are variable length, such as the file name and the data. If the attribute value is short enough to fit in the MFT record, it is placed there. If it is too long, it is placed elsewhere on the disk and a pointer to it is placed in the MFT record. This makes NTFS very efficient for small files, that is, those that can fit within the MFT record itself.

The first 16 MFT records are reserved for NTFS metadata files, as illustrated in Fig. 11-45. Each record describes a normal file that has attributes and data blocks, just like any other file. Each of these files has a name that begins with a dollar sign to indicate that it is a metadata file. The first record describes the MFT file itself. In particular, it tells where the blocks of the MFT file are located so that the system can find the MFT file. Clearly, Windows needs a way to find the first block of the MFT file in order to find the rest of the file-system information. The way it finds the first block of the MFT file is to look in the boot block, where its address is installed when the volume is formatted with the file system.

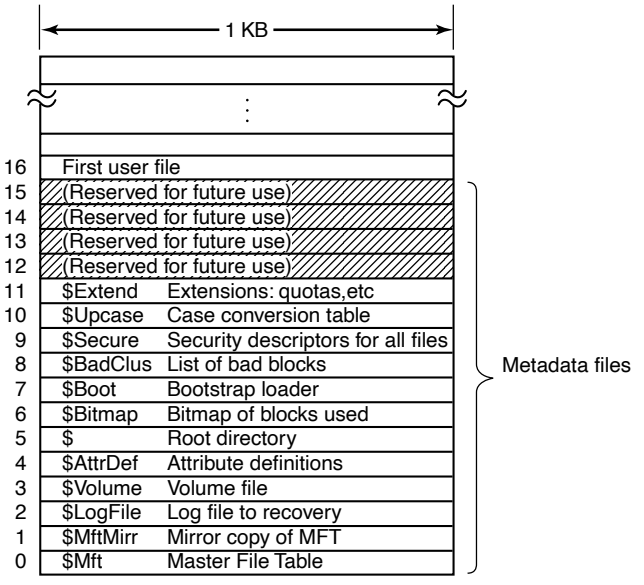

Figure 11-44. The NTFS master file table.

Record 1 is a duplicate of the early portion of the MFT file. This information is so precious that having a second copy can be critical in the event one of the first blocks of the MFT ever becomes unreadable. Record 2 is the log file. When structural changes are made to the file system, such as adding a new directory or removing an existing one, the action is logged here before it is performed, in order to increase the chance of correct recovery in the event of a failure during the operation, such as a system crash. Changes to file attributes are also logged here. In fact, the only changes not logged here are changes to user data. Record 3 contains information about the volume, such as its size, label, and version.

As mentioned above, each MFT record contains a sequence of (attribute header, value) pairs. The *$AttrDef* file is where the attributes are defined. Information about this file is in MFT record 4. Next comes the root directory, which itself is a file and can grow to arbitrary length. It is described by MFT record 5.

Free space on the volume is kept track of with a bitmap. The bitmap is itself a file, and its attributes and disk addresses are given in MFT record 6. The next MFT record points to the bootstrap loader file. Record 8 is used to link all the bad blocks together to make sure they never occur in a file. Record 9 contains the security information. Record 10 is used for case mapping. For the Latin letters A-Z case mapping is obvious (at least for people who speak Latin). Case mapping for other languages, such as Greek, Armenian, or Georgian (the country, not the state), is less obvious to Latin speakers, so this file tells how to do it. Finally, record 11 is a

directory containing miscellaneous files for things like disk quotas, object identifiers, reparse points, and so on. The last four MFT records are reserved for future use.

Each MFT record consists of a record header followed by the (attribute header, value) pairs. The record header contains a magic number used for validity checking, a sequence number updated each time the record is reused for a new file, a count of references to the file, the actual number of bytes in the record used, the identifier (index, sequence number) of the base record (used only for extension records), and some other miscellaneous fields.

NTFS defines 13 attributes that can appear in MFT records. These are listed in Fig. 11-45. Each attribute header identifies the attribute and gives the length and location of the value field along with a variety of flags and other information. Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT record, it may be put in separate disk blocks. Such an attribute is said to be a **nonresident attribute**. The data attribute is an obvious candidate. Some attributes, such as the name, may be repeated, but all attributes must appear in a fixed order in the MFT record. The headers for resident attributes are 24 bytes long; those for nonresident attributes are longer because they contain information about where to find the attribute on disk.

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Figure 11-45. The attributes used in MFT records.

The standard information field contains the file owner, security information, the timestamps needed by POSIX, the hard-link count, the read-only and archive bits, and so on. It is a fixed-length field and is always present. The file name is a variable-length Unicode string. In order to make files with non–MS-DOS names accessible to old 16-bit programs, files can also have an 8 + 3 MS-DOS **short**

name. If the actual file name conforms to the MS-DOS 8 + 3 naming rule, a secondary MS-DOS name is not needed.

In NT 4.0, security information was put in an attribute, but in Windows 2000 and later, security information all goes into a single file so that multiple files can share the same security descriptions. This results in significant savings in space within most MFT records and in the file system overall because the security info for so many of the files owned by each user is identical.

The attribute list is needed in case the attributes do not fit in the MFT record. This attribute then tells where to find the extension records. Each entry in the list contains a 48-bit index into the MFT telling where the extension record is and a 16-bit sequence number to allow verification that the extension record and base records match up.

NTFS files have an ID associated with them that is like the i-node number in UNIX. Files can be opened by ID, but the IDs assigned by NTFS are not always useful when the ID must be persisted because it is based on the MFT record and can change if the record for the file moves (e.g., if the file is restored from backup). NTFS allows a separate object ID attribute which can be set on a file and never needs to change. It can be kept with the file if it is copied to a new volume.

The **reparse point** tells the procedure parsing the file name that it has do something special. This mechanism is used for explicitly mounting file systems and for symbolic links. The two volume attributes are used only for volume identification. The next three attributes deal with how directories are implemented. Small ones are just lists of files but large ones are implemented using B+ trees. The logged utility stream attribute is used by the encrypting file system.

Finally, we come to the attribute that is the most important of all: the data stream (or in some cases, streams). An NTFS file has one or more data streams associated with it. This is where the payload is. The **default data stream** is unnamed (i.e., *dirpath\file name::$DATA*), but the **alternate data streams** each have a name, for example, *dirpath\file name:streamname:$DATA*.

For each stream, the stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the stream contains, or for streams of only a few hundred bytes (and there are many of these), the stream itself. Putting the actual stream data in the MFT record is called an **immediate file** (Mullender and Tanenbaum, 1984).

Of course, most of the time the data does not fit in the MFT record, so this attribute is usually nonresident. Let us now take a look at how NTFS keeps track of the location of nonresident attributes, in particular, data.

Storage Allocation

The model for keeping track of disk blocks is that they are assigned in runs of consecutive blocks, where possible, for efficiency reasons. For example, if the first logical block of a stream is placed in block 20 on the disk, then the system will try

hard to place the second logical block in block 21, the third logical block in 22, and so on. One way to achieve these runs is to allocate disk storage several blocks at a time, when possible.

The blocks in a stream are described by a sequence of records, each one describing a sequence of logically contiguous blocks. For a stream with no holes in it, there will be only one such record. Streams that are written in order from beginning to end all belong in this category. For a stream with one hole in it (e.g., only blocks 0–49 and blocks 60–79 are defined), there will be two records. Such a stream could be produced by writing the first 50 blocks, then seeking forward to logical block 60 and writing another 20 blocks. When a hole is read back, all the missing bytes are zeros. Files with holes are called **sparse files**.

Each record begins with a header giving the offset of the first block within the stream. Next comes the offset of the first block not covered by the record. In the example above, the first record would have a header of (0, 50) and would provide the disk addresses for these 50 blocks. The second one would have a header of (60, 80) and would provide the disk addresses for these 20 blocks.

Each record header is followed by one or more pairs, each giving a disk address and run length. The disk address is the offset of the disk block from the start of its partition; the run length is the number of blocks in the run. As many pairs as needed can be in the run record. Use of this scheme for a three-run, nine-block stream is illustrated in Fig. 11-46.

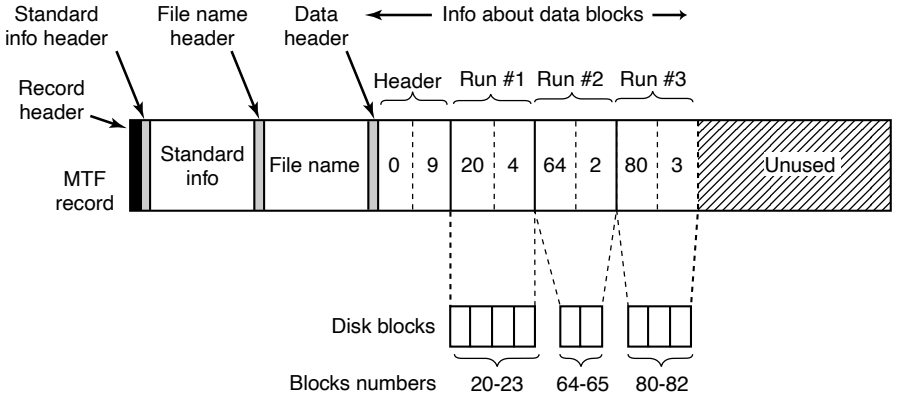

Figure 11-46. An MFT record for a three-run, nine-block stream.

In this figure, we have an MFT record for a short stream of nine blocks (header 0–8). It consists of the three runs of consecutive blocks on the disk. The first run is blocks 20–23, the second is blocks 64–65, and the third is blocks 80–82. Each of these runs is recorded in the MFT record as a (disk address, block count) pair. How many runs there are depends on how well the disk block allocator did in finding runs of consecutive blocks when the stream was created. For an *n*-block stream, the number of runs can be anything from 1 through *n*.

Several comments are worth making here. First, there is no upper limit to the size of streams that can be represented this way. In the absence of address compression, each pair requires two 64-bit numbers in the pair for a total of 16 bytes. However, a pair could represent 1 million or more consecutive disk blocks. In fact, a 20-GB stream consisting of 20 separate runs of 1 million 1-KB blocks each fits easily in one MFT record, whereas a 60-KB stream scattered into 60 isolated blocks does not.

Second, while the straightforward way of representing each pair takes 2×8 bytes, a compression method is available to reduce the size of the pairs below 16. Many disk addresses have multiple high-order zero-bytes. These can be omitted. The data header tells how many are omitted, that is, how many bytes are actually used per address. Other kinds of compression are also used. In practice, the pairs are often only 4 bytes.

Our first example was easy: all the file information fit in one MFT record. What happens if the file is so large or highly fragmented that the block information does not fit in one MFT record? The answer is simple: use two or more MFT records. In Fig. 11-47, we see a file whose base record is in MFT record 102. It has too many runs for one MFT record, so it computes how many extension records it needs, say, two, and puts their indices in the base record. The rest of the record is used for the first k data runs.

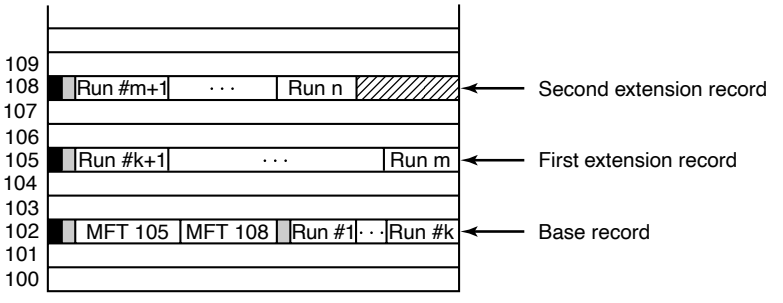

Figure 11-47. A file that requires three MFT records to store all its runs.

Note that Fig. 11-47 contains some redundancy. In theory, it should not be necessary to specify the end of a sequence of runs because this information can be calculated from the run pairs. The reason for "overspecifying" this information is to make seeking more efficient: to find the block at a given file offset, it is necessary to examine only the record headers, not the run pairs.

When all the space in record 102 has been used up, storage of the runs continues with MFT record 105. As many runs are packed in this record as fit. When this record is also full, the rest of the runs go in MFT record 108. In this way, many MFT records can be used to handle large fragmented files.

A problem arises if so many MFT records are needed that there is no room in the base MFT to list all their indices. There is also a solution to this problem: the

list of extension MFT records is made nonresident (i.e., stored in other disk blocks instead of in the base MFT record). Then it can grow as large as needed.

An MFT entry for a small directory is shown in Fig. 11-48. The record contains a number of directory entries, each of which describes one file or directory. Each entry has a fixed-length structure followed by a variable-length file name. The fixed part contains the index of the MFT entry for the file, the length of the file name, and a variety of other fields and flags. Looking for an entry in a directory consists of examining all the file names in turn.

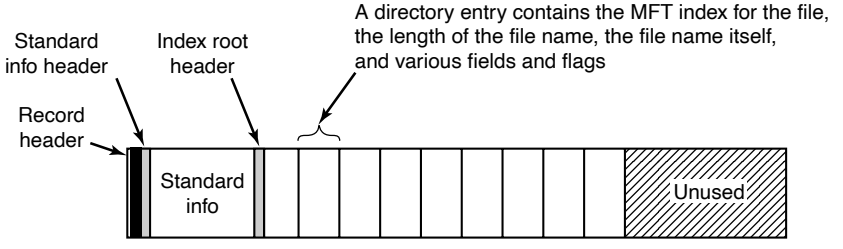

Figure 11-48. The MFT record for a small directory.

Large directories use a different format. Instead, of listing the files linearly, a B+ tree is used to make alphabetical lookup possible and to make it easy to insert new names in the directory in the proper place.

We now have enough information to finish describing how file-name lookup occurs for a file *\??\C:\foo\bar*. In Fig. 11-20, we saw how the Win32, the native NT system calls, and the object and I/O managers cooperated to open a file by sending an I/O request to the NTFS device stack for the *C:* volume. The I/O request asks NTFS to fill in a file object for the remaining path name, *\foo\bar*.

The NTFS parsing of the path *\foo\bar* begins at the root directory for *C:*, whose blocks can be found from entry 5 in the MFT (see Fig. 11-44). The string "foo" is looked up in the root directory, which returns the index into the MFT for the directory *foo*. This directory is then searched for the string "bar," which refers to the MFT record for this file. NTFS performs access checks by calling back into the security reference monitor, and if access checks pass, it searches the MFT record for the attribute *::$DATA*, which is the default data stream.

