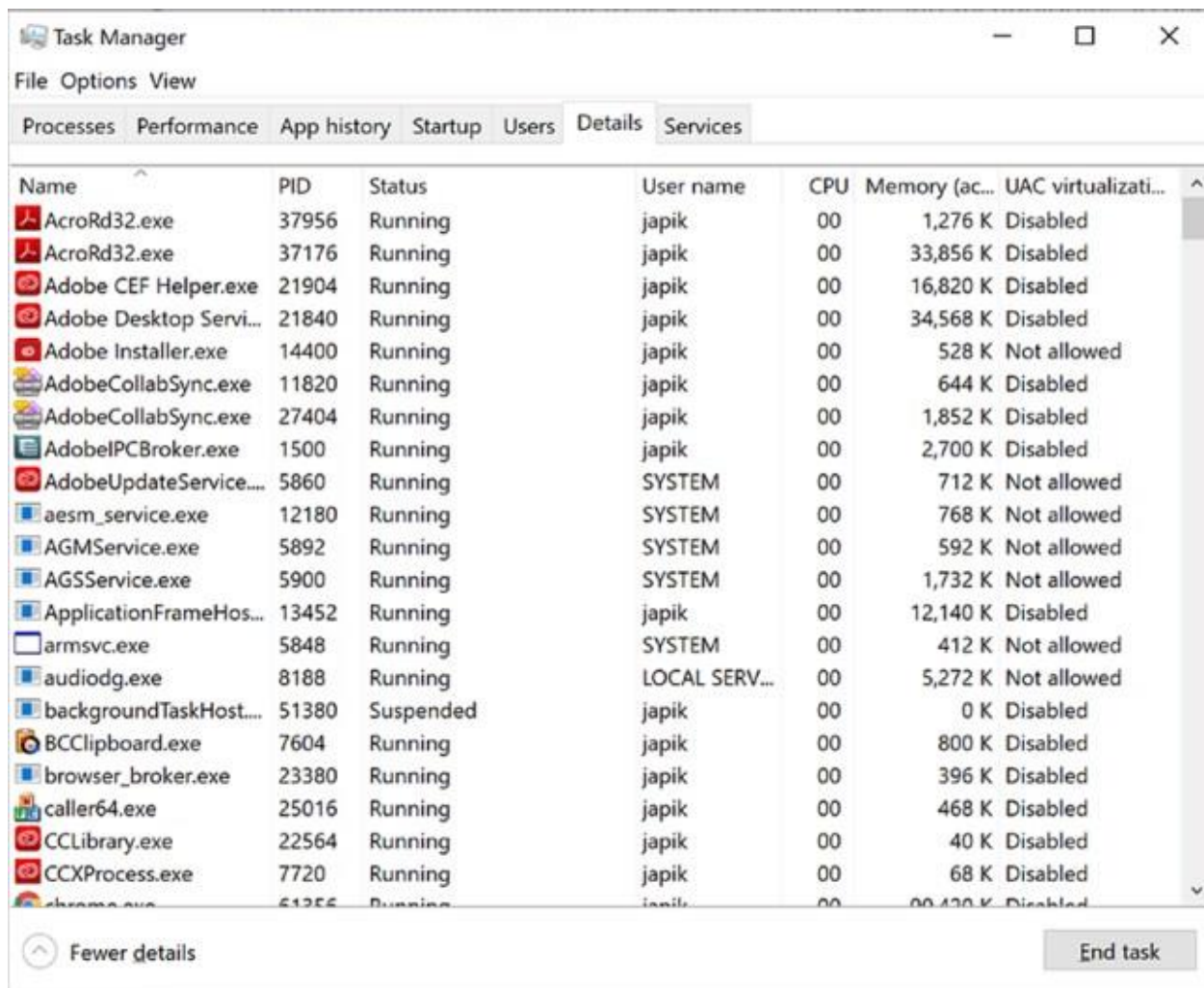


The concept of a “process” existed within Windows-based operating systems well before the release of the .NET/.NET Core platforms. In simple terms, a process is a running program. However, formally speaking, a process is an operating system–level concept used to describe a set of resources (such as external code libraries and the primary thread) and the necessary memory allocations used by a running application. For each .NET Core application loaded into memory, the OS creates a separate and isolated process for use during its lifetime.

