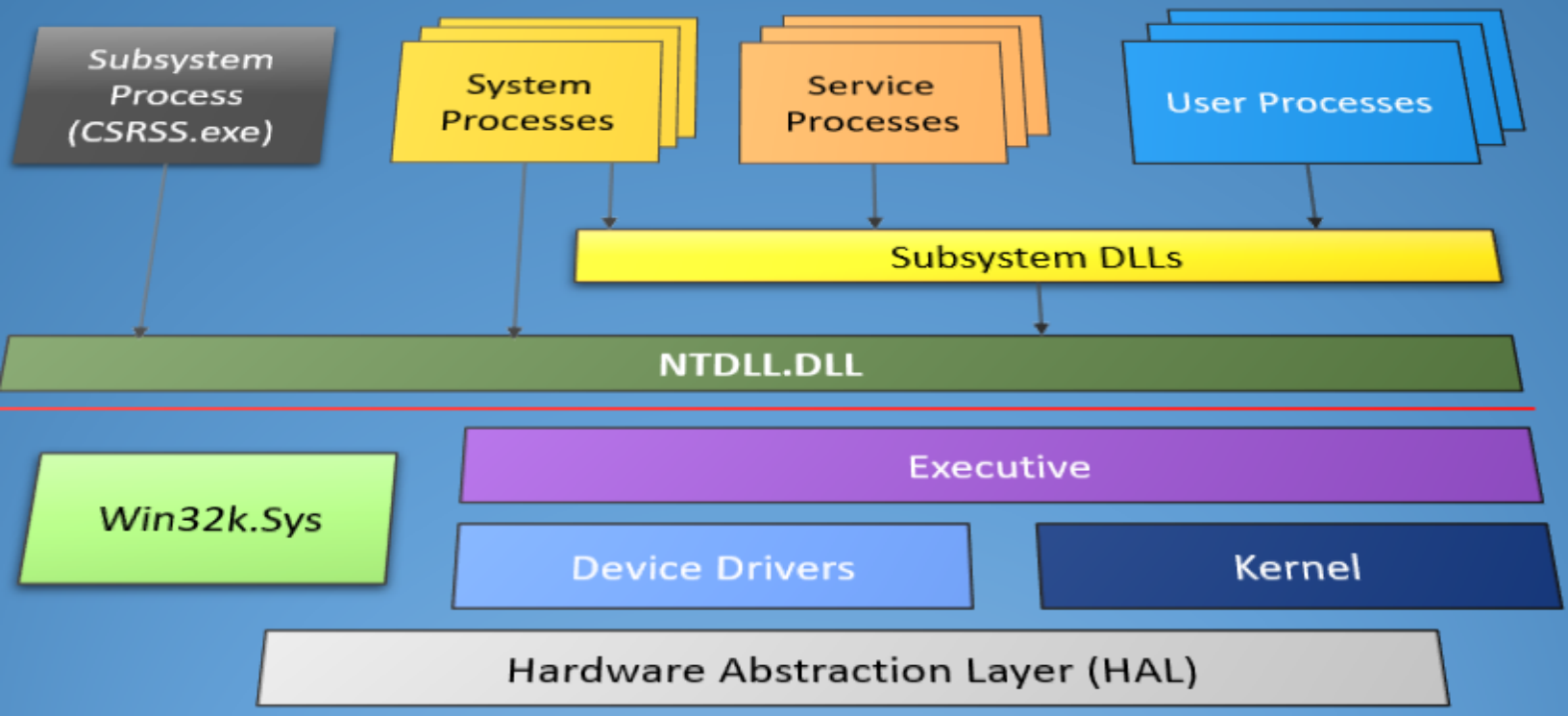
Windows Kernel Programming

Pavel Yosifovich



Windows Kernel Programming

Pavel Yosifovich

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

This version was published on 2020-12-10

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Chapter 1: Windows Internals Overview

This chapters describes the most important concepts in the internal workings of Windows. Some of the topics will be described in greater detail later in the book, where it's closely related to the topic at hand. Make sure you understand the concepts in this chapter, as these make the foundations upon any driver and even user mode low-level code, is built.

In this chapter:

- **Processes**
- **Virtual Memory**
- **Threads**
- **System Services**
- **System Architecture**
- **Handles and Objects**

Processes

A process is a containment and management object that represents a running instance of a program. The term "process runs" which is used fairly often, is inaccurate. Processes don't run – processes manage. Threads are the ones that execute code and technically run. From a high-level perspective, a process owns the following:

- An executable program, which contains the initial code and data used to execute code within the process.
- A private virtual address space, used for allocating memory for whatever purposes the code within the process needs it.
- A primary token, which is an object that stores the default security context of the process, used by threads executing code within the process (unless a thread assumes a different token by using impersonation).
- A private handle table to executive objects, such as events, semaphores and files.