Having found file *bar*, NTFS will set pointers to its own metadata in the file object passed down from the I/O manager. The metadata includes a pointer to the MFT record, information about compression and range locks, various details about sharing, and so on. Most of this metadata is in data structures shared across all file objects referring to the file. A few fields are specific only to the current open, such as whether the file should be deleted when it is closed. Once the open has succeeded, NTFS calls IoCompleteRequest to pass the IRP back up the I/O stack to the I/O and object managers. Ultimately a handle for the file object is put in the handle table for the current process, and control is passed back to user mode. On

subsequent ReadFile calls, an application can provide the handle, specifying that this file object for *C:\foo\bar* should be included in the read request that gets passed down the *C:* device stack to NTFS.

In addition to regular files and directories, NTFS supports hard links in the UNIX sense, and also symbolic links using a mechanism called **reparse points**. NTFS supports tagging a file or directory as a reparse point and associating a block of data with it. When the file or directory is encountered during a file-name parse, the operation fails and the block of data is returned to the object manager. The object manager can interpret the data as representing an alternative path name and then update the string to parse and retry the I/O operation. This mechanism is used to support both symbolic links and mounted file systems, redirecting the search to a different part of the directory hierarchy or even to a different partition.

Reparse points are also used to tag individual files for file-system filter drivers. In Fig. 11-20, we showed how file-system filters can be installed between the I/O manager and the file system. I/O requests are completed by calling IoCompleteRequest, which passes control to the completion routines each driver represented in the device stack inserted into the IRP as the request was being made. A driver that wants to tag a file associates a reparse tag and then watches for completion requests for file open operations that failed because they encountered a reparse point. From the block of data that is passed back with the IRP, the driver can tell if this is a block of data that the driver itself has associated with the file. If so, the driver will stop processing the completion and continue processing the original I/O request. Generally, this will involve proceeding with the open request, but there is a flag that tells NTFS to ignore the reparse point and open the file.

File Compression

NTFS supports transparent file compression. A file can be created in compressed mode, which means that NTFS automatically tries to compress the blocks as they are written to disk and automatically uncompresses them when they are read back. Processes that read or write compressed files are completely unaware that compression and decompression are going on.

Compression works as follows. When NTFS writes a file marked for compression to disk, it examines the first 16 (logical) blocks in the file, irrespective of how many runs they occupy. It then runs a compression algorithm on them. If the resulting compressed data can be stored in 15 or fewer blocks, they are written to the disk, preferably in one run, if possible. If the compressed data still take 16 blocks, the 16 blocks are written in uncompressed form. Then blocks 16–31 are examined to see if they can be compressed to 15 blocks or fewer, and so on.

Figure 11-49(a) shows a file in which the first 16 blocks have successfully compressed to eight blocks, the second 16 blocks failed to compress, and the third 16 blocks have also compressed by 50%. The three parts have been written as three runs and stored in the MFT record. The "missing" blocks are stored in the MFT

entry with disk address 0 as shown in Fig. 11-49(b). Here the header (0, 48) is followed by five pairs, two for the first (compressed) run, one for the uncompressed run, and two for the final (compressed) run.

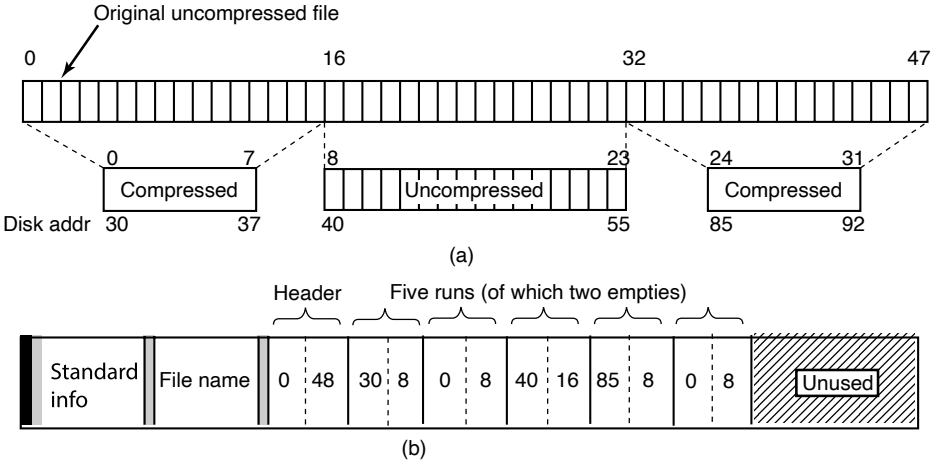

Figure 11-49. (a) An example of a 48-block file being compressed to 32 blocks. (b) The MFT record for the file after compression.

When the file is read back, NTFS has to know which runs are compressed and which ones are not. It can tell based on the disk addresses. A disk address of 0 indicates that it is the final part of 16 compressed blocks. Disk block 0 may not be used for storing data, to avoid ambiguity. Since block 0 on the volume contains the boot sector, using it for data is impossible anyway.

Random access to compressed files is actually possible, but tricky. Suppose that a process does a seek to block 35 in Fig. 11-49. How does NTFS locate block 35 in a compressed file? The answer is that it has to read and decompress the entire run first. Then it knows where block 35 is and can pass it to any process that reads it. The choice of 16 blocks for the compression unit was a compromise. Making it shorter would have made the compression less effective. Making it longer would have made random access more expensive. Because of this trade-off, it is generally better to use NTFS compression on files that are not randomly accessed.

Journaling

NTFS supports two mechanisms for programs to detect changes to files and directories. First is an operation, NtNotifyChangeDirectoryFile, that passes a buffer and returns when a change is detected to a directory or directory subtree. The result is that the buffer has a list of *change records*. If it is too small, records are lost.

The second mechanism is the NTFS change journal. NTFS keeps a list of all the change records for directories and files on the volume in a special file, which programs can read using special file-system control operations, that is, the *FSCTL_QUERY_USN_JOURNAL* option to the NtFsControlFile API. The journal file is normally very large, and there is little likelihood that entries will be reused before they can be examined. However, if journal entries do get reused before an application can examine them, then the app just needs to enumerate the directory tree it is interested in, to sync up with its state. After that, it can resume using the journal.

File Encryption

Computers are used nowadays to store all kinds of sensitive data, including plans for corporate takeovers, tax information, and love letters, which the owners do not especially want revealed to anyone. Information loss can happen when a notebook computer is lost or stolen, a desktop system is rebooted using an MS-DOS floppy disk to bypass Windows security, or a hard disk is physically removed from one computer and installed on another one with an insecure operating system.

Windows addresses these problems by providing an option to encrypt files, so that even in the event the computer is stolen or rebooted using MS-DOS, the files will be unreadable. The normal way to use Windows encryption is to mark certain directories as encrypted, which causes all the files in them to be encrypted, and new files moved to them or created in them to be encrypted as well. The actual encryption and decryption are not managed by NTFS itself, but by a driver called **EFS** (**Encryption File System**), which registers callbacks with NTFS.

EFS provides encryption for specific files and directories. There is also another encryption facility in Windows called **BitLocker** which runs as a block filter driver and encrypts almost all the data on a volume, which can help protect data no matter what—as long as the user takes advantage of the mechanisms available for strong keys. Given the number of systems that are lost or stolen all the time, and the great sensitivity to the issue of identity theft, making sure secrets are protected is very important. An amazing number of notebooks go missing every day. Major Wall Street companies supposedly average losing one notebook per week in taxicabs in New York City alone.

11.9 WINDOWS POWER MANAGEMENT

The **power manager** supervises power usage throughout the system. Historically management of power consumption consisted of shutting off the monitor display and stopping the disk drives from spinning. But the issue is rapidly becoming more complicated due to requirements for extending how long notebooks can run

on batteries, and energy-conservation concerns related to desktop computers being left on all the time and the high cost of supplying power to the huge server farms that exist today.

Newer power-management facilities include reducing the power consumption of components when the system is not in use by switching individual devices to standby states, or even powering them off completely using *soft* power switches. Multiprocessors shut down individual CPUs when they are not needed, and even the clock rates of the running CPUs can be adjusted downward to reduce power consumption. When a processor is idle, its power consumption is also reduced since it needs to do nothing except wait for an interrupt to occur.

On heterogeneous multiprocessor systems with multiple types of processors, significant power savings can be achieved by scheduling appropriate work to more efficient processors. The power manager closely collaborates with the kernel thread scheduler to influence its quality-of-service scheduling policies. For example, if the system is low on battery, the power manager can configure power policy such that all low-QoS threads exclusively get scheduled to efficiency cores.

Windows supports a special shut down mode called **hibernation**, which copies all of physical memory to disk and then shuts down the machine, reducing power consumption to zero. Because all the memory state is written to disk, you can even replace the battery on a notebook while it is hibernated. When the system resumes after hibernation, it restores the saved memory state (and reinitializes the I/O devices). This brings the computer back into the same state it was before hibernation, without having to login again and start up all the applications and services that were running. Windows optimizes this process by ignoring unmodified pages backed by disk already and compressing other memory pages to reduce the amount of I/O bandwidth required. The hibernation algorithm automatically tunes itself to balance between I/O and processor throughput. If there is more processor available, it uses expensive but more effective compression to reduce the I/O bandwidth needed. When I/O bandwidth is sufficient, hibernation will skip the compression altogether. With the current generation of multiprocessors, both hibernation and resume can be performed in a few seconds even on systems with many gigabytes of RAM.

An alternative to hibernation is **standby mode** where the power manager reduces the entire system to the lowest power state possible, using just enough power to the refresh the dynamic RAM. Because memory does not need to be copied to disk, this is somewhat faster than hibernation on some systems.

Despite the availability of hibernation and standby, many users are still in the habit of shutting down their PC when they finish working. Windows uses hibernation to perform a pseudo shutdown and startup, called **HiberBoot**, that is much faster than normal shutdown and startup. When the user tells the system to shutdown, HiberBoot logs the user off and then hibernates the system at the point they would normally login again. Later, when the user turns the system on again, HiberBoot will resume the system at the login point. To the user it looks like startup was

very, very fast because most of the system initialization steps are skipped. Of course, sometimes the system needs to perform a real shutdown in order to fix a problem or install an update to the kernel. If the system is told to reboot rather than shutdown, the system undergoes a real shutdown and performs a normal boot.

On phones and tablets, as well as the newest generation of laptops, computing devices are expected to be always on yet consume little power. To provide this experience, modern Windows implements a special version of power management called **CS** (**Connected Standby**). CS is possible on systems with special networking hardware which is able to listen for traffic on a small set of connections using much less power than if the CPU were running. A CS system always appears to be on, coming out of CS as soon as the screen is turned on by the user. Connected standby is different than the regular standby mode because a CS system will also come out of standby when it receives a packet on a monitored connection. Once the battery begins to run low, a CS system will go into the hibernation state to avoid completely exhausting the battery and perhaps losing user data.

Achieving good battery life requires more than just turning off the processor as often as possible. It is also important to keep the processor off as long as possible. The CS network hardware allows the processors to stay off until data have arrived, but other events can also cause the processors to be turned back on. In NT-based Windows device drivers, system services, and the applications themselves frequently run for no particular reason other than to *check on things*. Such *polling* activity is usually based on setting timers to periodically run code in the system or application. Timer-based polling can produce a cacophony of events turning on the processor. To avoid this, Windows requires that timers specify an imprecision parameter which allows the operating system to coalesce timer events and reduce the number of separate occasions one of the processors will have to be turned back on. Windows also formalizes the conditions under which an application that is not actively running can execute code in the background. Operations like checking for updates or freshening content cannot be performed solely by requesting to run when a timer expires. An application must defer to the operating system about when to run such background activities. For example, checking for updates might occur only once a day or at the next time the device is charging its battery. A set of system brokers provide a variety of conditions which can be used to limit when background activity is performed. If a background task needs to access a low-cost network or utilize a user's credentials, the brokers will not execute the task until the requisite conditions are present.

Many applications today are implemented with both local code and services in the cloud. Windows provides **WNS** (**Windows Notification Service**), which allows third-party services to push notifications to a Windows device in CS without requiring the CS network hardware to specifically listen for packets from the third party's servers. WNS notifications can signal time-critical events, such as the arrival of a text message or a VoIP call. When a WNS packet arrives, the processor will have to be turned on to process it, but the ability of the CS network hardware

to discriminate between traffic from different connections means the processor does not have to awaken for every random packet that arrives at the network interface.

11.10 VIRTUALIZATION IN WINDOWS

In the early 2000s, as computers were getting larger and more powerful, the industry started turning to virtual machine technology to partition large machine into a number of smaller virtual machines sharing the same physical hardware. This technology was originally used primarily in data centers or hosting environments. In the next decade, however, attention turned to more fine-grained software virtualization and containers came into fashion.

Docker Inc. popularized the use of containers on Linux with its popular Docker container manager. Microsoft added support for these types of containers to Windows in Windows 10 and Windows Server 2016 and partnered with Docker Inc. so that customers could use the same popular management platform on Windows. Additionally, Windows started shipping with the Microsoft Hyper-V hypervisor so that the OS itself could leverage hardware virtualization to increase security. In this section, we will first look at Hyper-V and its implementation of hardware virtualization. Then we will study containers built purely from software and describe some of the OS features that leverage hardware virtualization features.

11.10.1 Hyper-V

Hyper-V is Microsoft's virtualization solution for creating and managing virtual machines. The hypervisor sits at the bottom of the Hyper-V software stack and provides the core hardware virtualization functionality. It is a Type-1 (bare metal) hypervisor that runs directly on top of the hardware. The hypervisor uses virtualization extensions supported by the CPU to virtualize the hardware such that multiple *guest* operating systems can run concurrently, each in its own isolated virtual machine, called a **partition**. The hypervisor works with the other Hyper-V components in the **virtualization stack** to provide virtual machine management (such as startup, shutdown, pause, resume, live migration, snapshots, and device support). The virtualization stack runs in a special privileged partition called the **root partition**. The root partition must be running Windows, but any operating system, such as Linux, can be running in guest partitions which are also called **child partitions**. While it is possible to run guest operating systems that are completely unaware of virtualization, performance will suffer. Nowadays, most operating systems are **enlightened** to run as a guest and include guest counterparts to the root virtualization stack components which help provide higher-performance paravirtualized disk or network I/O. An overview of Hyper-V components is given in Fig. 11-50. We will discuss these components in the upcoming sections.

