10.1.1 Linux Processes

Click one of the buttons to take you to that part of the video.

Linux Processes 0:00-0:28

In this lesson, we're going to review how Linux handles processes. So what exactly is a process? For our purposes, a process is a program that's been loaded from a long-term storage device, like a hard disk drive, into your system RAM and is currently being processed by the CPU on the motherboard.

There are actually many different types of programs that can be run in order to create a process. Let's take a look at what they are.

Create a Process 0:27-2:41

The first type of program that you can run to create a process are binary executables. These are programs that were originally created as a text file using some type of program language, like C or C++ or something like that. Then, that text file was run through a compiler to create a binary executable file that can actually be processed by the CPU.

That's not the only way you can create a process on a Linux system. You can also create a process by running an internal shell command. Understand that many of the commands that you enter at the shell prompt aren't actual binary files.

Instead, they are commands that are rolled into the shell program itself. For example, if you were to enter the exit command at the shell prompt, you're actually running an internal shell command.

If you were to look in the file system, you would not find a binary executable file anywhere on your Linux system named exit. Instead, the necessary computer code associated with the exit function is stored within the shell program code itself.

The last way that you can create a process on a Linux system is to run a shell script. These are text files that are executed through the shell itself. Basically, you open up a text file, and then you put in all the commands that you want to run within the text of the shell script.

Then you make that shell script executable, and the shell will dynamically interpret that shell script when you run it and do what whatever it is you've programmed that script to do.

Because the shell scripts are not compiled, we say that they are interpreted. Binary executables are compiled. They're compiled into a binary file contained in the zeros and ones that the CPU can run.

Likewise, these internal shell commands are compiled into the shell program itself. The shell scripts, on the other hand, are not compiled into zeros and ones. They are interpreted on the fly by the shell itself, and then a new process is created from the interpreted code.

Multi-Tasking 2:40-4:45

Be aware that the Linux operating system can actually run multiple processes concurrently, at the same time, on a single CPU. Depending upon how your Linux system is being used, you may have a limited number of processes running at any given point in time.

If it's a heavily used system, it may have hundreds of processes running concurrently. In fact, I can almost guarantee you that your Linux system will have many, many process running at the same time.

Just a second ago, I said that it can have multiple processes running concurrently. I use the term concurrently loosely, because most CPUs can't truly run multiple processes at the same time.

Instead what happens, is the Linux operating system itself quickly switches between the various processes running on the CPU, making it appear as if the CPU is working on multiple processes concurrently.

The CPU, for example, might run this process for a second, then quickly switch to this second process, and then quickly switch to this third process, and then switch back to the first process again, go to the second one again, and then to the third one, and do the whole thing all over again.

This makes it appear as if the CPU is working on all these multiple processes concurrently. However, what's really happening is the CPU actually executes only a single process at a time. All the other processes that are currently running are actually waiting in the background for their turn with the CPU.

The operating system maintains a schedule that determines when each process is allowed to access the CPU. This whole thing is called multi-tasking.

Because the switching between processes happens so fast, it appears to me and you at least--to carbon-based life forms--that the CPU is actually executing these multiple processes all at the same time. Be aware, however, that there are two exceptions to this rule.

Multi-Core CPUs 4:44-5:53

The first exception are multi-core CPUs. We have just one physical CPU installed in a slot on the motherboard, but within that CPU we have two cores that function as two separate CPUs. Because there are multiple cores inside the CPU package, each one can run its own set of processes like we saw previously.

One core can execute one process while the other core works on a different process. Each core still multi-tasks like we saw before. The first core works on process 23, then it jumps to 24, then it jumps to 25, then it jumps back to 23, then it goes to 24, then it goes to 25, and so on. While it's working on 23, 24, and 25, the second core is working at the same time on process 26, 27, 28, and it's doing the same round robin multi-tasking thing that the first core is.

Effectively, this one CPU here could be running process 23 and process 26 at exactly the same time. One on the first core and one on the second core.

Multithreading CPUs 5:53-6:36

The second exception to this rule are hyper-threading CPUs. A hyper-threading CPU is designed such that one single CPU can run two processes at a time. Before we go any further, I need to point out that on Intel CPUs, this feature is called hyper-threading.

But on AMD CPUs, this feature is called symmetric multithreading. However, in common usage, the two terms tend to just be used interchangeably. So if you hear the word hyper-threading or multithreading, we're actually referring to the same thing. In this case, we can have one processor running process 23 and process 24.

Multi-Core Multithreading CPUs 6:36-7:55

Multithreading CPUs also multi-task, so it runs process 23 and 24, then it switches--as we talked about earlier--to process 25 and 26. Then it jumps back and works on 23 and 24 again, and then it switches to 25 and 26 again, then jumps back to 23 and 24 again, and keeps going through the multi-tasking loop.

Multithreading CPUs, like multi-core CPUs, can run more than one process at once. This really speeds up the processing in the Linux system. If you really want to speed things up, you can look at a multi-core CPU that supports hyper-threading.

In this situation, we have two or more cores in the processor, and each of those cores is capable of running on two processes at a time. Depending upon how many cores you have in the processor, this can make for a wicked fast system.

In this situation, we have one processor. It has two cores, and each of these cores supports hyper-threading, so this one CPU in the slot can run four processes at the same time. It'll first work on process 23 and 24 on the first core, and process 27 and 28 on the second core.

User Processes 7:51-8:53

It also uses a multi-tasking, just like a standard single-core single-thread CPU. It'll switch and work on process 25 and 26 and 29 and 30. They key point to remember, though, is that it's working on four separate processes concurrently, at the same time, on the system.

Before we go on, you need to understand that the Linux operating system implements several types of processes. Not all of the processes running on your Linux system are the same.

First of all, some processes are created by the end user when they run a command from the shell prompt, or maybe they click on an icon in the graphical user interface. These are user processes because they were created by the user.

User processes are usually associated with some kind of end-user program running on the system. In this example, I ran the LibreOffice command from the shell prompt, which will load the LibreOffice suite on the system.

System Processes 8:49-10:36

When I did this, two user processes were created: oosplash and soffice.bin. The important thing to remember about user process is that they're called from within a shell, and they are therefore associated with that shell session.

Not all processes running on your system are user processes. In fact, most of the processes running on a Linux system will be system processes. These are created by daemons. Unlike a user process, a system process usually does not provide any type of end-user interface that they can use to interact with it.

Instead, it's used to provide a system service-like a web server, or an FTP server, or file sharing, or printer sharing. These system processes run in the background, and your end users don't typically interact directly with them.

A list of system processes are shown here. You'll notice right away that this system has many, many system processes running. You see just a fraction of the total number of processes listed by the output of this command here. System processes are usually, but not always, loaded by the Linux operating system itself when the system first boots up.

Therefore, because they're loaded before system boot up, they're not associated with a particular shell instance. This is a key difference between user processes and system processes. User processes are tied to the particular shell that they were called from, whereas system processes are not.

Summary 10:37-10:49

That's it for this lesson. In this lesson, we introduced you to Linux processes. We talked about how processes are created. We talked about multi-tasking. We talked about multi-core and multi-threading. We also talked about the differences between system and user processes.