- One or more threads of execution. A normal user mode process is created with one thread (executing the classic main/WinMain function). A user mode process without threads is mostly useless and under normal circumstances will be destroyed by the kernel.

These elements of a process are depicted in figure 1-1.

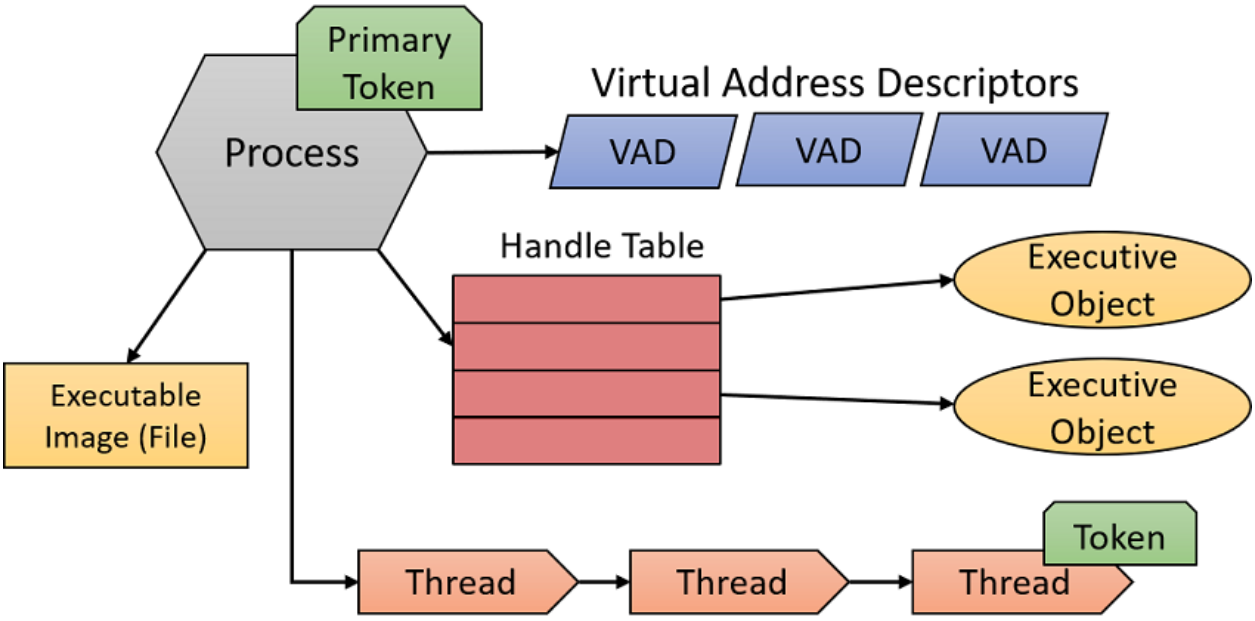


Figure 1-1: Important ingredients of a process

A process is uniquely identified by its Process ID, which remains unique as long as the kernel process object exists. Once it's destroyed, the same ID may be reused for new processes. It's important to realize that the executable file itself is not a unique identifier of a process. For example, there may be five instances of `notepad.exe` running at the same time. Each process has its own address space, its own threads, its own handle table, its own unique process ID, etc. All those five processes are using the same image file (`notepad.exe`) as their initial code and data. Figure 1-2 shows a screen shot of Task Manager's Details tab showing five instances of `Notepad.exe`, each with its own attributes.