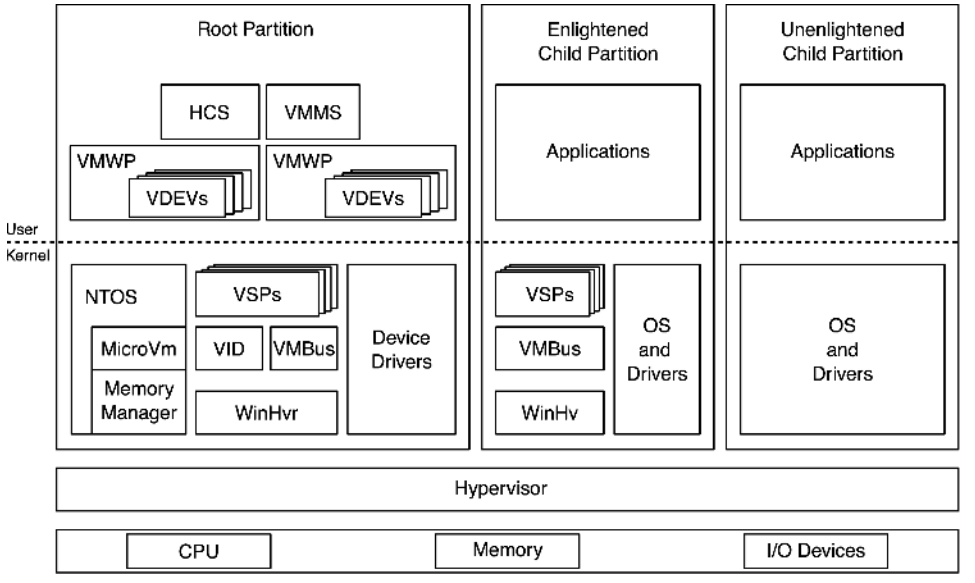

Figure 11-50. Hyper-V virtualization components in the root and child partitions.

Hypervisor

The hypervisor is a thin layer of software that runs between the hardware and the operating systems it is hosting. It is the most privileged software on the system and therefore needs to have a minimal attack surface. For this reason, it delegates as much functionality as possible to the virtualization stack running in the root partition.

The hypervisor's most important job is to virtualize hardware resources for its partitions: processors, memory, and devices. Each partition is assigned a set of **virtual processors (VPs)** and guest physical memory. The hypervisor manages these resources very similar to processes and threads in the operating system. The hypervisor internally represents each partition with a process data structure and each VP with a thread. From this perspective, each partition is an address space and each VP is a schedulable entity. As such, the hypervisor also includes a scheduler to schedule VPs on physical processors.

In order to virtualize processors and memory, the hypervisor relies on virtualization extensions provided by the underlying hardware. Intel, AMD, and ARM have slight variations in what they offer, but they are all conceptually similar. In a nutshell, the hardware defines a higher privilege level for the hypervisor and allows it to intercept various operations that occur while a processor is executing in guest mode. For example, when a clock interrupt occurs, the hypervisor gets control and can decide to switch out the currently running VP and pick another one, potentially

belonging to a different partition. Or, it can decide to inject the interrupt into the currently running VP for the guest OS to handle. Guest partitions can explicitly call the hypervisor—similar to how a user-mode process can make a system call into the kernel—using a **hypercall**, which is a trap to the hypervisor, analogous to a system call, which traps to the kernel.

For memory virtualization, the hypervisor takes advantage of **SLAT** (**Second Level Address Translation**) support provided by the CPU which essentially adds another level of page tables to translate **GPA**s (**Guest Physical Addresses**) to **SPA**s (**Server Physical Addresses**) This is known as **EPT** (**Extended Page Tables**) on Intel, **NPT** (**Nested Page Tables**) on AMD, and **stage 2 translation** on arm64. The hypervisor uses the SLAT to ensure that partitions cannot see each other's or the hypervisor's memory (unless explicitly desired). The SLAT for the root partition is set up in a 1:1 mapping such that root GPAs correspond to SPAs. The SLAT also allows the hypervisor to specify access rights (read, write, execute) on each translation which override any access rights the guest may have specified in it first-level page tables. This is important as we will see later.

When it comes to scheduling VPs on physical processors, the hypervisor supports three different schedulers:

1. **Classic scheduler**: The classic scheduler is the default scheduler used by the hypervisor. It schedules all non-idle VPs in round-robin fashion, but it allows adjustments such as setting *affinity* for VPs to a set of processors, reserving a percentage of processor capacity and setting limits and relative weights which are used when deciding which VP should run next.
2. **Core scheduler**: The core scheduler is relevant on CPUs that implement **SMT** (**Symmetric Multi-Threading**). SMT exposes two **LP**s (**Logical processors**), which share the resources of a single processor core. This is done to maximize utilization of processor hardware resources, but has two potentially significant downsides (so far). First, one SMT thread can impact the performance of its sibling because they share hardware resources like caches. Also, one SMT thread can use hardware side-channel vulnerabilities to infer data accessed by its sibling. For these reasons, it is not a great idea, from a performance and security isolation perspective, to run VPs belonging to different partitions on SMT siblings. That's the problem the core scheduler solves; it schedules an entire core, with all of its SMT threads to a single partition at a time. Typically, the partition is SMT-aware, so it has two VPs corresponding to the LPs in that core. Azure exclusively uses the core scheduler.
3. **Root scheduler**: When the root scheduler is enabled, the hypervisor itself does not do any VP scheduling. Instead, a virtualization stack component running in the root, known as the **VID** (**Virtualization**

Infrastructure Driver) creates a system thread for each guest VP, called **VP-backing threads** to be scheduled by the Windows thread scheduler. Whenever one of these threads gets to run, it makes a hypercall to tell the hypervisor to run the associated VP. Whereas the other schedulers treat guest VPs as black boxes—as should be the case for most virtual machine scenarios—the root scheduler allows for various **enlightenments** (paravirtualizations) enabling better integration between the guest and the host. For example, one enlightenment allows the guest to inform the host about the priorities of threads currently running on its VPs. The host scheduler can reflect these priority hints onto the corresponding VP-backing threads and schedule them accordingly, relative to other host threads. The root scheduler is enabled by default on client versions of Windows.

The Virtualization Stack

While the hypervisor provides hardware virtualization for guest partitions, it takes a lot more than that to run virtual machines. The virtualization stack, composed of several component across kernel-mode and user-mode, manages virtual machine memory, handles device access, and orchestrates VM states such as start, stop, suspend, resume, live migration, and snapshot.

As shown in Fig. 11-50, *WinHvr.sys* is the lowest layer of the virtualization stack in the root OS. Its enlightened guest counterpart is *WinHv.sys* in a Windows guest or *LinuxHv* in a Linux guest. It's the **hypervisor interface driver** which exposes APIs to facilitate communicating with the hypervisor rather than directly issuing hypercalls. It's the logical equivalent of *ntdll.dll* in user-mode which hides the system call interface behind a nicer set of exports.

VID.sys, the virtualization infrastructure driver, is responsible for managing memory for virtual machines. It exposes interfaces to user-mode virtualization stack components to construct the GPA space of a guest which includes regular guest memory as well as memory-mapped I/O space (MMIO). In response to these requests, the VID allocates physical memory from the kernel memory manager and asks the hypervisor, via WinHvr.sys, to map guest GPAs to those SPAs. The hypervisor needs physical memory to construct the SLAT hierarchy for each guest. The necessary memory for such metadata is allocated by the VID and *deposited* into the hypervisor, as necessary.

VMBus is another keykernel-mode virtualization stack component. Its job is to facilitate communication between partitions. It does this by setting up shared memory between partitions (e.g., a guest and the root) and taking advantage of **synthetic interrupt support** in the hypervisor to get an interrupt injected into the relevant partition when a message is pending. VMBus is used in paravirtualized I/O.

VSPs and *VSCs* are virtual service providers and clients that run in the root and guest partitions, respectively. The VSPs communicate with their guest counterparts

over VMBus to provide various services. The most common use of VSPs is for paravirtualized and accelerated devices, but other applications such as syncing time in the guest or implementing dynamic memory via ballooning also exist.

The user-mode virtualization components are for managing VMs as well as device support and orchestration of VM operations such as start, stop, pause, resume, live migration, snapshot, etc. **VMMS** (**Virtual Machine Management Service**) exposes interfaces for other management tools to query and manage virtual machines. *HCS* performs a similar task for containers. For each VM, VMMS creates a virtual machine worker process, *VMWP.exe*. VMWP manages the state of the VM and its state transitions. It includes **VDEVs** (**Virtual Devices**), which represent things like the virtual motherboard, disks, networking devices, BIOS, keyboard, mouse, etc. As the virtual machine boots and VDEVs are "powered on," they set up I/O ports or MMIO ranges in the GPA space through the VID driver or they communicate with their VSP driver to initiate VMBus channel set up with the corresponding VSC in the guest.

Device I/O

There are several ways Hyper-V can expose devices to its guests depending on how enlightened the guest OS is and the level of virtualization support in the hardware.

1. **Emulated devices**: An unenlightened guest communicates with devices through I/O ports or memory-mapped device registers. For emulated devices, the VDEV sets up these ports and GPA ranges to cause hypervisor intercepts when accessed. The intercepts are then forwarded to the VDEV running in the VM worker process through the VID driver. In response, the VDEV initiates the I/O requested by the guest and resumes the guest VP. Typically, when the I/O is complete, the VDEV will inject a synthetic interrupt into the guest via the VID and the hypervisor to signal completion. Emulated devices require too many context switches between the guest and the host and are not appropriate for high-bandwidth devices, but are perfectly OK for devices like keyboard and mouse.
2. **Paravirtualized devices**: When a synthetic device is exposed to a guest partition from its VDEV, an enlightened guest will load the corresponding VSC driver which sets up VMBus communication with its VSP in the root. A very common example of this is storage. *Virtual hard disks* are typically used with VMs and are exposed via the StorVSP and StorVSC drivers. Once the VMBus channel is set up, I/O requests received by the StorVSC are communicated to the StorVSP which then issues them to the corresponding virtual hard disk via the *vhdmp.sys* driver. Figure 11-51 illustrates this flow.

3. **Hardware-accelerated devices**: While paravirtualized I/O is much more efficient than device emulation, it still has too much root CPU overhead especially when it comes to today's high-end networking devices used in data centers or NVMe disks. Such devices support **SR-IOV** (**Single-Root I/O Virtualization**) or **DDA** (**Discrete Device Assignment**). Either way, the virtual PCI VDEV, working with the vPCI VSP/VSC, exposes the device to the guest on the virtual PCI bus. This is either a virtual function (VF) for SR-IOV devices or a physical function (PF) for DDA. The guest loads the corresponding device driver and is able to communicate directly with the device because its MMIO space is mapped into guest memory via the IOMMU. The IOMMU is also configured by the hypervisor to ensure that the device can only perform I/O to pages exposed to the guest.

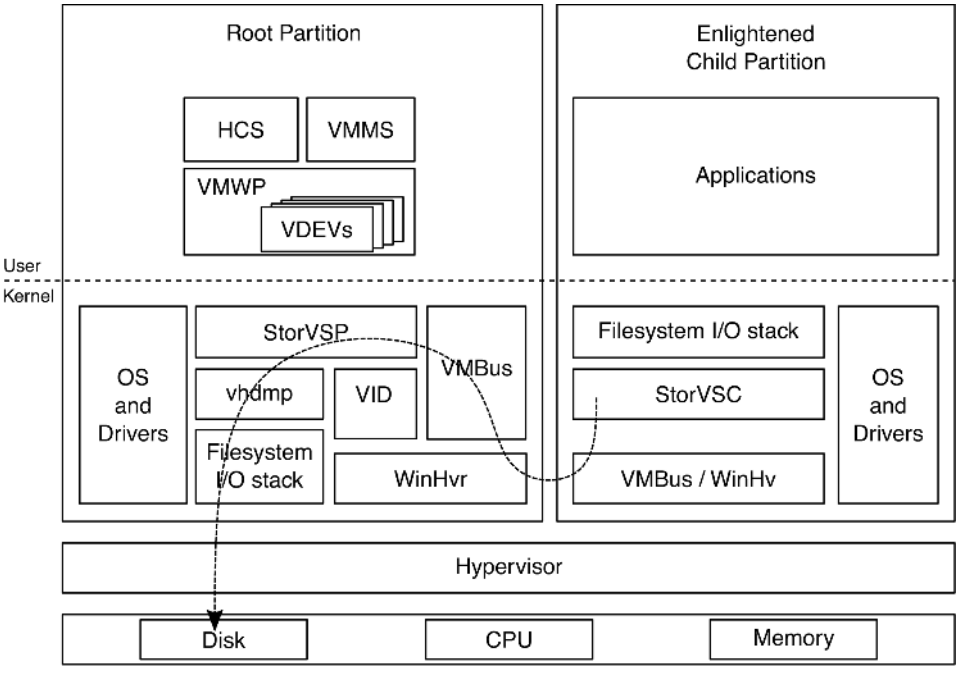

Figure 11-51. Flow of parvirtualized I/O for an enlightened guest OS.

VA-backed VMs

Typically, the VID driver allocates dedicated physical memory for each virtual machine and maps it into the GPA space through the SLAT. This memory belongs to the VM whether it is using it or not. Hyper-V also supports a different model for managing VM memory, called **VA-backed VMs**, which provides more flexibility.

Instead of allocating physical pages up-front, VA-backed VM GPA space is backed by *virtual memory* allocated from a minimal process (see Sec. 11.4.3) called **vmmem**. The VID creates a vmmem process for each VA-backed VM and allocates virtual memory in that process corresponding to the RAM size configured for the VM, using an internal variant of VirtualAlloc. The mapping between the vmmem virtual address range and the guest GPA space is managed by an NT kernel component called **MicroVm**, which is tightly integrated with the memory manager.

A VA-backed VM starts booting with a largely empty SLAT. As its VPs access guest physical pages, they hit SLAT page faults, leading to *memory intercepts* into the hypervisor which are forwarded to the VID and then to MicroVm. MicroVm determines the virtual address that correspond to the faulting GPA and asks the memory manager to perform regular demand-zero fault handling, which involves allocating a new physical page and updating the PTE corresponding to the vmmem virtual address. After the fault is resolved and the virtual address is added to the vmmem working set, MicroVm calls the hypervisor to update the SLAT mapping from the faulting GPA to the newly allocated page. After that, the VID can return back to the hypervisor, resolving the guest fault and resuming the guest VP.

The reverse can also happen. If the host memory manager decides to trim a valid page from the vmmem working set, MicroVm will ask the hypervisor to invalidate the SLAT mapping for the corresponding GPA. The next time guest accesses that GPA, it will take a SLAT fault which will need to be resolved as described earlier.