Every Windows process is assigned a unique process identifier (PID) and may be independently loaded and unloaded by the OS as necessary (as well as programmatically). As you might be aware, the Processes tab of the Windows Task Manager utility (activated via the Ctrl+Shift+Esc keystroke combination on Windows) allows you to view various statistics regarding the processes running on a given machine. The Details tab allows you to view the assigned PID and image name (see image).



The screenshot shows the Windows Task Manager application with the 'Details' tab selected. The table lists various running processes, including Adobe applications, system services, and background tasks. Each row displays the process name, PID, status, user name, CPU usage, memory usage, and UAC virtualization status.

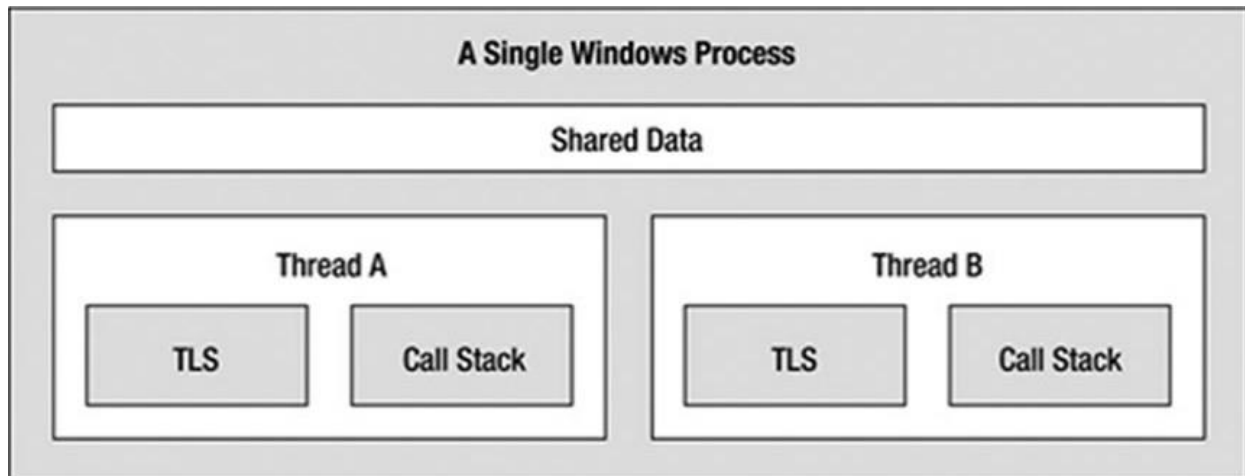
| Name | PID | Status | User name | CPU | Memory (ac... | UAC virtualizati... |
|------------------------|-------|-----------|---------------|-----|---------------|---------------------|
| AcroRd32.exe | 37956 | Running | japik | 00 | 1,276 K | Disabled |
| AcroRd32.exe | 37176 | Running | japik | 00 | 33,856 K | Disabled |
| Adobe CEF Helper.exe | 21904 | Running | japik | 00 | 16,820 K | Disabled |
| Adobe Desktop Servi... | 21840 | Running | japik | 00 | 34,568 K | Disabled |
| Adobe Installer.exe | 14400 | Running | japik | 00 | 528 K | Not allowed |
| AdobeCollabSync.exe | 11820 | Running | japik | 00 | 644 K | Disabled |
| AdobeCollabSync.exe | 27404 | Running | japik | 00 | 1,852 K | Disabled |
| AdobePCBroker.exe | 1500 | Running | japik | 00 | 2,700 K | Disabled |
| AdobeUpdateService.... | 5860 | Running | SYSTEM | 00 | 712 K | Not allowed |
| aesm_service.exe | 12180 | Running | SYSTEM | 00 | 768 K | Not allowed |
| AGMSERVICE.exe | 5892 | Running | SYSTEM | 00 | 592 K | Not allowed |
| AGSSERVICE.exe | 5900 | Running | SYSTEM | 00 | 1,732 K | Not allowed |
| ApplicationFrameHos... | 13452 | Running | japik | 00 | 12,140 K | Disabled |
| armsvc.exe | 5848 | Running | SYSTEM | 00 | 412 K | Not allowed |
| audiodg.exe | 8188 | Running | LOCAL SERV... | 00 | 5,272 K | Not allowed |
| backgroundTaskHost.... | 51380 | Suspended | japik | 00 | 0 K | Disabled |
| BCClipboard.exe | 7604 | Running | japik | 00 | 800 K | Disabled |
| browser_broker.exe | 23380 | Running | japik | 00 | 396 K | Disabled |
| caller64.exe | 25016 | Running | japik | 00 | 468 K | Disabled |
| CCLibrary.exe | 22564 | Running | japik | 00 | 40 K | Disabled |
| CCXProcess.exe | 7720 | Running | japik | 00 | 68 K | Disabled |
| chrome.exe | 51356 | Running | japik | 00 | 90,420 K | Disabled |