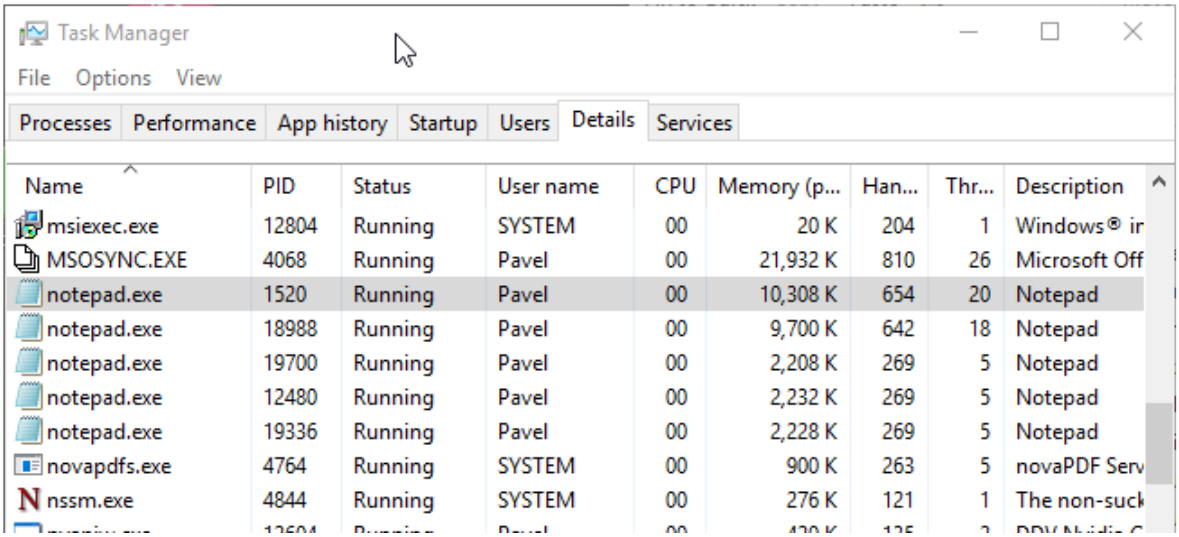


Figure 1-2: Five instances of notepad

Virtual Memory

Every process has its own virtual, private, linear address space. This address space starts out empty (or close to empty, since the executable image and `NtDll.Dll` are the first to be mapped, followed by more subsystem DLLs). Once execution of the main (first) thread begins, memory is likely to be allocated, more DLLs loaded, etc. This address space is private, which means other processes cannot access it directly. The address space range starts at zero (although technically the first 64KB of address cannot be allocated or used in any way), and goes all the way to a maximum which depends on the process "bitness" (32 or 64 bit) and the operating system "bitness" as follows:

- For 32-bit processes on 32-bit Windows systems, the process address space size is 2 GB by default.
- For 32-bit processes on 32-bit Windows systems that use the increase user address space setting, that process address space size can be as large as 3 GB (depending on the exact setting). To get the extended address space range, the executable from which the process was created must have been marked with the `LARGEADDRESSAWARE` linker flag in its header. If it was not, it would still be limited to 2 GB.
- For 64-bit processes (on a 64-bit Windows system, naturally), the address space size is 8 TB (Windows 8 and earlier) or 128 TB (Windows 8.1 and later).
- For 32-bit processes on a 64-bit Windows system, the address space size is 4 GB if the executable image is linked with the `LARGEADDRESSAWARE` flag. Otherwise, the size remains at 2 GB.

The requirement of the `LARGEADDRESSAWARE` flag stems from the fact that a 2 GB address range requires 31 bits only, leaving the most significant bit (MSB) free for application use. Specifying this flag indicates that the program is not using bit 31 for anything and so setting that bit to 1 (which would happen for addresses larger than 2 GB) is not an issue.

Each process has its own address space, which makes any process address relative, rather than absolute. For example, when trying to determine what lies in address 0x20000, the address itself is not enough; the process to which this address relates to must be specified.

The memory itself is called *virtual*, which means there is an indirect relationship between an address range and the exact location where it's found in physical memory (RAM). A buffer within a process may be mapped to physical memory, or it may temporarily reside in a file (such as a page file). The term virtual refers to the fact that from an execution perspective, there is no need to know if the memory about to be accessed is in RAM or not; if the memory is indeed mapped to RAM, the CPU will access the data directly. If not, the CPU will raise a page fault exception that will cause the memory manager's page fault handler to fetch the data from the appropriate file, copy it to RAM, make the required changes in the page table entries that map the buffer, and instruct the CPU to try again. Figure 1-3 shows this mapping from virtual to physical memory for two processes.

