10.2.6 Process Termination

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

Process Termination 0:00-0:52

Let's spend a few minutes discussing how you go about killing a running process on a Linux system from the shell prompt. Normally, you don't do this. Instead, you would use the appropriate exit function that's coded into nearly all applications to end its process.

For example, if I were in the vi editor and I wanted to get out, I would enter a colon (:) to enter into command line mode and then enter exit to end the vi process. However, there are times when this won't work.

For example, you may have a process that is hung, and no matter what you do, you cannot get that process to exit out properly. In this situation, you may need to manually kill that hung process.

There are two different ways that you can do this. One way is to use either the kill or the killall command. Another option is to use the pkill command.

Use kill 0:53-5:03

Let's begin this lesson by looking at the kill command. As its name implies, the kill command is used to terminate a running process, and the syntax is shown here.

We enter kill, and then a dash (-), and then we enter which kill signal we want to send, followed by the process ID number of the process that we want to get rid of. Understand that the kill command is very flexible.

You actually have about 64 different types of kill signals that you can send to a process. Some are very kind and gentle; some are very brutal.

Some of the most useful kill signals are shown here. The first one you need to be familiar with is SIGHUP, or signal hang up, is what it stands for. This is kill signal number one. That's a very kind and gentle kill signal.

Another one you can send is the SIGINT kill signal. This is kill signal number two. This one does try to stop the process, but it does so very politely. This is the equivalent of pressing 'Ctrl+C' on the keyboard to stop the process from running. If the process is behaving properly, it should respond. If it's not, it probably won't.

If you need to get a little meaner, you can use SIGTERM, or kill signal 15. This signal tells the process to terminate immediately. In fact, if you don't specify a signal here with the kill command, this is the one that will be sent by default.

The key thing to remember about SIGTERM, kill signal 15, is the fact that it tries to be a little bit polite, in that it allows the process to clean up after itself as it's exiting.

For some processes that get hung, they're not going to respond to either SIGINT or SIGTERM; in which case, you're going to have to go to SIGKILL. This is kill signal number 9. This one is a lot more brutal.

If you have a process that's hung and it's hung badly, this option will force it to stop. And, as I said, it's kind of a brute force signal, because if you execute it against the process that's hung, it does not really give the process an opportunity to clean up after itself.

When I say clean up after itself, I'm talking about freeing up resources that may have been allocated to that process, such as memory addresses associated with that process. Those resources are going to remain allocated to the process that you killed until you restart the system itself.

When you run the kill command, you can use either the name of the kill signal, or you can use the number of the kill signal. Either one works just fine.

One important thing to remember about using the kill command is the fact that you do have to specify the process ID number of the process that you want to kill. Therefore, more than likely, you're going to need to use ps or possibly top at the shell prompt to first identify the process ID number of the process that you want to kill.

In this example, I have the gedit program running. Its process ID number is 4133. If I wanted to kill that process, I would enter kill, followed by the signal that I want to send, followed by the process ID number that I want to kill.

Notice that I used SIGTERM in this command. That's the same as signal 15, which basically allows gedit to clean up after itself as it's exiting, freeing up any system resources that were allocated to it. This will allow the gedit process to exit out cleanly.

This brings up a mistake that I've seen many new Linux system administrators make when they start working with the kill command. They just want to bypass all the polite kill signals and go right for the mean brutal ones. They're going for the jugular first.

This really isn't the best way to do things. Yes, using SIGKILL will work most of the time, but it's actually better if you try the other, cleaner, more polite signals first. Only if these kill signals fail should you try a more brutal kill signal.

Kill Processes Properly 5:04-5:50

If you experience a hung process that has to be killed with a kill command, I suggest that you use the sequence that you see here. First, try sending a SIGINT kill signal with the kill command, just see if the process responds.

If it doesn't, then try sending it a SIGTERM kill signal. My experience has been that usually, nine times out of ten, this actually fixes the problem. And it's better because it allows that process to exit cleanly, freeing up any resources, system resources, that are allocated to it.

But, on rare occasions, you may find that SIGTERM doesn't work; in which case, you need to go on to Step 3 and then get mean with it and send it a SIGKILL--kill signal 9.

Use killall 5:51-6:57

In addition to kill, there is a second command you can use to kill processes. That is the killall command. killall is very similar to kill. In fact, the syntax is almost the same.

The key difference is the fact that with killall, you specify the command name of the process to be killed, instead of its process ID number. For example, here, once again, we need to kill that pesky gedit process. I'm going to send it a kill signal 15.

To do this with killall, I enter killall -15, followed by the name of the process, gedit. This will send the SIGTERM signal to the process named gedit. Basically, it allows you to kill the process without having to figure out what its process ID number is.

I strongly suggest that you spend a little bit of time looking at the man page for both kill and killall. Both of these commands are actually quite extensive, and there are a lot of different things you can do with it.

We don't have time or space to cover all of the different options here. For example, in the man page, you'll see that you can use the dash u option with the kill and killall commands to kill processes that are owned by a specific user.