The design of VA-backed VMs allows the host memory management to treat the virtual machine (represented by the vmmem process) just like any other process and apply its memory management bag of tricks to it. Mechanisms like aging, trimming, paging, prefetching, page combining, and compression can be used to manage VM memory more efficiently.

VA-backed VMs enable another significant memory optimization: file sharing. While there are many applications of file sharing, a particularly important one is when multiple guests are running the same OS or when a guest is running the same OS as the host. Similar to how guest RAM is associated with a virtual address range in vmmem, a binary can be mapped to the vmmem address space using the equivalent of MapViewOfFile. The resulting address range is exposed to the guest as a new GPA range and the mapping is tracked by MicroVm. That way, accesses to the GPA range will result in memory intercepts which will be resolved by file pages backed by the binary. The critical point is that host processes that map the same file will use the exact same file page in physical memory.

So far, we described how a file mapping can be exposed to the guest as a GPA range while being shared by host processes (or by GPA ranges in other VMs). How does the guest use the GPA range as a file? In the guest, an enlightened file system driver (called *wcifs.sys* on Windows) takes advantage of a memory manager feature called **Direct Map** to expose CPU-addressable memory as file pages that the

memory manager can directly use. Rather than allocating new physical pages, copying file data into those pages and then pointing PTEs to them, the memory manager updates PTEs to point directly to the CPU-addressable file pages themselves. This mechanism allows all processes in the guest OS to share the same GPAs that were exposed from the vmmem file mapping. Figure 11-52 shows how VA-backed VM memory is organized.

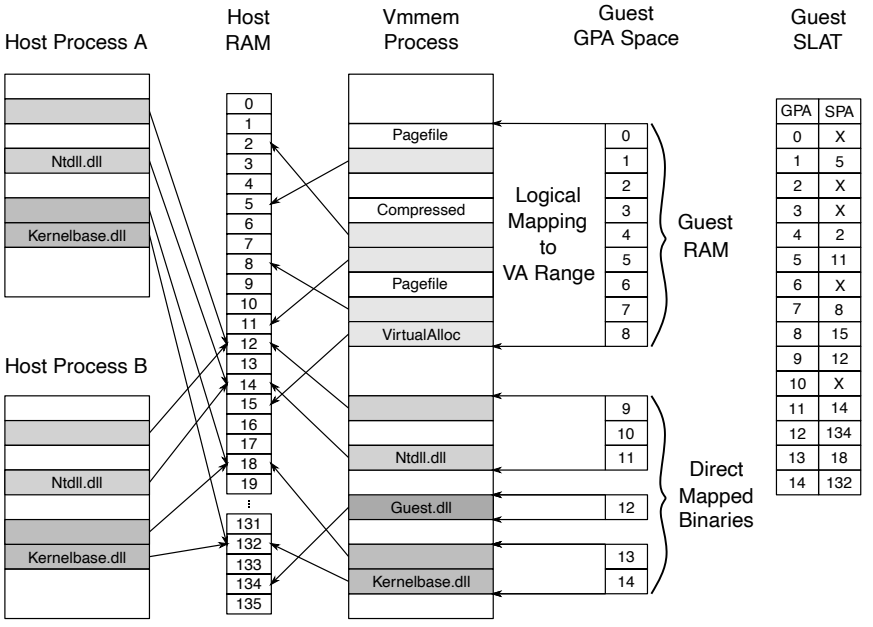

Figure 11-52. VA-backed VM's GPA space is logically mapped to virtual address ranges in its vmmem process on the host.

In addition to the optimizations described so far, the VA-backed VM design also allows various memory management enlightenments in the guest to further optimize memory usage. One important example is hot/cold memory enlightenments. Via hypercalls, the guest memory manager can provide hints to the host about GPAs that are more or less likely to get accessed soon. In response, the host can make sure that those pages are resident and valid in the SLAT (for "hot" pages) or trim them out of the vmmem working set (for "cold" pages). Windows guests take advantage of this enlightenment to cold-hint pages in the back of the zeroed page list. That results in the underlying host physical pages getting freed into the zero page list on the host because of the zero-page detection done by the memory manager during working set trim (see Sec. 11.5.3). Hot hints are used for pages at the head of the free, zero, and standby lists if these have previously been cold-hinted.

11.10.2 Containers

Hardware-based virtualization is very powerful but sometimes provides more isolation than desired. In many cases, it is preferable to have more fine-grained virtualization. Windows 10 added support for containers which leverages fine-grained software virtualization. This section will investigate a few uses for more fine-grained virtualization and then examine how it is implemented.

Earlier in Sec. 11.2.1 the modern app architecture was discussed, one of the benefits being reliable install/uninstall and the ability to deliver apps via the Microsoft Store. In Windows 8, only modern apps were delivered via the store—leaving out the enormous library of existing Windows applications. Microsoft wanted to provide a way for software vendors to package their existing applications to be delivered from the store while maintaining the benefits the store was meant to provide. The solution was to encourage applications to be distributed via MSIX packages and allow the installation of the application to be virtualized. Rather than requiring an installer to modify the file system and registry to install the app, those modifications would be virtualized. When the application is launched, the system creates a container with an alternate view of the file system and registry namespace that make it look like the application has been installed (to the application about to be run). If the user decides to uninstall the application, the MSIX package is deleted, but there is no longer a need to go and remove application files and state from the file system and registry.

Windows 10 also introduced a feature similar to Linux containers, known as **Windows Server Containers**. A Windows Server Container provides an environment that looks like a full virtual machine. The container gets its own IP address, can have its own computer name on the network, its own set of user accounts, etc. However, a Windows Server Container is much lighter weight than a VM because it shares the kernel with the host, only the user mode processes are replicated. These types of containers do not provide the same level of isolation as a VM but provide a very convenient deployment model and reduce the concern of running two applications that normally could not coexist.

Namespace Virtualization

The underlying technology that containers build on is known as namespace virtualization. Rather than virtualizing hardware, as VMs do, containers make it possible for one or more processes to run with a slightly different view of various namespaces.

To provide namespace virtualization support, Windows 10 introduced the notions of **Silos**. Silos are an extension to the **job object** (see Sec. 11.4.1) that allow for namespace virtualization. Silos make it possible to provide alternative views of namespaces to the processes running within them. Silos are the fundamental building block for implementing container support in Windows. There are

in fact two types of Silos. The first is known as an **Application Silo**. App Silos provide namespace virtualization only. A job is converted into a silo via a SetInformationJobObject API call to enable the namespace virtualization features on the job. Rather than require a separate call to promote a job object into a silo, Microsoft could have just changed the implementation of job objects such that all jobs had namespace virtualization support. However, that would have caused all job objects to require more memory so instead a pay for play model was adopted. The second type of silo is known as a **Server Silo**. Server silos are used to implement Windows server containers (see below). Because server containers provide the illusion of a full machine, some kernel mode state needs to be instanced per container. Server silos build on app silos in that in addition to namespace virtualization it also allows various kernel components to maintain separate copies of their state per container. Server silos require much more storage than an app silo so the pay-for-play model is again adopted so that this extra storage is only required for jobs promoted into full server silos.

When a job is created and promoted into an app silo, it is considered a namespace container. Prior to launching processes within the container, the various namespaces being virtualized must be configured. The most prominent namespaces are the file system and registry namespaces. Virtualizing these namespaces is done via filter drivers. During silo initialization, a user mode component will send IOCTLs to the various namespace filters to configure them for how to virtualize the given namespace. However, no container state is associated with the filters themselves. Instead, the model is to associate all state required to do namespace virtualization with the silo itself. During startup, namespace filter drivers request a silo slot index from the system and store it in a global variable. The silo then provides a key/value store to the drivers. They can store any object manager object (see Sec. 11.3.3) in the slot associated with their index. If a driver wants to store state that is not in the form of an object manager object, it can use the new kernel API PsCreateSiloContext to create an object with storage of the required size and pool type. The namespace filter packages up the state required for virtualizing the namespace and stores it in the silo slot for future reference.

Once all namespace providers are configured, the first application in the container is launched. As that application starts to run, it will inevitably start to access various namespaces. When an IO request reaches a given namespace, the namespace filter will check to see if virtualization is required. It will call the PsGetSiloContext API passing its slot index to retrieve any configuration required to virtualize the namespace. If the given namespace is not being virtualized for the running thread, then the call will return a status code indicating there is nothing in the slot, and the namespace filter will simply pass the IO request to the next driver in the stack (see Sec. 11.7.3 for details on driver stacks). However, if configuration information was found in the slot, the namespace filter will use it to determine how to virtualize the namespace. For example, the filter may need to modify the name of the file being opened before passing the request down the stack.

The benefit of associating all configuration with the silo and having the storage slots hold object manager objects is that cleanup is simple. When the last process in the silo goes away, and the last reference to the silo goes away the system can just run down the entries in each storage slot and drop the reference to the associated object. This is very similar to what the system does when a process exits, and its handle table is run down.

Server containers are a bit more complicated than application silos as many more namespaces must be virtualized to create the illusion of an isolated machine. For example, application silos typically share most namespaces with the host and often only need a few new resources inserted into the observed namespace. With server containers all namespaces must be virtualized. This includes the full object manager namespace, the file system and registry, the network namespace, the process and thread ID namespace, etc. If, for example, the network namespace was not virtualized a process in one container might use a port that a process in another container needed. By giving each container its own IP address and port space, such conflicts are avoided. Additionally, the process and thread ID namespaces are virtualized to avoid one container seeing or having access to processes and threads from another container.

In addition to the larger set of namespaces to be virtualized, server containers also require private copies of various kernel state. A Windows administrator can normally configure certain global system state that effects the entire machine. To provide an administrative process running within the container this same type of control, the kernel was updated to allow this state to apply per container rather than globally. The result is that much of the kernel state that was previously stored in global variables is now referenced per container. There is a notion of a **host container** which is where the host's state is stored.

Booting a server silo begins in the same way as creating an application silo. The job object is promoted into a silo and the namespace configuration is done. Unlike standard app containers, server containers get a full private object manager namespace. The root of the server containers namespace is an object manager directory on the host. This allows the host full visibility and access to the container which aids in management tasks. For example, the following directory may represent the root of a server container namespace: *\Silos\100*. In this example, 100 is the job identifier of the silo backing the server container. This directory is also prepopulated with a variety of objects such that the object manager namespace for the container will look like what the host's namespace looks like just before launching the first user mode process. Some of those objects are shared with the host and are exposed to the container with a special type of symbolic link that allow host objects to be accessed from within the container.

Once the container's namespace is setup, the next step is to promote the silo to a server silo. This is done with another SetInformationJobObject call. Promoting the silo to a server silo allocates additional data structures used to maintain instanced copies of kernel state. Then the kernel invokes enlightened kernel components

and give them an opportunity to initialize their state and do any other prep work required. If any of these steps fail, then the server silo boot fails, and the container is torn down.

Finally, the initial user mode process *smss.exe* is started within the container. At this p,oint the user mode portions of the OS boot up. A new instance of csrss.exe is started (the Win32 subsystem process), a new instance of lsass.exe (the local security authority subsystem), a new service control manager, etc. For the most part, everything works in the same way it would if booting user mode on the host. Some things are different in the container, though. For example, an interactive user session is not created—it is not needed since the container is headless. But these changes are just configuration changes, driven by existing mechanisms. The difference in behavior is because the virtualized registry state for the container is configured that way.

As the container boots, it is booting from a **VHD** (**Virtual Hard Disk**). However, that VHD is mostly empty. The file system virtualization driver, *wcifs.sys*, provides the appearance to the processes running within the container that the hard disk is fully populated. The backing store for the container's disk contents is spread across one or more directories on the host as illustrated in Fig. 11-53. Each of these host directories represents an image layer. The bottom-most layer is known as the **base layer** and is provided by Microsoft. Subsequent layers are various deltas to this bottom layer, potentially changing configuration settings in the virtualized registry hives, or additions, changes, or deletions (represented with special **tombstone files** to the file system. At runtime, the file system namespace filter merges each of these directories together to create the view exposed to the container. Each of these layers is immutable and can be shared across containers. As the container runs and makes changes to the file system, those changes are captured on the VHD exposed to the container. In this way, the VHD will contain deltas from the layers below. It is possible to later shut down the container and make a new layer based on the contents of the VHD. Or if the container is no longer needed, it can be disposed of, and all persisted side effects deleted.

Certain operations are blocked within a container. For example, a container is not allowed to load a kernel driver as doing so might allow an avenue to escape the containment. Additionally, certain functionality such as changing the time is blocked within the container. Typically, such operations are protected by privilege checks. These privilege checks are augmented when running in the container so that the operations that should be blocked within a container are blocked regardless of the privilege enabled in the caller's token. Other operations, such as changing the time zone, are allowed if the required privilege is held but the operation is virtualized so that only processes within the container use the new time zone.

A container can be terminated in a few ways. First, it can be terminated from the outside (via the management stack) which is like a forced shutdown. Second, it can be terminated from inside the container when a process calls a Win32 API to shut down Windows, such as ExitWindowsEx or InitiateSystemShutdown. When

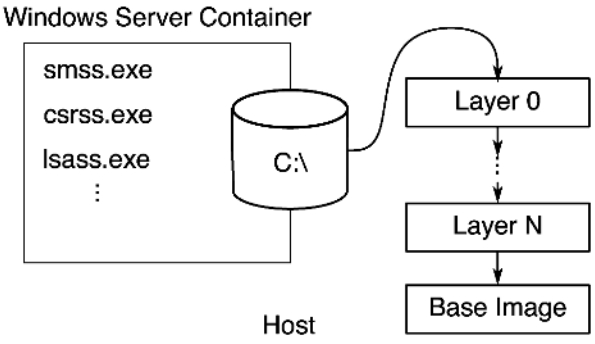

Figure 11-53. The contents of the VHD exposed to the container is backed by a set of host directories that are merged at runtime to make up the container file system contents.

the request to shut down the machine reaches the kernel and if the request originated in a container, the kernel terminates the container rather the shutting down the host. A container can also be shut down if a critical process within the container crashes. This would normally result in a host blue screen, but if the critical process was in a container, the container will be terminated rather than causing a blue screen.

Hyper-V Isolated Containers

Server Silos provide a high degree of isolation based on namespace isolation. Microsoft advertises these containers as being suitable for enterprise multitenancy or non-hostile workloads. However, there are times when it is desirable to run hostile workloads within a container. For those scenarios, **Hyper-V isolated containers** are the solution. These containers leverage hardware-based virtualization mechanism to provide a very secure boundary between the container and its host.