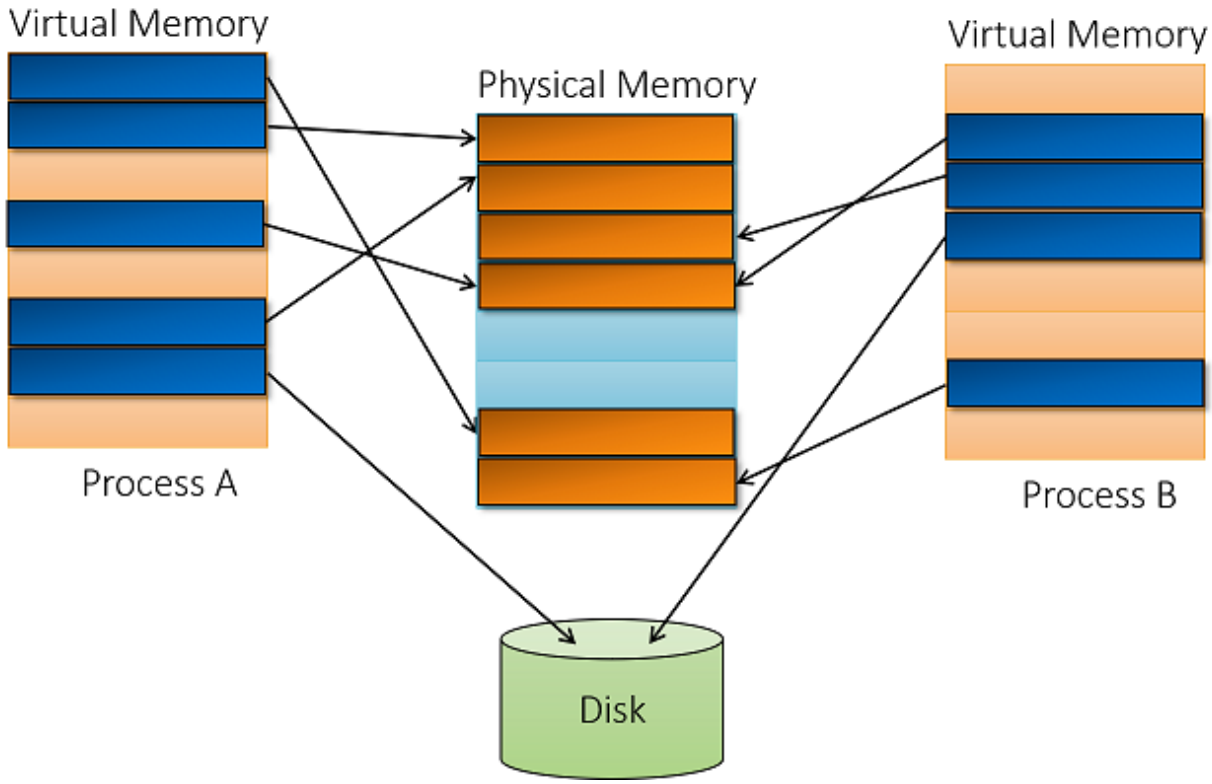


Figure 1-3: virtual memory mapping

The unit of memory management is called a *page*. Every attribute related to memory is always at a page's granularity, such as its protection. The size of a page is determined by CPU type (and on some processors, may be configurable), and in any case the memory manager must follow suit. Normal (sometimes called small) page size is 4 KB on all Windows supported architectures.

Apart from the normal (small) page size, Windows also supports large pages. The size of a large page is 2 MB (x86/x64/ARM64) and 4 MB (ARM). This is based using the Page Directory Entry (PDE) to map the large page without using a page table. This results in quicker translation, but most importantly better use the Translation Lookaside Buffer (TLB) – a cache of recently translated pages maintained by the CPU. In the case of a large page, a single TLB entry is able to map significantly more memory than a small page. The downside of large pages is the need to have the memory contiguous in RAM, which can fail if memory is tight or very fragmented. Also, large pages are always non-pageable and must be protected with read/write access only. Huge pages of 1 GB in size are supported on Windows 10 and Server 2016 and later. These are used automatically with large pages if an allocation is at least 1 GB in size and that page can be located as contiguous in RAM.

Page States

Each page in virtual memory can be in one of three states:

- Free – the page is not allocated in any way; there is nothing there. Any attempt to access that page would cause an access violation exception. Most pages in a newly created process are free.
- Committed – the reverse of free; an allocated page that can be accessed successfully sans protection attributes (for example, writing to a read only page causes an access violation). Committed pages are usually mapped to RAM or to a file (such as a page file).
- Reserved – the page is not committed, but the address range is reserved for possible future commitment. From the CPU's perspective, it's the same as Free – any access attempt raises an access violation exception. However, new allocation attempts using the `VirtualAlloc` function (or `NtAllocateVirtualMemory`, the related native API) that does not specify a specific address would not allocate in the reserved region. A classic example of using reserved memory to maintain contiguous virtual address space while conserving memory is described later in this chapter in the section "Thread Stacks".

System Memory

The lower part of the address space is for processes' use. While a certain thread is executing, its parent process address space is visible from address zero to the upper limit as described in the previous section. The operating system, however, must also reside somewhere – and that somewhere is the upper address range that's supported on the system, as follows:

- On 32-bit systems running without the increase user virtual address space setting, the operating system resides in the upper 2 GB of virtual address space, from address `0x8000000` to `0xFFFFFFFF`.
- On 32-bit systems configured with the increase user virtual address space setting, the operating system resides in the address space left. For example, if the system is configured with 3 GB user address space per process (the maximum), the OS takes the upper 1 GB (from address `0xC0000000` to `0xFFFFFFFF`). The entity that suffers mostly from this address space reduction is the file system cache.
- On 64-bit systems on Windows 8, Server 2012 and earlier, the OS takes the upper 8 TB of virtual address space.
- On 64-bit systems on Windows 8.1, Server 2012 R2 and later, the OS takes the upper 128 TB of virtual address space.

System space is not process-relative – after all, it's the same "system", the same kernel, the same drivers that service every process on the system (the exception is some system memory that is on a per-session basis but is not important for this discussion). It follows that any address in system space is absolute rather than relative, since it "looks" the same from every process context. Of course, actual access from user mode into system space results in an access violation exception.

System space is where the kernel itself, the Hardware Abstraction Layer (HAL) and kernel drivers reside once loaded. Thus, kernel drivers are automatically protected from direct user mode access. It also means they have a potentially system-wide impact. For example, if a kernel driver leaks memory, that memory will not be freed even after the driver unloads. User mode processes, on the other hand, can never leak anything beyond their life time. The kernel is responsible for closing and freeing everything private to a dead process (all handles are closed and all private memory is freed).