Use pkill 6:58-9:12

In addition to the kill and killall commands, you can also use another command to kill running processes, and that's the pkill command. The pkill command is a close cousin of the pgrep command. In fact, if you look the man page for pkill, you'll see that it's exactly the same man page as that for pgrep.

The two commands use exactly the same man page and use the exact same options. pkill is really useful. You can use pkill to search for processes that match a specific search criteria that you specify, and then send those matching processes a particular kill signal.

In this example, I need it to search through all of my running processes for any process whose name contains the term gedit. I actually only have one, but it is possible that I could have multiple processes that have gedit it in their name. To find all those processes, I would enter pkill, followed by the kill signal I want to send--either its name or number.

In this case, we’re going to send kill signal 15, SIGTERM, and then specify the term that I want to search for in the name; in which case, I'm looking for gedit. And this will kill every process that has gedit in its process name.

The last topic we need to address before we end this lesson, is that of how to keep a process running, even if you log out from the system. As we've discussed, various signals can be sent to running processes to indicate that a system event of some type has occurred, and that the process needs to respond in some way.

A very commonly used signal is the hang up signal, which we talked about earlier in this lesson--SIGHUP. Understand that Linux keeps track of which processes are being run by each shell session, and when a user logs out of a shell session, Linux will send a SIGHUP signal to all the programs associated with that shell session.

Normally, each process will respond as it's supposed to when it receives a SIGHUP signal. However, a process can also be told to ignore any SIGHUP signals, which will allow it to remain running, even if you log out of your shell session. One way to do this is to use the nohup utility. nohup stands for no hang up.

Use nohup 9:13-11:05

For example, let's suppose I've created a shell script called updatemydb that automatically updates a local database with information from an external data source. This script takes a long time to run.

It may take three or four hours, and I'm leaving for the night, and I really don't want to leave my system logged in while it runs, nor do I want to stay here for three or four more hours, waiting for it to finish.

In order to let it run and for me to be able log out and leave, I could enter nohup at the shell prompt, followed by the command that I want to run, updatemydb, then I'm going to run it in the background with the ampersand (&) command. I can then go ahead and log out, and even though I've logged out, the process created by this command will continue to run.

Because the shell session where I ran the command from is going to be gone when I log out, we've got a problem if the command generates something that's sent to the standard out, such as text on the screen like, "Command was successful" or whatever, or if there's an error message going to the standard out.

Because the shell session is gone, standard out and standard error have no place to display their information. Therefore, if the command generates output that is sent to the standard out or standard error, what nohup will do is redirect the standard out or standard error to a text file in your /home directory called nohup.out, and you can look at it after the command is done, when you come back in--in the morning, for example--and make sure everything ran properly.

It's important to note that any command that you run with nohup is only immune to signal hang up signals. All other kill signals will still work. For example, I could run updatemydb with nohup, but then still send a SIGTERM kill signal with kill, and it will respond. It will shut down.

Use screen Commands 11:06-13:32

Another command you can use to accomplish a similar thing is the screen command. The screen command is kind of interesting, and it's very useful, especially if you're accessing a Linux system remotely over the network, using an SSH connection.

The key benefit of screen is it allows you to use multiple shell windows from within a single SSH session. This is especially beneficial, because it can keep an SSH shell active, even if the network goes down. Whatever it was you were running will continue to run, even though you've lost your connection.

You can even use screen to disconnect and then reconnect to a particular shell session from a different location, without having to stop and then restart whatever process you were working on. You run screen by simply entering the screen command at the shell prompt.

After you do so, you'll see a shell prompt displayed, and it doesn't look any different than the shell prompt you were working with before. However, it is different, because now you're inside of a screen window instead of within the shell session itself.

The window functions just like a normal shell session; you can run commands and interact with programs just as you would from any other shell prompt.

However, you can do other things. If you press Ctrl+A within the screen window, then whatever you type after CTRL+A will be sent to the screen process instead of for whatever process you're running in the shell session.

For example, if you press Ctrl+A, followed by C, it will cause a new screen window to be created, and the old window that you were working in will remain active, along with any processes that were running within it. This allows you, basically, to run multiple commands within one single SSH session.

For example, you can run top in one window, open up a new window, and then use an email client to check your email. If you press Ctrl+A, followed by N, you can toggle between all of your open windows within the screen. Pressing CTRL+A, followed by D will detach your screen window and drop you back to your original shell prompt, but whatever processes you had running in all the various screen windows will remain running.

In fact, you can completely log out of the server, and even though you're logged out and gone, everything that you started running will continue running within that detached window. And, most likely, you're going to want to reattach to that window at some point.

To do that, you use the screen -r command, and that will reattach you to any detached screen windows you may have started previously. If you have multiple screen windows, you'll have to specify which one you want to reattach to.

Summary 13:33-13:36

That's it for this lesson. In this lesson we reviewed killing a running process on a Linux system from the shell prompt.