One of the primary design goals of Windows Containers was to not require an administrator to decide upfront what type of container to use. The same artifacts should be usable with either a Windows Server Container or a Hyper-V Isolated Container. The approach taken was to always run a server silo for the container, but in some cases, it is run on the host (Windows Server Containers) and in others, it is run within something known as a **Utility VM** (Hyper-V Isolated Containers). The Utility VM is created as a VA-backed VM to optimize memory usage and to allow in memory sharing of container base image binaries across running containers which significantly improves density.

The utility VM also runs a very scaled down OS instance, designed to host nothing other than server silos so that it boots quickly and uses minimal memory. When the Hyper-V isolated container is instantiated, the Utility VM is started first.

Then the **HCS** (**Host Compute Service**) communicates to the **GCS** (**Guest Compute Service**) running in the Utility VM and requests the server silo to be started.

Since Hyper-V isolated containers run their own copy of the Windows kernel in the Utility VM, even a hostile workload that manages to take advantage of a flaw in the Windows kernel will not be able to attack the host. Administrators can alternate between running a server container in either a process isolated or Hyper-V isolated scenario with one command line switch.

Hardware Isolated Processes

Windows 10 has also introduced support for running certain processes that represent a high attack surface within Hardware Isolated Containers in some Windows editions. **MDAG** (**Microsoft Defender Application Guard**) supports running the Edge browser within a hardware isolated container. The Edge team has worked very hard to protect users when navigating to a malicious website. However, Edge is also very large and complicated and so is the underlying OS. There will always be latent bugs that bad actors can try to exploit. By running the Edge browser in a Utility VM-type environment, malicious activity can be limited to the container. And since the container's side effects can be discarded after each run, it is possible to provide a pristine environment for each launch.

Unlike server containers which are headless, users need to see the Edge browser. This is achieved by leveraging a technology known as **RAIL** (**Remote Apps Integrated Locally**). The **RDP** (**Remote Desktop Protocol**) is used to remote the window for a single application, in this case the Edge browser, to the host. The effect is the user has the same experience as running Edge locally but with the backend processing done in a container. Copy and paste functionality is limited over RDP to avoid malicious attacks via the clipboard. Display performance is quite good due to shared memory between the host and the guest for display purposes, and a virtual GPU can even be exposed to the guest so that the guest can leverage the host GPU for rendering purposes.

In later versions of Windows 10, MDAG was extended to support running Microsoft Office applications as well. For other applications not supported directly by MDAG, there is a feature known as **Windows Sandbox**. Windows Sandbox uses the same underlying technology as MDAG and Hyper-V isolated containers but provides the user with a full desktop environment. The user can launch Windows Sandbox to run programs they are hesitant to run on the host.

MDAG and Windows Sandbox leverage the same OS instance installed on the host and when the host OS is serviced so is the MDAG/Sandbox environment. They also benefit from the same VA-backed VM optimizations listed above like direct mapped memory and integrated scheduler reducing the cost of running these relative to a classic VM.

VA-backed VMs are also used for running certain guest operating systems other than Windows. **WSL** (**Windows Subsystem for Linux**) and **WSA**

(**Windows Subsystem for Android**) are also built on VA-backed VMs to run Linux and Android operating systems on top of Windows in a more efficient way than regular VMs. While these operating systems do not (yet) implement all of the memory management and root scheduler enlightenments Windows guests do, they are able to take full advantage of host-side memory management optimizations like memory compression and paging.

11.10.3 Virtualization-Based Security

We covered how virtualization can be used to run virtual machines, containers, and security-isolated processes. Windows also leverages virtualization to improve its own security. The fundamental problem is that there is too much code running in kernel-mode, both as part of Windows and third-party drivers. The breadth of Windows in the world and the diversity of hardware it supports has resulted in a very healthy ecosystem of kernel-mode drivers, despite moving a lot of them into user-mode. All kernel-mode code executes at the same CPU privilege level and, therefore, any security vulnerability can enable an attacker to disrupt code flow, modify or steal security-sensitive data in the kernel. A higher privilege level is necessary to "police" kernel mode and to protect security-sensitive data.

Virtual Secure Mode provides a secure execution environment by leveraging virtualization to establish new trust boundaries for the operating system. These new trust boundaries can limit and control the set of memory, CPU and hardware resources kernel-mode software can access such that even if kernel-mode is compromised by an attacker, the entire system is not compromised.

VSM provides these trust boundaries through the concept of **VTLs** (**Virtual Trust Levels**). At its core, a VTL is a set of memory access protections. Each VTL can have a different set of protections, controlled by code running at a higher, more privileged VTL. Therefore, higher VTLs can police lower VTLs by configuring what access they have to memory. Semantically, this is similar to the relationship between user-mode and kernel-mode enforced by CPU hardware. For example, a higher VTL can use this capability in the following ways:

1. It can prevent a lower VTL from accessing certain pages which may contain security-sensitive data or data owned by the higher VTL.
2. It can prevent a lower VTL from writing to certain pages to prevent overwrite of critical settings, data structures, or code.
3. It can prevent a lower VTL from executing code pages unless they are "approved" by the higher VTL.

For each partition, including the root and guest partitions, the hypervisor supports multiple VTLs. Being in the same partition, all VTLs share the same set of virtual processors, memory, and devices, but each VTL can have different access rights to those resources. Memory protections for VTLs are implemented using a

per-VTL SLAT. The IOMMU is leveraged to enforce memory access protection for devices. As such, it is not possible for even kernel-mode code to circumvent these protections. Similar to how CPUs implement different privilege levels, each VTL has its own virtual processor state, isolated from lower VTLs. A virtual processor can transition between VTLs (similar to making a system call from user-mode into kernel-mode and back). When entering a particular VTL, the VP context is updated with the target VTL processor state and the VP is subject to that VTL's memory access protections. Higher VTLs can also prevent lower VTLs from accessing or modifying privileged CPU registers or I/O ports, which could otherwise be used to disable the hypervisor or tamper with secure devices (like fingerprint readers). Finally, each VTL has its own interrupt subsystem, such that it can enable, disable, and dispatch interrupts without interference from lower VTLs. Even though many VTLs can be supported by the hypervisor, we will focus on VTL0 and VTL1 in this chapter.

VTL0 is the VSM *normal mode* in which Windows, with its user-mode and kernel-mode components, runs. VTL1 is referred to as the *secure mode* in which a security-focused micro-OS called the Secure Kernel runs. Figure 11-54 shows this organization. The Secure Kernel provides various security services to Windows as well as **IUM** (**Isolated User Mode**) the ability to run VTL1 user-mode programs which are completely shielded from VTL0. Windows includes IUM processes, called **trustlets**, which securely manage user credentials, encryption keys, as well as biometric information for fingerprint or face authentication. The overall collection of these security mechanisms is termed VBS.

In the next section, we are going to cover the basics of Windows security and then go deeper into various security services provided by VBS.

11.11 SECURITY IN WINDOWS

NT was originally designed to meet the U.S. Department of Defense's C2 security requirements (DoD 5200.28-STD), the Orange Book, which secure DoD systems must meet. This standard requires operating systems to have certain properties in order to be classified as secure enough for certain kinds of military work. Although Windows was not specifically designed for C2 compliance, it inherits many security properties from the original security design of NT, including these:

1. Secure login with anti-spoofing measures.
2. Discretionary access controls.
3. Privileged access controls.
4. Address-space protection per process.
5. New pages must be zeroed before being mapped in.
6. Security auditing.

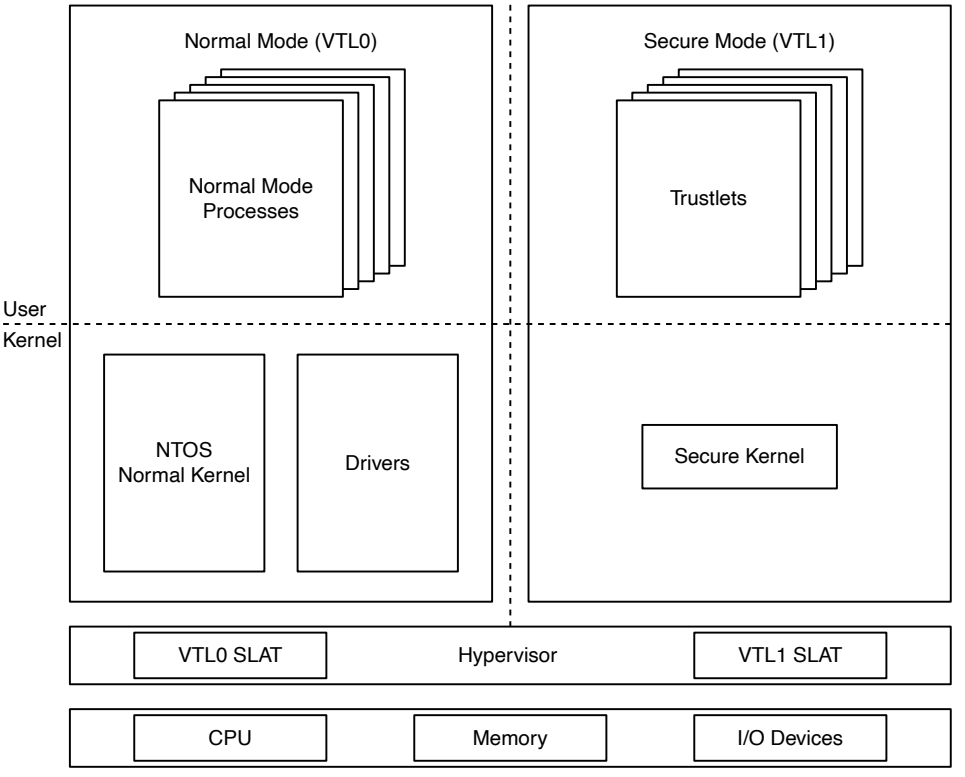

Figure 11-54. Virtual Secure Mode architecture with NT kernel in VTL0 and Secure Kernel in VTL1. VTLs share memory, CPUs, and devices, but each VTL has its own access protections for these resources, controlled by higher VTLs.

Let us review these items briefly. Secure login means that the system administrator can require all users to have a password in order to log in. Spoofing is when a malicious user writes a program that displays the login prompt or screen and then walks away from the computer in the hope that an innocent user will sit down and enter a name and password. The name and password are then written to disk and the user is told that login has failed. Windows prevents this attack by instructing users to hit CTRL-ALT-DEL to log in. This key sequence is always captured by the keyboard driver, which then invokes a system program that puts up the genuine login screen. This procedure works because there is no way for user processes to disable CTRL-ALT-DEL processing in the keyboard driver. But NT can and does disable use of the CTRL-ALT-DEL secure attention sequence in some cases, particularly for consumers and in systems that have accessibility for the disabled enabled, on phones, tablets, and the Xbox, where there is almost never a physical keyboard with a user entering commands.

In Windows 10 and newer releases, password-less authentication schemes are preferred over passwords, which are either hard-to-remember or easy-to-guess. **Windows Hello** is the umbrella name for the set of password-less authentication technologies users can use to log into Windows. Hello supports biometrics-based face, iris, and fingerprint recognition as well as per-device PIN. The data path from the infrared camera hardware to the VTL1 trustlet that implements face recognition is protected against VTL0 access via Virtualization-based Security memory and IOMMU protections. Biometric data is encrypted by the trustlet and stored on disk.

Discretionary access controls allow the owner of a file or other object to say who can use it and in what way. Privileged access controls allow the system administrator (superuser) to override them when needed. Address-space protection simply means that each process has its own protected virtual address space not accessible by any unauthorized process. The next item means that when the process heap grows, the pages mapped in are initialized to zero so that processes cannot find any old information put there by the previous owner (hence the zeroed page list in Fig. 11-37, which provides a supply of zeroed pages for this purpose). Finally, security auditing allows the administrator to produce a log of certain security-related events.

While the Orange Book does not specify what is to happen when someone steals your notebook computer, in large organizations one theft a week is not unusual. Consequently, Windows provides tools that a conscientious user can use to minimize the damage when a notebook is stolen or lost (e.g., secure login, encrypted files, etc.). In addition, organizations can use a mechanism called **Group Policy** to push down such secure machine configuration for all of its users before they can gain access to corporate network resources.

In the next section, we will describe the basic concepts behind Windows security. After that we will look at the security system calls. Finally, we will conclude by seeing how security is implemented and learn about Windows' defenses against online threats.

11.11.1 Fundamental Concepts

Every Windows user (and group) is identified by an **SID** (**Security ID**). SIDs are binary numbers with a short header followed by a long random component. Each SID is intended to be unique worldwide. When a user starts up a process, the process and its threads run under the user's SID. Most of the security system is designed to make sure that each object can be accessed only by threads with authorized SIDs.

Each process has an **access token** that specifies an SID and other properties. The token is normally created by *winlogon*, as described below. The format of the token is shown in Fig. 11-55. Processes can call GetTokenInformation to acquire this information. The header contains some administrative information. The expiration time field could tell when the token ceases to be valid, but it is currently not

used. The *Groups* field specifies the groups to which the process belongs. The default **DACL** (**Discretionary ACL**) is the access control list assigned to objects created by the process if no other ACL is specified. The user SID tells who owns the process. The restricted SIDs are to allow untrustworthy processes to take part in jobs with trustworthy processes but with less power to do damage.

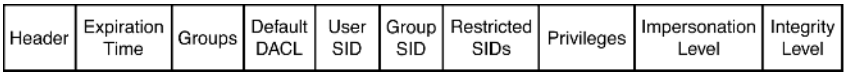

Figure 11-55. Structure of an access token.

Finally, the privileges listed, if any, give the process special powers denied ordinary users, such as the right to shut the machine down or access files to which access would otherwise be denied. In effect, the privileges split up the power of the superuser into several rights that can be assigned to processes individually. In this way, a user can be given some superuser power, but not all of it. In summary, the access token tells who owns the process and which defaults and powers are associated with it.

When a user logs in, *winlogon* gives the initial process an access token. Subsequent processes normally inherit this token on down the line. A process' access token initially applies to all the threads in the process. However, a thread can acquire a different access token during execution, in which case the thread's access token overrides the process' access token. In particular, a client thread can pass its access rights to a server thread to allow the server to access the client's protected files and other objects. This mechanism is called **impersonation**. It is implemented by the transport layers (i.e., ALPC, named pipes, and TCP/IP) and used by RPC to communicate from clients to servers. The transports use internal interfaces in the kernel's security reference monitor component to extract the security context for the current thread's access token and ship it to the server side, where it is used to construct a token which can be used by the server to impersonate the client.