Threads

The actual entities that execute code are threads. A Thread is contained within a process, using the resources exposed by the process to do work (such as virtual memory and handles to kernel objects). The most important information a thread owns is the following:

- Current access mode, either user or kernel.
- Execution context, including processor registers and execution state.
- One or two stacks, used for local variable allocations and call management.
- Thread Local Storage (TLS) array, which provides a way to store thread-private data with uniform access semantics.
- Base priority and a current (dynamic) priority.
- Processor affinity, indicating on which processors the thread is allowed to run on.

The most common states a thread can be in are:

- Running – currently executing code on a (logical) processor.
- Ready – waiting to be scheduled for execution because all relevant processors are busy or unavailable.
- Waiting – waiting for some event to occur before proceeding. Once the event occurs, the thread goes to the Ready state.

Figure 1-4 shows the state diagram for these states. The numbers in parenthesis indicate the state numbers, as can be viewed by tools such as Performance Monitor. Note that the Ready state has a sibling state called Deferred Ready, which is similar, and really exists to minimize some internal locking.

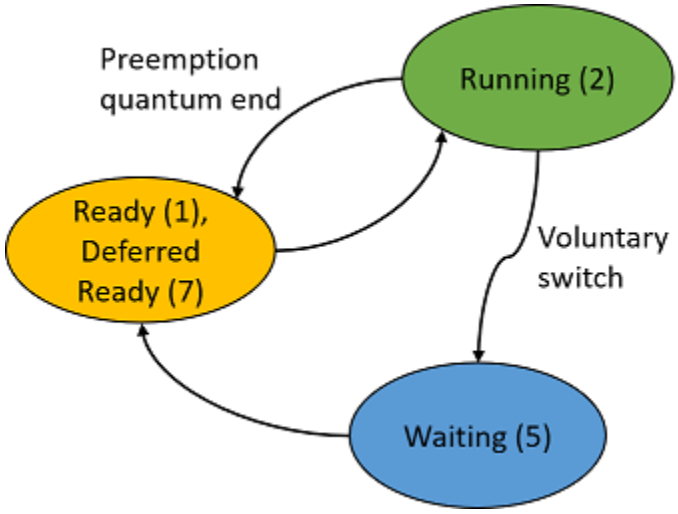


Figure 1-4: Common thread states

Thread Stacks

Each thread has a stack it uses while executing, used for local variables, parameter passing to functions (in some cases) and where return addresses are stored prior to function calls. A thread has at least one stack residing in system (kernel) space, and it's pretty small (default is 12 KB on 32-bit systems and 24 KB on 64-bit systems). A user mode thread has a second stack in its process user space address range and is considerably larger (by default can grow to 1 MB). An example with three user mode threads and their stacks is shown in figure 1-5. In the figure, threads 1 and 2 are in process A and thread 3 is in process B.

The kernel stack always resides in RAM while the thread is in the Running or Ready states. The reason for this is subtle and will be discussed later in this chapter. The user mode stack, on the other hand, may be paged out, just like any user mode memory.

The user mode stack is handled differently than the kernel mode stack, in terms of its size. It starts out with a small amount of memory committed (could be as small as a single page), with the rest of the stack address space as reserved memory, meaning it's not allocated in any way. The idea is to be able to grow the stack in case the thread's code needs to use more stack space. To make this work, the next page (sometimes more than one) right after the committed part is marked with a special protection called `PAGE_GUARD` – this is a guard page. If the thread needs more stack space it would write to the guard page which would throw an exception that is handled by the memory manager. The memory manager then removes the guard protection and commits the page and marks the next page as a guard page. This way, the stack grows as needed and the entire stack memory is not committed upfront. Figure 1-6 shows the way a user mode's thread stack looks like.