The Role of Threads

Every Windows process contains an initial “thread” (or the main thread) that functions as the entry point for the application. Chapter 15 examines the details of building multithreaded applications under the .NET Core platform; however, to facilitate the topics presented here, you need a few working definitions. First, a thread is a path of execution within a process. Formally speaking, the first thread created by a process’s entry point is termed the primary thread(or the main thread). Any .NET Core program (console application, Windows service, WPF application, etc.) marks its entry point with the `Main()` method. When this method is invoked, the primary thread is created automatically

Processes that contain a single primary thread of execution are intrinsically thread-safe, given the fact that there is only one thread that can access the data in the application at a given time. However, a single-threaded process (especially one that is GUI based) will often appear a bit unresponsive to the user if this single thread is performing a complex operation (such as printing out a lengthy text file, performing a mathematically intensive calculation, or attempting to connect to a remote server located thousands of miles away). Given this potential drawback of single-threaded applications, the operating systems that are supported by .NET Core (as well as the .NET Core platform) make it possible for the primary thread to spawn additional secondary threads (also termed worker threads) using a handful of API functions such as `CreateThread`.

Each thread (primary or secondary) becomes a unique path of execution in the process and has concurrent access to all shared points of data within the process. As you might have guessed, developers typically create additional threads to help improve the program’s overall responsiveness. Multithreaded processes provide the illusion that numerous activities are happening at more or less the same time. For example, an application may spawn a worker thread to perform a labor-intensive unit of work (again, such as printing a large text file). As this secondary thread is churning away, the main thread is still responsive to user input, which gives the entire process the potential of delivering greater performance. However, this may not actually be the case: using too many threads in a single process can actually degrade performance, as the CPU must switch between the active threads in the process (which takes time). On some machines, multithreading is most commonly an illusion provided by the OS. Machines that host a single (non-hyperthreaded) CPU do not have the ability to literally handle multiple threads at the same time. Rather, a single CPU will execute one thread for a unit of time (called a time slice) based in part on the thread’s priority level. When a thread’s time slice is up, the existing thread is suspended to allow another thread to perform its business. For a thread to remember what was happening before it was kicked out of the way, each thread is given the ability to write to Thread Local Storage (TLS) and is provided with a separate call stack, as illustrated in Figure



If the subject of threads is new to you, don't sweat the details. At this point, just remember that a thread is a unique path of execution within a Windows process. Every process has a primary thread (created via the executable's entry point) and may contain additional threads that have been programmatically created.

thread was defined as a path of execution within an executable application. While many .NET Core applications can live happy and productive single-threaded lives, an assembly's primary thread (spawned by the runtime when `Main()` executes) may create secondary threads of execution at any time to perform additional units of work. By creating additional threads, you can build more responsive (but not necessarily faster executing on single-core machines) applications. The `System.Threading` namespace was released with .NET 1.0 and offers one approach to build multithreaded applications. The `Thread` class is perhaps the core type, as it represents a given thread. If you want to programmatically obtain a reference to the thread currently executing a given member, simply call the static `Thread.CurrentThread` property, like so:

ThreadPool

The next thread-centric topic you will examine in this chapter is the role of the runtime thread pool. There is a cost with starting a new thread, so for purposes of efficiency, the thread pool holds onto created (but inactive) threads until needed. To allow you to interact with this pool of waiting threads, the `System.Threading` namespace provides the `ThreadPool` class type

Asynchrony— This means that your program performs non-blocking operations. For example, it can initiate a request for a remote resource via HTTP and then go on to do some other task while it waits for the response to be received. It's a bit like when you send an email and then go on with your life without waiting for a response.

Parallelism— This means that your program leverages the hardware of multi-core machines to execute tasks at the same time by breaking up work into tasks, each of which is executed on a separate core. It's a bit like singing in the shower: you're actually doing two things at exactly the same time.

Multithreading— This is a software implementation allowing different threads to be executed concurrently. A multithreaded program appears to be doing several things at the same time even when it's running on a single-core machine. This is a bit like chatting with different people through various IM windows; although you're actually switching back and forth, the net result is that you're having multiple conversations at the same time.