Another basic concept is the **security descriptor**. Every object has a security descriptor associated with it that tells who can perform which operations on it. The security descriptors are specified when the objects are created. The NTFS file system and the registry maintain a persistent form of security descriptor, which is used to create the security descriptor for File and Key objects (the object-manager objects representing open instances of files and keys).

A security descriptor consists of a header followed by a DACL with one or more **ACEs** (**Access Control Entries**). The two main kinds of elements are Allow and Deny. An Allow element specifies an SID and a bitmap that specifies which operations processes that SID may perform on the object. A Deny element works the same way, except a match means the caller may not perform the operation. For example, Ida has a file whose security descriptor specifies that everyone has read access, Elvis has no access. Cathy has read/write access, and Ida herself has full

access. This simple example is illustrated in Fig. 11-56. The SID Everyone refers to the set of all users, but it is overridden by any explicit ACEs that follow.

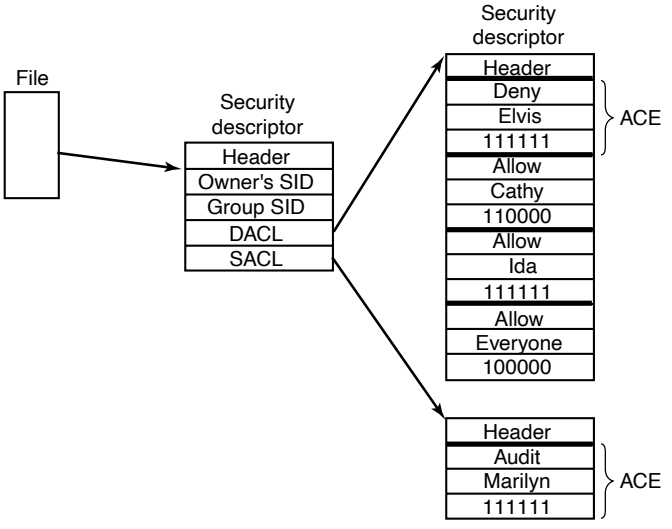

Figure 11-56. An example security descriptor for a file.

In addition to the DACL, a security descriptor also has a **SACL** (**System Access Control list**), which is like a DACL except that it specifies not who may use the object, but which operations on the object are recorded in the systemwide security event log. In Fig. 11-56, every operation that Marilyn performs on the file will be logged. The SACL also contains the **integrity level**, which we will describe shortly.

11.11.2 Security API Calls

Most of the Windows access-control mechanism is based on security descriptors. The usual pattern is that when a process creates an object, it provides a security descriptor as one of the parameters to the CreateProcess, CreateFile, or other object-creation call. This security descriptor then becomes the security descriptor attached to the object, as we saw in Fig. 11-56. If no security descriptor is provided in the object-creation call, the default security in the caller's access token (see Fig. 11-55) is used instead.

Many of the Win32 API security calls relate to the management of security descriptors, so we will focus on those here. The most important calls are listed in Fig. 11-57. To create a security descriptor, storage for it is first allocated and then initialized using InitializeSecurityDescriptor. This call fills in the header. If the owner SID is not known, it can be looked up by name using LookupAccountSid. It

can then be inserted into the security descriptor. The same holds for the group SID, if any. Normally, these will be the caller's own SID and one of the called's groups, but the system administrator can fill in any SIDs.

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

Figure 11-57. The principal Win32 API functions for security.

At this point, the security descriptor's DACL (or SACL) can be initialized with InitializeAcl. ACL entries can be added using AddAccessAllowedAce and AddAccessDeniedAce. These calls can be repeated multiple times to add as many ACE entries as are needed. DeleteAce can be used to remove an entry, that is, when modifying an existing ACL rather than when constructing a new ACL. When the ACL is ready, SetSecurityDescriptorDacl can be used to attach it to the security descriptor. Finally, when the object is created, the newly minted security descriptor can be passed as a parameter to have it attached to the object.

11.11.3 Implementation of Security

Security in a stand-alone Windows system is implemented by a number of components, most of which we have already seen (networking is a whole other story and beyond the scope of this chapter). Logging in is handled by *winlogon* and authentication is handled by *lsass*. The result of a successful login is a new GUI shell (*explorer.exe*) with its associated access token. This process uses the SECURITY and SAM hives in the registry. The former sets the general security policy and the latter contains the security information for the individual users, as discussed in Sec. 11.2.3.

Once a user is logged in, security operations happen when an object is opened for access. Every OpenXXX call requires the name of the object being opened and the set of rights needed. During processing of the open, the security reference monitor (see Fig. 11-11) checks to see if the caller has all the rights required. It performs this check by looking at the caller's access token and the DACL associated with the object. It goes down the list of ACEs in the ACL in order. As soon as it finds an entry that matches the caller's SID or one of the caller's groups, the

access found there is taken as definitive. If all the rights the caller needs are available, the open succeeds; otherwise it fails.

DACLs can have Deny entries as well as Allow entries, as we have seen. For this reason, it is usual to put entries denying access in front of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access.

After an object has been opened, a handle to it is returned to the caller. On subsequent calls, the only check that is made is whether the operation now being tried was in the set of operations requested at open time, to prevent a caller from opening a file for reading and then trying to write on it. Additionally, calls on handles may result in entries in the audit logs, as required by the SACL.

Windows added another security facility to deal with common problems securing the system by ACLs. There are new mandatory **integrity-level SIDs** in the process token, and objects specify an integrity-level ACE in the SACL. The integrity level prevents write-access to objects no matter what ACEs are in the DACL. There are five major integrity levels: untrusted, low, medium, high, and system. In particular, the integrity-level scheme is used to protect against a web browser process that has been compromised by an attacker (perhaps by the user ill-advisedly downloading code from an unknown Website). In addition to using severely restricted tokens, browser sandboxes run with an integrity level of *low* or *untrusted*. By default all files and registry keys in the system have an integrity level of *medium*, so browsers running with lower integrity levels cannot modify them.

Even though highly security-conscious applications like browsers make use of system mechanisms to follow the principle of least privilege, there are many popular applications that do not. In addition, there is the chronic problem in Windows where most users run as administrators. The design of Windows does not require users to run as administrators, but many common operations unnecessarily required administrator rights and most user accounts ended up getting created as administrators. This also led to many programs acquiring the habit of storing data in global registry and file system locations to which only administrators have write access. This neglect over many releases made it just about impossible to use Windows successfully if you were not an administrator. Being an administrator all the time is dangerous. Not only can user errors easily damage the system, but if the user is somehow fooled or attacked and runs code that is trying to compromise the system, the code will have administrative access, and can bury itself deep in the system.

In order to deal with this problem, Windows introduced **UAC** (**User Account Control**). With UAC, even administrator users run with standard user rights. If an attempt is made to perform an operation requiring administrator access, the system overlays a special desktop and takes control so that only input from the user can authorize the access (similarly to how CTRL-ALT-DEL works for C2 security). This is called an **elevation**. Under the covers, UAC creates two tokens for the user session during administrator user logon: one is a regular administrator token and the other is a restricted token for the same user, but with the administrator rights

stripped. Applications launched by the user are assigned the standard token, but when elevation is necessary and is approved, the process switches to the actual administrator token.

Of course, without becoming administrator it is possible for an attacker to destroy what the user really cares about, namely his personal files. But UAC does help foil existing types of attacks, and it is always easier to recover a compromised system if the attacker was unable to modify any of the system data or files.

Another important security feature in Windows is support for **protected processes**. As we mentioned earlier, protected processes provide a stronger security boundary from user-mode attacks, including from administrators. Normally, the user (as represented by a token object) defines the privilege boundary in the system. When a process is created, the user has access to the process through any number of kernel facilities for process creation, debugging, path names, thread injection, and so on. Protected processes are shut off from user-mode access. User-mode callers cannot read or write its virtual memory, cannot inject code or threads into its address space. The original use of this was to allow digital rights management software to better protect content. Later, protected processes were expanded to more user-friendly purposes, like securing the system against attackers rather than securing content against attacks by the system owner. While protected processes are able to thwart straightforward attacks, defending a process against administrator users is very difficult without hardware-based isolation. Administrators can easily load drivers into kernel-mode and access any VTL0 process. As such, protected processes should be viewed as a layer of defense, but not more.

As mentioned above, the *lsass* process handles user authentication and therefore needs to maintain various secrets associated with credentials like password hashes and Kerberos tickets in its address space. As such, it runs as a protected process to guard against user-mode attacks, but malicious kernel-mode code can easily leak these secrets. **Credential Guard** is a VBS feature introduced in Windows 10 which protects these secrets in an IUM trustlet called *LsaIso.exe*. *lsass* communicates with *LsaIso* to perform authentication such that credential secrets are never exposed to VTL0 and even kernel-mode malware cannot steal them.

Microsoft's efforts to improve the security of Windows have been accelerating since early 2000s as more and more attacks have been launched against systems around the world. Attackers range from casual hackers to paid professionals to very sophisticated nation states with virtually unlimited resources engaging in cyber warfare. Some of these attacks have been very successful, taking entire countries and major corporations offline, and incurring costs of billions of dollars.

11.11.4 Security Mitigations

It would be great for users if computer software (and hardware) did not have any bugs, particularly bugs that are exploitable by hackers to take control of their computer and steal their information, or use their computer for illegal purposes

such as distributed denial-of-service attacks, compromising other computers, and distribution of spam or other illicit materials. Unfortunately, this is not *yet* feasible in practice, and computers continue to have security vulnerabilities. The industry continues to make progress towards producing more secure systems code with better developer training, more rigorous security reviews and improved source code annotations (e.g., *SAL*) with associated static analysis tools. On the validation front, intelligent **fuzzers** automatically stress-test interfaces with random inputs to cover all code paths and **address sanitizers** inject checks for invalid memory accesses to find bugs. More and more systems code is moving to languages like *Rust* with strong memory safety guarantees. On the hardware front, research and development of new CPU features like Intel's **CET** (**Control-flow Enforcement Technology**) ARM's **MTE** (**Memory Tagging Extensions**) and the emerging CHERI architecture help eliminate classes of vulnerabilities as we will describe below.

As long as humans continue to build software, it is going to have bugs, many of which lead to security vulnerabilities. Microsoft has been following a multi-pronged approach pretty successfully since Windows Vista to mitigate these vulnerabilities such that they are difficult and costly to leverage by attackers. The components of this strategy are listed below.

1. Eliminate classes of vulnerabilities.
2. Break exploitation techniques.
3. Contain damage and prevent persistence of exploits.
4. Limit the window of time to exploit vulnerabilities.

Let us study each of these components in more detail.

Eliminating Vulnerabilities

Most code vulnerabilities stem from small coding errors that lead to buffer overruns, using memory after it is freed, type-confusion due to incorrect casts and using uninitialized memory. These vulnerabilities allow an attacker to disrupt code flow by overwriting return addresses, virtual function pointers, and other data that control the execution or behavior of programs. Indeed, memory safety issues have consistently accounted for about 70% of exploitable bugs in Windows.

Many of these problems can be avoided if type-safe languages such as C# and Rust are used instead of C and C++. Fortunately, a lot of new development is shifting to these languages. And even with these unsafe languages many vulnerabilities can be avoided if students and professional developers are better trained to understand the pitfalls of parameter and data validation, and the many dangers inherent in memory allocation APIs. After all, many of the software engineers who write code at Microsoft today were students only a few years earlier, just as many of you

reading this case study are now. Many books are available on the kinds of small coding errors that are exploitable in pointer-based languages and how to avoid them (e.g., Howard and LeBlank, 2009).

Compiler-based techniques can also make C/C++ code safer. Windows 11 build system leverages a mitigation called *InitAll* which zero-initializes stack variables and simple types to eliminate vulnerabilities due to uninitialized variables.

There's also significant investment in hardware advances to help eliminate memory safety vulnerabilities. One of them is the ARMv8.5 Memory Tagging Extensions. This associates a 4-bit memory *tag*, stored elsewhere in RAM with each 16-byte granule of memory. Pointers also have a tag field (in reserved address bits) which is set, for example, by memory allocators. When memory is accessed via the pointer, the CPU compares its tag with the tag stored in memory and raises an exception if a mismatch occurs. This approach eliminates bugs like buffer overruns because the memory beyond the buffer will have a different tag. Windows does not currently support MTE. CHERI is a more comprehensive approach that uses 128-bit unforgeable *capabilities* to access memory, providing very fine-grained access control. It is a promising approach with durable safety guarantees, but it has a much higher implementation cost compared to extensions like MTE because it requires porting and recompiling all software.

Breaking Exploitation Techniques

The security landscape is constantly changing. The broad availability of the Internet made it much easier for attackers to exploit vulnerabilities at a much larger scale, causing significant damage. At the same time, digital transformation is moving more and more enterprise processes into software, thus creating new targets for attackers. As software defenses improve, attackers continually adapt and invent new types of exploits. It's a cat-and-mouse game, but exploits are certainly getting harder to build and deploy.

In the early 2000s, life was much easier for attackers. It was possible to exploit stack buffer overruns to copy code to the stack, overwrite the function return address to start executing the code when the function returns. Over multiple releases, several OS mitigations almost completely wiped out this attack vector. The first one was **/GS** (**Guarded Stack**), released in Windows XP Service Pack 2. /GS is a randomized stack canary implementation where a function entry point saves a known value, called a *security cookie* on its stack and verifies, before returning, that the cookie has not been overwritten. Since the security cookie is generated randomly at process creation time and combined with the stack frame address, it is not easy to guess. So, /GS provides good protection against linear buffer overflows, but does not detect the overflows until the end of the function and does not detect out-of-band writes to the return address if canary is not corrupted.

Another important security mitigation included in Windows XP Service Pack 2 was **DEP** (**Data Execution Prevention**). DEP leverages processor support for the

No-eXecute (NX) protection in page table entries to properly mark non-code portions of the address space, such as thread stacks and heap data, as non-executable. As a result, the practice of exploiting stack or heap buffer overruns and copying code to the stack or heap for execution was no longer possible. In response, attackers started resorting to **ROP** (**Return-Oriented Programming**) which involves overwriting the function return address or function pointers to point them at executable code fragments (typically OS DLLs) already loaded in the address space. Such code fragments ending with ' return instruction, called **gadgets**, can be strung together by overwriting the stack with pointers to the desired fragments to run. It turns out there are enough usable gadgets in most address spaces to construct any program; and tools exist to find them. Also, in a given release, OS DLLs were loaded at consistent addresses, so ROP attacks were easy to put together once the gadgets were identified.