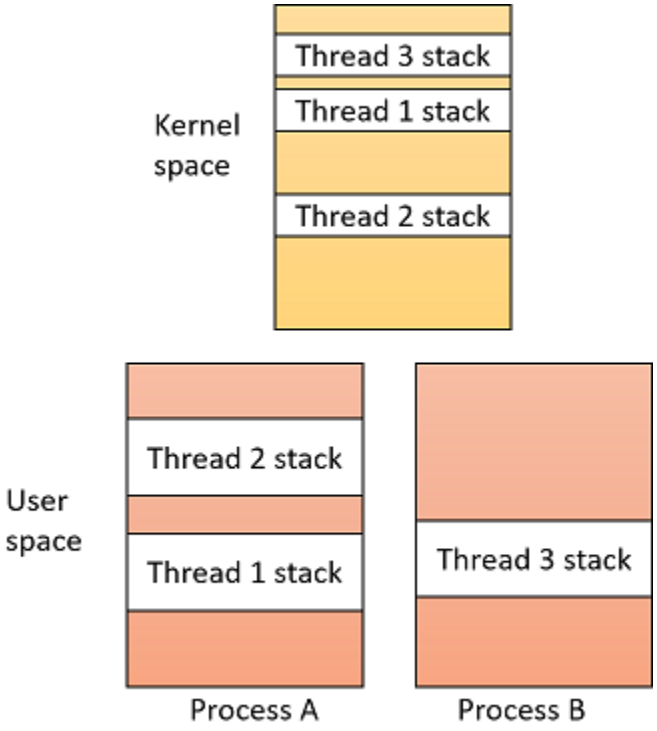


Figure 1-5: User mode threads and their stacks

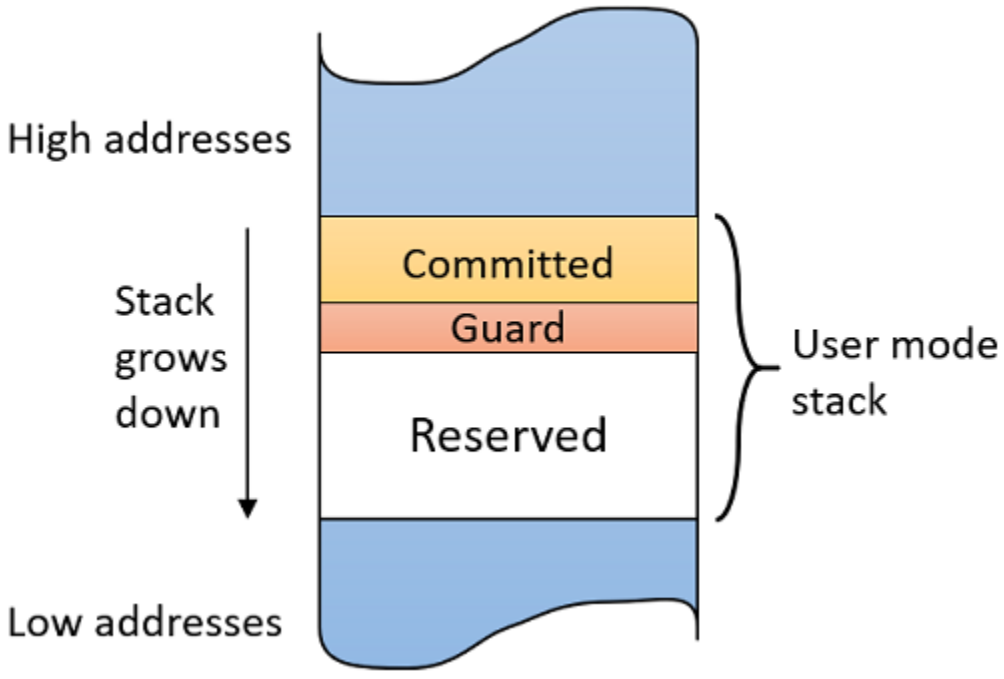


Figure 1-6: Thread's stack in user space

The sizes of a thread's user mode stack are determined as follows:

- The executable image has a stack commit and reserved values in its Portable Executable (PE) header. These are taken as defaults if a thread does not specify alternative values.
- When a thread is created with `CreateThread` (and similar functions), the caller can specify its requires stack size, either the upfront committed size or the reserved size (but not both), depending on a flag provided to the function; specifying zero goes with the default according the above bullet.

Curiously enough, the function `CreateThread` and `CreateRemoteThread(Ex)` only allow specifying a single value for the stack size and can be the committed or the reserved size, but not both. The native (undocumented) function, `NtCreateThreadEx` allows specifying both values.

System Services (a.k.a. System Calls)

Applications need to perform various operations that are not purely computational, such as allocating memory, opening files, creating threads, etc. These operations can only be ultimately performed by code running in kernel mode. So how would user-mode code be able to perform such operations? Let's take a classic example: a user running a Notepad process uses the File menu to request opening a file. Notepad's code responds by calling the `CreateFile` documented Windows API function. `CreateFile` is documented as implemented in `kernel32.Dll`, one of the Windows subsystem DLLs. This function still runs in user mode, so there is no way it can directly open a file. After some error checking, it calls `NtCreateFile`, a function implemented in `NTDLL.dll`, a foundational DLL that implements the API known as the "Native API", and is in fact the lowest layer of code which is still in user mode. This (officially undocumented) API is the one that makes the transition to kernel mode. Before the actual transition, it puts a number, called system service number, into a CPU register (`EAX` on Intel/AMD architectures). Then it issues a special CPU instruction (`syscall` on x64 or `sysenter` on x86) that makes the actual transition to kernel mode while jumping to a predefined routine called the system service dispatcher.