With Windows Vista, however, ROP attacks became much more difficult to mount because of **ASLR** (**Address Space Layout Randomizations**), a feature where the layout of code and data in the user-mode address space is randomized. Even though ASLR was not enabled for every single binary initially—allowing attackers to use non-ASLR'd binaries for ROP attacks—Windows 8 enabled ASLR for all binaries. Windows 10 also brought ASLR to kernel-mode. Addresses of all kernel-mode code, pools, and critical data structures like the PFN database and page tables are all randomized. It should be noted that ASLR is far more effective in a 64-bit address space since there are a lot more addresses to choose from vs. 32-bit, making attacks like heap spraying to overwrite virtual function pointers impractical.

With these mitigations in place, attackers must find and exploit an *arbitrary read/write* vulnerability discover locations of DLLs, heaps, or stacks. Then, they need to corrupt function pointers or return addresses to gain control via ROP. Even if an attacker has defeated ASLR and can read or write anything in the victim address space, Windows has additional mitigations to prevent the attacker from gaining arbitrary code execution. There are two aspects of these mitigations, preventing control-flow hijacking and preventing arbitrary code generation.

In order to hijack control flow, most exploits corrupt a function pointer (typically a C++ virtual function table) to redirect it to a ROP gadget. **CFG** (**Control Flow Guard**) is a mitigation that enforces coarse-grained control-flow integrity for indirect calls (such as virtual method calls) to prevent such attacks. It relies on metadata placed in code binaries which describe the set of code locations that can be called indirectly. During module load, this information is encoded by the kernel into a process-global bitmap, called the *CFG bitmap*, covering every binary in the address space. The CFG bitmap is protected to be read-only in user-mode. Each indirect call site performs a *CFG check* to verify that the target address is indeed marked as indirectly callable in the global bitmap. If not, the process is terminated. Since the vast majority of functions in a binary are not intended to be called indirectly, CFG significantly cuts down the options available to an attacker when

corrupting function pointers. In particular, function pointers can only point to the first instruction of a function, aligned at 16-bytes, rather than arbitrary ROP gadgets.

With Windows 10, it became possible to enable CFG in kernel-mode with **KCFG** (**Kernel CFG**) even though it was only enabled with Virtualization-based security. Unsurprisingly, KCFG leverages VSM to enable the Secure Kernel to maintain the CFG bitmap and prevent anybody in VTL0 kernel-mode from modifying it. With Windows 11, Kernel CFG is enabled by default on all machines.

One weakness in CFG is that every indirect call target is treated the same; a function pointer can call any indirectly callable function regardless of the number or types of parameters. An improved version of CFG, called **XFG** (**Extended Flow Guard**) was developed to address this shortcoming. Instead of a global bitmap, XFG relies on *function signature hashes* to ensure that a call site is compatible with the target of a function pointer. Each indirectly-callable function is preceded by a hash covering its complete type, including the number and types of its parameters. Each call site knows the signature hash of the function it is intending to call and validates whether the target of the function pointer is a match. As a result, XFG is much more selective than CFG in its validation and does not leave too many options to attackers. Even though the initial release of Windows 11 does not include XFG, it is present in Windows Insider flights and is likely to ship in a subsequent official release.

CFG and XFG only protect the *forward edge* of code flow by validating indirect calls. However, as we described earlier, many attacks corrupt stack return addresses to hijack code flow when the victim function returns. Reliably defending against return address hijacking using a software-only mechanism turns out to be very difficult. In fact, Microsoft internally implemented such a defense in 2017, called **RFG** (**Return Flow Guard**). RFG used a software **shadow stack** into which return addresses on the call stack were saved on function entry and validated by the function epilogue. Even though an incredible amount of engineering effort went into this project across the compiler, operating systems, and security teams, the project was ultimately shelved because an internal security researcher identified an attack with a high success rate that corrupted the return address on the stack before it was copied to the shadow stack. Such an attack was previously considered, but thought to be infeasible due to its low expected success rate. RFG also relied on the shadow stack being hidden from software running in the process (otherwise an attacker could just corrupt the shadow stack as well). Soon after RFG was canceled, other security researchers identified reliable ways to locate such frequently accessed data structures in the address space. These were some very important takeaways from the project: security features that rely on hiding things and probabilistic mechanisms do not tend to be durable.

A robust defense against return address hijacking had to wait until Intel's CET was released in late 2020. CET is a hardware implementation of shadow stacks without any (known) race conditions and does not depend on keeping the shadow

stacks hidden. When CET is enabled, the function call instruction pushes the return address to both the call stack and the shadow stack and the subsequent return compares them. The shadow stack is identified to the processor by PTE entries and is not writable by regular store instructions. Windows 10 implemented support for CET in user-mode and Windows 11 extended protection to kernel-mode with **KCET** (**Kernel-mode CET**). Similar to KCFG, KCET relies on the Secure Kernel to protect and maintain the shadow stack for each thread.

An alternative approach to defending against return address hijacking is ARM's **PAC** (**Pointer Authentication**) mechanism. Instead of maintaining a shadow stack, PAC cryptographically signs return addresses on the stack and verifies the signature before returning. The same mechanism can be used to protect other function pointers to implement forward-edge code-flow integrity (which is handled through CFG on Windows). In general, PAC is considered to be a weaker protection than CET because it relies on the secrecy of the keys used for signing and authentication, but it also may be subject to substitution attacks when the same stack location is reused for a different call. Regardless, PAC is much stronger than having no protection, so Windows 11 is built with PAC instructions and supports PAC in user-mode. In its documentation, Microsoft refers to these return address protection mechanisms generically as **HSP** (**Hardware-enforced Stack Protection**).

So far, we described how Windows protects forward and backward control-flow integrity using CFG and HSP. Defending against arbitrary code execution also requires that the code itself is protected. Attackers should not be able to overwrite existing code and they should not be able to load unauthorized code or generate new code in the address space. In fact, careful readers may have noticed that the protection offered by CFG/KCFG, CET/KCET, or PAC can trivially be defeated if the relevant instructions are simply overwritten by the attacker.

CIG (**Code Integrity Guard**) is the Windows 10 security feature which allows a process to require that all code binaries loaded into the process be signed by a recognized entity, thus preventing arbitrary, attacker-controlled code from loading into the process. In kernel-mode, 64-bit Windows has always required drivers to be properly signed. The remaining attack vectors are closed off with **ACG** (**Arbitrary Code Guard**) which enforces two restrictions:

1. *Code is immutable*: Also known as *W^X*, it ensures that Writable and Executable page protections cannot both be enabled on a page.
2. *Data cannot become code*: Executable pages can only be *born*; page protections cannot be changed to enable execution later.

The kernel memory manager enforces CIG and ACG on processes that opt-in. Since many applications rely on code injection into other processes, CIG and ACG cannot be enabled globally due to compatibility concerns, but sensitive processes that do not do this (like browsers) do enable them.

In kernel-mode, ACG guarantees are provided by **HVCI** (**Hypervisor-enforced Code Integrity**), which is a Virtualization-based Security component and lives in the Secure Kernel. It leverages SLAT protections to enforce W^X and code signing requirements for VTL1 kernel mode and for code that loads into IUM trustlets. Windows 11 enables HVCI by default. When VBS is not enabled, a kernel component called **PatchGuard** is responsible for enforcing code integrity. With no VBS and therefore no SLAT protection, it is not possible to deterministically prevent attacks on code. PatchGuard relies on capturing hashes of pristine code pages and verifying the hash at random times in the future. As such, it does not prevent code modification, but will typically detect it over time, unless the attacker is able to restore things back to their original state in time. In order to evade detection and tamper, PatchGuard keeps itself hidden and its data structures obfuscated.

Attackers who possess an arbitrary read/write primitive do not always attack code, or code-flow; they can also attack various data structures to gain execution or change system behavior. For that reason, PatchGuard also verifies the integrity of numerous kernel data structures, global variables, function pointers, and sensitive processor registers which can be used to take control of the system. With VBS enabled, **HyperGuard**, the VTL1 counterpart to PatchGuard is responsible for maintaining the integrity of kernel data structures. Many of these data structures can be protected deterministically via SLAT protections and secure intercepts that can be configured to fire when VTL0 modifies sensitive processor registers. And KCFG protects function pointers. Still, maintaining the integrity of writable data structures like the list of processes or object type descriptors cannot easily be done with SLAT protections, so even when HyperGuard is enabled, PatchGuard is still active, albeit in a reduced functionality mode. Figure 11-58 summarizes the security facilities we have discussed.

Containing Damage

Despite all efforts to prevent exploits, it is possible (and likely) that malicious intrusions will happen sooner or later. In the security world, it is not wise to rely on a single layer of security. Damage containment mechanisms in Windows provide additional **defense-in-depth** against attacks that are able to work around existing mitigations. These are all the sandboxing mechanisms we have already covered in this chapter:

1. AppContainers (Sec. 11.2.1)
2. Browser Sandbox (Sec. 11.11.3)
3. Microsoft Defender Application Guard (Sec. 11.10.2)
4. Windows Sandbox (Sec. 11.10.2)
5. IUM Trustlets (Sec. 11.10.3)

Mitigation	VBS-only	Description
InitAll	No	Zero-initializes stack variables to avoid vulnerabilities
/GS	No	Add canary to stack frames to protect return addresses
DEP	No	Data Execution Prevention. Stacks and heaps are not executable
ASLR/KASLR	No	Randomize user/kernel address space to make ROP attacks difficult
CFG	No	Control Flow Guard. Protect integrity of forward-edge control flow
KCFG	Yes	Kernel-mode CFG. Secure Kernel maintains CFG bitmap
XFG	No	Extended Flow Guard. Much finer grained protection than CFG
CET	No	Strong defense against ROP attacks using shadow stacks
KCET	Yes	Kernel-mode CET. Secure Kernel maintains shadow stacks.
PAC	No	Protects stack return addresses using signatures
CIG	No	Enforces that code binaries are properly signed
ACG	No	User-mode enforcement for W^X and that data cannot become code
HVCI	Yes	Kernel-mode enforcement for W^X and that data cannot become code
PatchGuard	No	Detect attempts to modify kernel code and data
HyperGuard	Yes	Stronger protection than PatchGuard
Windows Defender	No	Built-in antimalware software

Figure 11-58. Some of the principal security protections in Windows.

Limiting Window of Time to Exploit

The most direct way to limit exploitation of a security bug is to fix or mitigate the issue and to deploy broadly as quickly as possible. **Windows Update** is an automated service providing fixes to security vulnerabilities by patching the affected programs and libraries within Windows. Many of the vulnerabilities fixed were reported by security researchers, and their contributions are acknowledged in the notes attached to each fix. Ironically the security updates themselves pose a significant risk. Many vulnerabilities used by attackers are exploited only after a fix has been published by Microsoft. This is because reverse engineering the fixes themselves is the primary way most hackers discover vulnerabilities in systems. Systems that did not have all known updates immediately applied are thus susceptible to attack. The security research community is usually insistent that companies patch all vulnerabilities found within a reasonable time. The current monthly patch frequency used by Microsoft is a compromise between keeping the community happy and how often users must deal with patching to keep their systems safe.

A significant cause of delay in fixing security issues is the need for a reboot after the updated binaries are deployed to customer machines. Reboots are very inconvenient when many applications are open and the user is in the middle of work. The situation is similar on server machines where any downtime may result in Websites, file servers, database becoming inaccessible. In cloud datacenters, host

OS downtime results in all hosted virtual machines becoming unavailable. In summary, there's never a good time to reboot machines to install security updates.

As a result, many customer machines remain vulnerable to attacks for multiple days even though the fix is sitting on their disk. Windows Update does its best to nudge the user to reboot, but it needs to walk a fine line between securing the machine and upsetting the user by forcing a reboot.

Hotpatching is a reboot-less update technology that can eliminate these difficult trade-offs. Instead of replacing binaries on disk with updated ones, hotpatching deploys a **patch binary**, loads it into memory at runtime, and dynamically redirects code flow from the **base binary** to the patch binary based on metadata embedded in the patch binary. Instead of replacing entire binaries, hotpatching works at the individual function level and redirects only select functions to their updated versions in the patch binary. These are called **forward patches**. Unmodified functions always run in the base binary such that they can be patched later, if necessary. As a result, if an updated function in the patch binary calls an unmodified function, the unmodified function needs to be **back-patched** to the base binary. In addition, if patch functions need to access global variables, such accesses need to be redirected to the base binary's globals through an indirection.

Patch binaries are regular portable executable (PE) images that include patch metadata. Patch metadata identifies the base image to which the patch applies and lists the image-relative addresses of the functions to patch, including forward and backward patches. Due to the differences in instruction sets, patch application differs slightly between x64 and arm64, but code flow remains the same. In both cases, an **HPAT** (**Hotpatch Address Table**) is allocated right after every binary (including patch binaries). Each HPAT entry is populated with the necessary code to redirect execution to the target. So, the act of applying a forward or backward patch to a function amounts to overwriting the first instruction of the function to make it jump to its corresponding HPAT entry. On x64, this requires 6 bytes of padding to be present before every function, but arm64 does not have that requirement.

Figure 11-59 illustrates code and data flow in a hotpatch with an example where functions foo( ) and baz( ) are updated in *mylib_patch.dll*. When applying this patch, the patch engine is going to populate the HPAT for *mylib.dll* with redirection code targeting foo( ) and baz( ) in the patch binary, labeled as foo' and baz'. Also, since foo( ) calls bar( ) and bar( ) was not updated, the patch engine is going to populate the HPAT for the patch binary to redirect bar( ) back to its implementation in the base binary. Finally, since foo( ) references a global variable, the code emitted by the compiler for foo( ) in the patch binary will indirectly access the global through a pointer. So, the patch engine will also update that pointer to refer to the global variable in the base binary.