The system service dispatcher, in turn, uses the value in that `EAX` register as an index into a System Service Dispatch Table (SSDT). Using this table, the code jumps to the system service (system call) itself. For our Notepad example, the SSDT entry would point to the I/O manager's `NtCreateFile` function. Notice the function has the same name as the one in `NTDLL.dll` and in fact has the same arguments as well. Once the system service is complete, the thread returns to user mode to execute the instruction following sysenter/syscall. This sequence of events is depicted in figure 1-7.

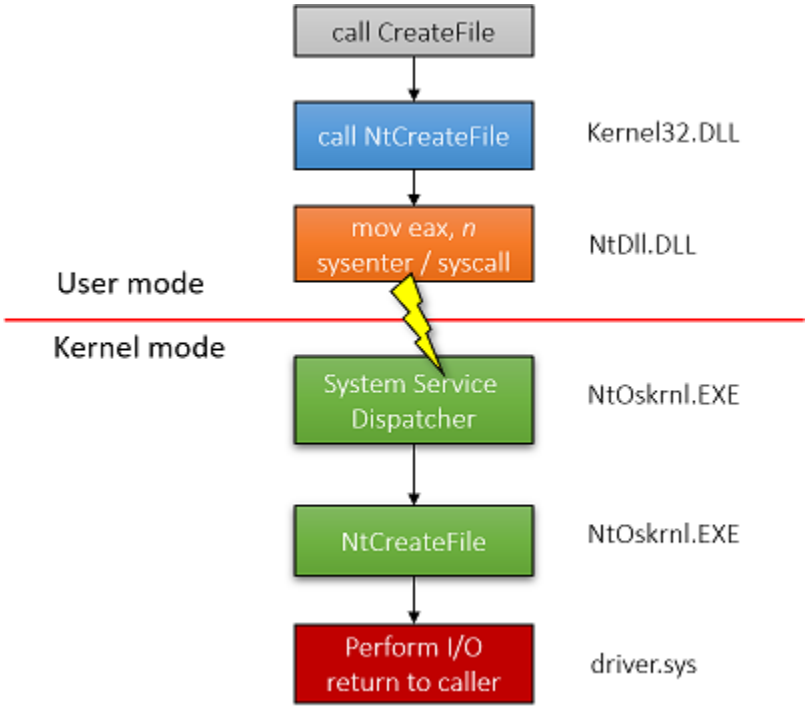


Figure 1-7: System service function call flow

General System Architecture

Figure 1-8 shows the general architecture of Windows, comprising of user mode and kernel mode components.

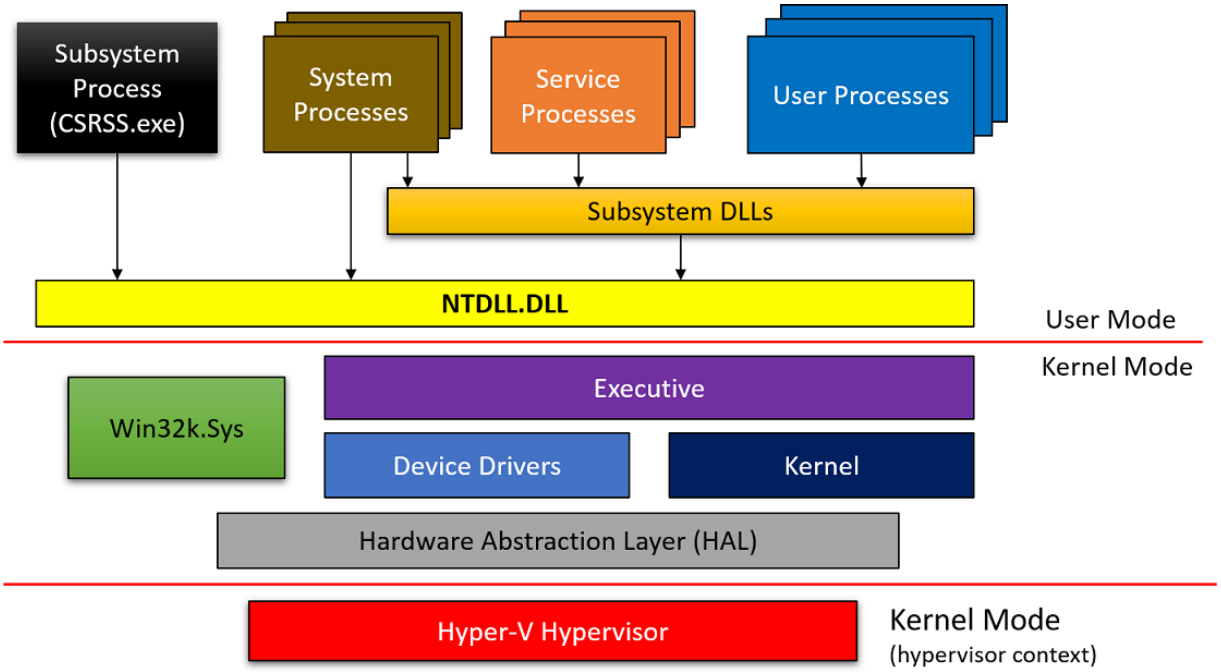


Figure 1-8: Windows system architecture

Here's a quick rundown of the named boxes appearing in figure 1-8:

- **User processes**

  These are normal processes based on image files, executing on the system, such as instances of Notepad.exe, cmd.exe, explorer.exe and so on.
- **Subsystem DLLs**

  Subsystem DLLs are dynamic link libraries (DLLs) that implement the API of a subsystem. A subsystem is a certain view of the capabilities exposed by the kernel. Technically, starting from Windows 8.1, there is only a single subsystem – the Windows Subsystem. The subsystem DLLs include well-known files, such as kernel32.dll, user32.dll, gdi32.dll, advapi32.dll, combase.dll and many others. These include mostly the officially documented API of Windows.
- **NTDLL.DLL**

  A system-wide DLL, implementing the Windows native API. This is the lowest layer of code which is still in user mode. Its most important role is to make the transition to kernel mode for system call invocation. NTDLL also implements the Heap Manager, the Image Loader and some part of the user mode thread pool.
- **Service Processes**

  Service processes are normal Windows processes that communicate with the Service Control Manager (SCM, implemented in services.exe) and allow some control over their lifetime. The SCM can start, stop, pause, resume and send other messages to services. Services typically execute under one of the special Windows accounts – local system, network service or local service.

- **Executive**

  The Executive is the upper layer of NtOskrnl.exe (the "kernel"). It hosts most of the code that is in kernel mode. It includes mostly the various "managers": Object Manager, Memory Manager, I/O Manager, Plug & Play Manager, Power Manager, Configuration Manager, etc. It's by far larger than the lower Kernel layer.
- **Kernel**

  The Kernel layer implements the most fundamental and time sensitive parts of kernel mode OS code. This includes thread scheduling, interrupt and exception dispatching and implementation of various kernel primitives such as mutex and semaphore. Some of the kernel code is written in CPU-specific machine language for efficiency and for getting direct access to CPU-specific details.
- **Device Drivers**

  Device drivers are loadable kernel modules. Their code executes in kernel mode and so has the full power of the kernel. This book is dedicated to writing certain types of kernel drivers.
- **Win32k.sys**

  The "kernel mode component of the Windows subsystem". Essentially this is a kernel module (driver) that handles the user interface part of Windows and the classic Graphics Device Interface (GDI) APIs. This means that all windowing operations (CreateWindowEx, GetMessage, PostMessage, etc.) is handles by this component. The rest of the system has little-to-none knowledge of UI.
- **Hardware Abstraction Layer (HAL)**

  The HAL is an abstraction later over the hardware closest to the CPU. It allows device drivers to use APIs that do not require detailed and specific knowledge of things like Interrupt Controller or DMA controller. Naturally, this layer is mostly useful for device drivers written to handle hardware devices.
- **System Processes**

  System processes is an umbrella term used to describe processes that are typically "just there", doing their thing where normally these processes are not communicated with directly. They are important nonetheless, and some in fact, critical to the system's well-being. Terminating some of them is fatal and causes a system crash. Some of the system processes are native processes, meaning they use the native API only (the API implemented by NTDLL). Example system processes include `Smss.exe`, `Lsass.exe`, `Winlogon.exe`, `Services.exe` and others.
- **Subsystem Process**

  The Windows subsystem process, running the image `Csrss.exe`, can be viewed as a helper to the kernel for managing processes running under the Windows system. It is a critical process, meaning if killed, the system would crash. There is normally one `Csrss.exe` instance per session, so on a standard system two instances would exist – one for session `0` and one for the logged-on user session (typically `1`). Although `Csrss.exe` is the "manager" of the Windows subsystem (the only one left these days), its importance goes beyond just this role.
- **Hyper-V Hypervisor**

The Hyper-V hypervisor exists on Windows 10 and server 2016 systems if they support Virtualization Based Security (VBS). VBS provides an extra layer of security, where the actual machine is in fact a virtual machine controlled by Hyper-V. VBS is beyond the scope of this book. For more information, check out the *Windows Internals* book.

Windows 10 version 1607 introduced the Windows Subsystem for Linux (WSL). Although this may look like yet another subsystem, like the old POSIX and OS/2 subsystems supported by Windows, it is not quite like that at all. The old subsystems were able to execute POSIX and OS/2 apps if theses were compiled on a Windows compiler. WSL, on the other hand, has no such requirement. Existing executables from Linux (stored in ELF format) can be run as-is on Windows, without any recompilation.

To make something like this work, a new process type was created – the Pico process together with a Pico provider. Briefly, a Pico process is an empty address space (minimal process) that is used for WSL processes, where every system call (Linux system call) must be intercepted and translated to the Windows system call(s) equivalent using that Pico provider (a device driver). There is a true Linux (the user-mode part) installed on the Windows machine.

End of sample