Hotpatching is supported for user-mode, kernel-mode, VTL1 kernel-mode and even the hypervisor. User-mode hotpatches are applied by NTOS, VTL0 kernel-mode hotpatches are applied by the Secure Kernel (which is also able to patch

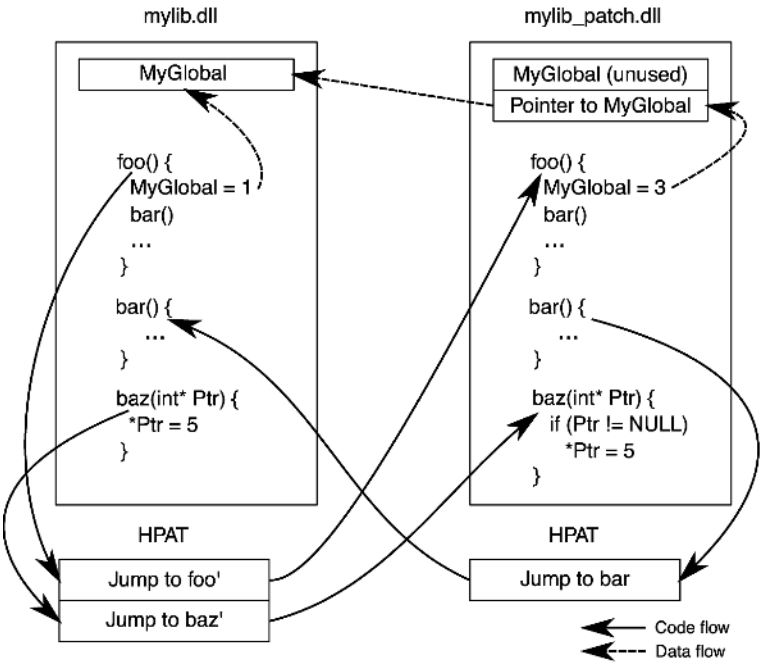

Figure 11-59. Hotpatch application for *mylib.dll*. Functions foo( ) and baz( ) are updated in the patch binary, *mylib_patch.dll*.

itself) and the hypervisor patches itself. As such, VBS is a prerequisite for hotpatching. NTOS is responsible for validating proper signing for user-mode hotpatches and SK validates all other types of hotpatches such that a malicious actor cannot just hotpatch the kernel.

Hotpatching is vital for the Azure fleet and has been in use since mid-2010s. Every month, millions of machines in datacenters are hotpatched with various fixes and feature updates, with zero downtime for customer virtual machines. Hotpatching is also supported on the Azure Edition of Windows Server 2019 and 2022. These operating systems can be configured to receive cumulative hotpatch packages from Windows Update for multiple months followed by a reboot-required, non-hotpatch update. A regular reboot-required update is necessary every few months because it is not always possible to fix every issue with a hotpatch.

Antimalware

In addition to all the security mechanisms we described in this section, another layer of defense is antimalware software which has become a critical tool for combating malicious code. Antimalware can detect and quarantine malicious code even

before it gets to attack. Windows includes a full-featured antimalware package called **Windows Defender**. This type of software hooks into kernel operations to detect malware inside files, as well as recognize the behavioral patterns that are used by specific instances (or general categories) of malware. These behaviors include the techniques used to survive reboots, modify the registry to alter system behavior, and launching particular processes and services needed to implement an attack. Windows Defender provides a good protection against common malware and similar software packages are also available from third-party providers.

11.12 SUMMARY

Kernel mode in Windows is structured in the HAL, the kernel and executive layers of NTOS, and a large number of device drivers implementing everything from device services to file systems and networking to graphics. The HAL hides certain differences in hardware from the other components. The kernel layer manages the CPUs to support multithreading and synchronization, and the executive implements most kernel-mode services.

The executive is based on kernel-mode objects that represent the key executive data structures, including processes, threads, memory sections, drivers, devices, and synchronization objects—to mention a few. User processes create objects by calling system services and get back handle references which can be used in subsequent system calls to the executive components. The operating system also creates objects internally. The object manager maintains a namespace into which objects can be inserted for subsequent lookup.

The most important objects in Windows are processes, threads, and sections. Processes have virtual address spaces and are containers for resources. Threads are the unit of execution and are scheduled by the kernel layer using a priority algorithm in which the highest-priority ready thread always runs, preempting lower-priority threads as necessary. Sections represent memory objects, like files, that can be mapped into the address spaces of processes. EXE and DLL program images are represented as sections, as is shared memory.

Windows supports demand-paged virtual memory. The paging algorithm is based on the working-set concept. The system maintains several types of page lists, to optimize the use of memory. The various page lists are fed by trimming the working sets using complex formulas that try to reuse physical pages that have not been referenced in a long time. The cache manager manages virtual addresses in the kernel that can be used to map files into memory, dramatically improving I/O performance for many applications because read operations can be satisfied without accessing the disk.

I/O is performed by device drivers, which follow the Windows Driver Model. Each driver starts out by initializing a driver object that contains the addresses of the procedures that the system can call to manipulate devices. The actual devices

are represented by device objects, which are created from the configuration description of the system or by the plug-and-play manager as it discovers devices when enumerating the system buses. Devices are stacked and I/O request packets are passed down the stack and serviced by the drivers for each device in the device stack. I/O is inherently asynchronous, and drivers commonly queue requests for further work and return back to their caller. File-system volumes are implemented as devices in the I/O system.

The NTFS file system is based on a master file table, which has one record per file or directory. All the metadata in an NTFS file system is itself part of an NTFS file. Each file has multiple attributes, which can be either in the MFT record or nonresident (stored in blocks outside the MFT). NTFS supports Unicode, compression, journaling, and encryption among many other features.

Finally, Windows has a sophisticated security system based on access control lists and integrity levels. Each process has an authentication token that tells the identity of the user and what special privileges the process has, if any. Each object has a security descriptor associated with it. The security descriptor points to a discretionary access control list that contains access control entries that can allow or deny access to individuals or groups. Windows has added numerous security features in recent releases, including BitLocker for encrypting entire volumes, and address-space randomization, nonexecutable stacks, and other measures to make successful attacks more difficult.

PROBLEMS

1. Give one advantage and one disadvantage of the registry vs. having individual *.ini* files.
2. A mouse can have one, two, or three buttons. All three types are in use. Does the HAL hide this difference from the rest of the operating system? Why or why not?
3. The HAL keeps track of time starting in the year 1601. Give an example of an application where this feature is useful.
4. In Sec. 11.3.3, we described the problems caused by multithreaded applications closing handles in one thread while still using them in another. One possibility for fixing this would be to insert a sequence field. How could this help? What changes to the system would be required?
5. Many components of the executive (Fig. 11-11) call other components of the executive. Give three examples of one component calling another one, but use (six) different components in all.
6. How would you design a mechanism to achieve **BNO** (**BaseNamedObjects**) isolation for non-UWP applications?
7. An alternative to using DLLs is to statically link each program with precisely those library procedures it actually calls, no more and no less. If this scheme were to be introduced, what would be the benefits and drawbacks?

8. Why is \?? directory specially handled in the object manager rather than dealing with it in the Win32 layer in *kernelbase.dll* like BNO?

9. Windows uses 2-MB large pages because it improves the effectiveness of the TLB, which can have a profound impact on performance. Why is this? Why are 2-MB large pages not used all the time?

10. Is there any limit on the number of different operations that can be defined on an executive object? If so, where does this limit come from? If not, why not?

11. The Win32 API call WaitForMultipleObjects allows a thread to block on a set of synchronization objects whose handles are passed as parameters. As soon as any one of them is signaled, the calling thread is released. Is it possible to have the set of synchronization objects include two semaphores, one mutex, and one critical section? Why or why not? (*Hint*: This is not a trick question but it does require some careful thought.)

12. When initializing a global variable in a multithreaded program, a common programming error is to allow a race condition where the variable can be initialized twice. Why could this be a problem? Windows provides the InitOnceExecuteOnce API to prevent such races. How might it be implemented?

13. Why is it a bad idea to allow recursive lock acquisition even for shared acquires?

14. How would you implement a bounded buffer using an SRW lock and a condition variable? The operations to implement are Add( ) and Remove( ) where Add( ) adds an item to the buffer, blocking if space is not available. Remove( ) removes an item, waiting until one is available.

15. Name three reasons why a desktop process might be terminated. What additional reason might cause a process running a modern application to be terminated?

16. Modern applications must save their state to disk every time the user switches away from the application. This seems inefficient, as users may switch back to an application many times and the application simply resumes running. Why does the operating system require applications to save their state so often rather than just giving them a chance at the point the application is actually going to be terminated?

17. As described in Sec. 11.4, there is a special handle table used to allocate IDs for processes and threads. The algorithms for handle tables normally allocate the first available handle (maintaining the free list in LIFO order). In recent releases of Windows, this was changed so that the ID table always keeps the free list in FIFO order. What is the problem that the LIFO ordering potentially causes for allocating process IDs, and why does not UNIX have this problem?

18. Suppose that the quantum is set to 20 msec and the current thread, at priority 24, has just started a quantum. Suddenly an I/O operation completes and a priority 28 thread is made ready. About how long does it have to wait to get to run on the CPU?

19. In Windows, the current priority is always greater than or equal to the base priority. Are there any circumstances in which it would make sense to have the current priority be lower than the base priority? If so, give an example. If not, why not?

20. Windows uses a facility called Autoboost to temporarily raise the priority of a thread that holds the resource that is required by a higher-priority thread. How do you think this works?

21. In Windows, it is easy to implement a facility where threads running in the kernel can temporarily attach to the address space of a different process. Why is this so much harder to implement in user mode? Why might it be interesting to do so?

22. Name two ways to give better response time to the threads in important processes.

23. Even when there is plenty of free memory available, and the memory manager does not need to trim working sets, the paging system can still often be writing to disk. Why?

24. Windows swaps the processes for modern applications rather than reducing their working set and paging them. Why would this be more efficient? (*Hint*: It makes much less of a difference when the disk is an SSD.)

25. Why does the self-map used to access the physical pages of the page directory and page tables for a process always occupy the same 512 GB of kernel virtual addresses (with 4-level page tables mapping 48-bit address space on the x64)?

26. On x64, with 4-level page tables, what would be the virtual address of the self-map entry if the self-map entry were at index 0x155 instead of 0x1ED?

27. If a region of virtual address space is reserved but not committed, do you think a VAD is created for it? Defend your answer.

28. Which of the transitions shown in Fig. 11-37 are policy decisions, as opposed to required moves forced by system events (e.g., a process exiting and freeing its pages)?

29. Suppose that a page is shared and in two working sets at once. If it is evicted from one of the working sets, where does it go in Fig. 11-37? What happens when it is evicted from the second working set?

30. What are the other ways workloads can interfere with one another on a machine even if we run them on different processor cores, use memory partitions and use different disks (or use disk io rate controls)?

31. What are some other potential benefits of an infrastructure like memory compression beyond what has been mentioned in this chapter so far? What are some possibilities?

32. Suppose that a dispatcher object representing some type of exclusive lock (like a mutex) is marked to use a notification event instead of a synchronization event to announce that the lock has been released. Why would this be bad? How much would the answer depend on lock hold times, the length of quantum, and whether the system was a multiprocessor?

33. To support POSIX, the native NtCreateProcess API supports duplicating a process in order to support fork. In UNIX, fork is usually followed by an exec. One example where this was used historically was in the Berkeley dump program which would back-up disks to magnetic tape. Fork was used as a way of checkpointing the dump program so it could be restarted if there was an error with the tape device. Give an example of how Windows might do something similar using NtCreateProcess. (*Hint*: Consider processes that host DLLs to implement functionality provided by a third party.)

34. A file has the following mapping. Give the MFT run entries.

Offset	0	1	2	3	4	5	6	7	8	9	10
Disk address	50	51	52	22	24	25	26	53	54	-	60

35. Consider the MFT record of Fig. 11-46. Suppose that the file grew and a 10th block was assigned to the end of the file. The number of this block is 66. What would the MFT record look like now?

36. In Fig. 11-49(b), the first two runs are each of length 8 blocks. Is it just an accident that they are equal, or does this have to do with the way compression works? Explain your answer.

37. Suppose that you wanted to build Windows Lite. Which of the fields of Fig. 11-55 could be removed without weakening the security of the system?

38. The mitigation strategy for improving security despite the continuing presence of vulnerabilities has been very successful. Modern attacks are very sophisticated, often requiring the presence of multiple vulnerabilities to build a reliable exploit. One of the vulnerabilities that is usually required is an *information leak*. Explain how an information leak can be used to defeat address-space randomization in order to launch an attack based on return-oriented programming.

39. An extension model used by many programs (Web browsers, Office, COM servers) involves *hosting* DLLs to hook and extend their underlying functionality. Is this a reasonable model for an RPC-based service to use as long as it is careful to impersonate clients before loading the DLL? Why not?

40. When running on a NUMA machine, whenever the Windows memory manager needs to allocate a physical page to handle a page fault it attempts to use a page from the NUMA node for the current thread's ideal processor. Why? What if the thread is currently running on a different processor?

41. Give a couple of examples where an application might be able to recover easily from a backup based on a volume shadow copy rather the state of the disk after a system crash.

42. In Sec. 11.10, providing new memory to the process heap was mentioned as one of the scenarios that require a supply of zeroed pages in order to satisfy security requirements. Give one or more other examples of virtual memory operations that require zeroed pages.

43. Windows contains a hypervisor which allows multiple operating systems to run simultaneously. This is available on clients, but is far more important in cloud computing. When a security update is applied to a guest operating system, it is not much different than patching a server. However, when a security update is applied to the root operating system, this can be a big problem for the users of cloud computing. What is the nature of the problem? What can be done about it?

44. Section 11.10 describes three different approaches for scheduling logical processors for VMs. One of these is known as the root scheduler, which uses the host threads to back a virtual processor in the VM. This scheduling scheme takes into account the priority of the thread running on the virtual processor as a hint to what the host thread

priority should be. What advantages does this have and why is the remote thread priority just a hint?

45. Figure 11-53 illustrates how the file system namespace exposed to a Windows Server Container is backed by a number of host directories. Why do you suppose things were implemented this way? What advantages does this have? Are there disadvantages?

46. Windows 10 introduced a feature known as Microsoft Defender Application Guard that allows the Edge browser and Microsoft Office apps to run a hardware isolated container, and remotes the UI back to the host. The result is that the application appears to the user to be running locally even though its actually hosted in a type of VM. What subtle user experience issues could this cause?

47. What are some examples of code changes that may not be hotpatchable or difficult to hotpatch? What can be done to make more changes hotpatchable?

48. Does hotpatching break CFG guarantees by introducing new indirect jumps?

49. The *regedit* command can be used to export part or all of the registry to a text file under all current versions of Windows. Save the registry several times during a work session and see what changes. If you have access to a Windows computer on which you can install software or hardware, find out what changes when a program or device is added or removed.

50. Write a UNIX program that simulates writing an NTFS file with multiple streams. It should accept a list of one or more files as arguments and write an output file that contains one stream with the attributes of all arguments and additional streams with the contents of each of the arguments. Now write a second program for reporting on the attributes and streams and extracting all the components.