# Chapter 8: Monitoring and Managing Linux Processes

[**Listing Processes**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Quiz: Listing Processes**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s02/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Controlling Jobs**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s03/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Controlling Jobs**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s04/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Killing Processes**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s05/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Killing Processes**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s06/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Monitoring Process Activity**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s07/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Monitoring Process Activity**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s08/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Lab: Monitoring and Managing Linux Processes**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s09/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Summary**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch08s10/9a2ef70f-4e72-42df-a498-b694b274af27)

# Abstract

|  |  |
| --- | --- |
| **Goal** | * Evaluate and control processes running on a Red Hat Enterprise Linux system. |
| **Objectives** | * Get information about programs running on the system so that you can determine status, resource use, and ownership, so you can control them. * Use Bash job control to manage multiple processes started from the same terminal session. * Control and terminate processes that are not associated with your shell, and forcibly end user sessions and processes. * Describe what load average is and determine processes responsible for high resource use on a server. |
| **Sections** | * Listing Processes (and Quiz) * Controlling Jobs (and Guided Exercise) * Killing Processes (and Guided Exercise) * Monitoring Process Activity (and Guided Exercise) |
| **Lab** | * Monitoring and Managing Linux Processes |

# Listing Processes

## Objectives

After completing this section, you should be able to get information about programs running on a system to determine status, resource use, and ownership, so you can control them.

## Definition of a Process

A process is a running instance of a launched, executable program. A process consists of:

* An address space of allocated memory
* Security properties including ownership credentials and privileges
* One or more execution threads of program code
* Process state

The environment of a process includes:

* Local and global variables
* A current scheduling context
* Allocated system resources, such as file descriptors and network ports

An existing (parent) process duplicates its own address space (**fork**) to create a new (child) process structure. Every new process is assigned a unique process ID (PID) for tracking and security. The PID and the parent's process ID (PPID) are elements of the new process environment. Any process may create a child process. All processes are descendants of the first system process, **systemd** on a Red Hat Enterprise Linux 8 system).

|  |
| --- |
|  |

Figure 8.1: Process life cycle

Through the fork routine, a child process inherits security identities, previous and current file descriptors, port and resource privileges, environment variables, and program code. A child process may then exec its own program code. Normally, a parent process sleeps while the child process runs, setting a request (wait) to be signaled when the child completes. Upon exit, the child process has already closed or discarded its resources and environment. The only remaining resource, called a zombie, is an entry in the process table. The parent, signaled awake when the child exited, cleans the process table of the child's entry, thus freeing the last resource of the child process. The parent process then continues with its own program code execution.

### **Describing Process States**

In a multitasking operating system, each CPU (or CPU core) can be working on one process at a single point in time. As a process runs, its immediate requirements for CPU time and resource allocation change. Processes are assigned a state, which changes as circumstances dictate.

|  |
| --- |
|  |

Figure 8.2: Linux process states

Linux process states are illustrated in the previous diagram and described in the following table:

## ****Table 8.1. Linux Process States****

| **Name** | **Flag** | **Kernel-defined state name and description** |
| --- | --- | --- |
| Running | R | TASK\_RUNNING: The process is either executing on a CPU or waiting to run. Process can be executing user routines or kernel routines (system calls), or be queued and ready when in the Running (or Runnable) state. |
| Sleeping | S | TASK\_INTERRUPTIBLE: The process is waiting for some condition: a hardware request, system resource access, or signal. When an event or signal satisfies the condition, the process returns to Running. |
| D | TASK\_UNINTERRUPTIBLE: This process is also Sleeping, but unlike S state, does not respond to signals. Used only when process interruption may cause an unpredictable device state. |
| K | TASK\_KILLABLE: Identical to the uninterruptible D state, but modified to allow a waiting task to respond to the signal that it should be killed (exit completely). Utilities frequently display Killable processes as D state. |
| I | TASK\_REPORT\_IDLE: A subset of state D. The kernel does not count these processes when calculating load average. Used for kernel threads. Flags TASK\_UNINTERRUPTABLE and TASK\_NOLOAD are set. Similar to TASK\_KILLABLE, also a subset of state D. It accepts fatal signals. |
| Stopped | T | TASK\_STOPPED: The process has been Stopped (suspended), usually by being signaled by a user or another process. The process can be continued (resumed) by another signal to return to Running. |
| T | TASK\_TRACED: A process that is being debugged is also temporarily Stopped and shares the same T state flag. |
| Zombie | Z | EXIT\_ZOMBIE: A child process signals its parent as it exits. All resources except for the process identity (PID) are released. |
| X | EXIT\_DEAD: When the parent cleans up (reaps) the remaining child process structure, the process is now released completely. This state will never be observed in process-listing utilities. |

## ****Why Process States are Important****

When troubleshooting a system, it is important to understand how the kernel communicates with processes and how processes communicate with each other. At process creation, the system assigns the process a state. The S column of the **top** command or the STAT column of the **ps** show the state of each process. On a single CPU system, only one process can run at a time. It is possible to see several processes with a state of R. However, not all of them will be running consecutively, some of them will be in status waiting.

**[user@host ~]$ top**

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1 root 20 0 244344 13684 9024 **S** 0.0 0.7 0:02.46 systemd

2 root 20 0 0 0 0 **S** 0.0 0.0 0:00.00 kthreadd

...output omitted...

**[user@host ~]$ ps aux**

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

...output omitted...

root 2 0.0 0.0 0 0 ? **S** 11:57 0:00 [kthreadd]

student 3448 0.0 0.2 266904 3836 pts/0 **R+** 18:07 0:00 ps aux

...output omitted...

Process can be suspended, stopped, resumed, terminated, and interrupted using signals. Signals are discussed in more detail later in this chapter. Signals can be used by other processes, by the kernel itself, or by users logged into the system.

## Listing Processes

The **ps** command is used for listing current processes. It can provide detailed process information, including:

* User identification (UID), which determines process privileges
* Unique process identification (PID)
* CPU and real time already expended
* How much memory the process has allocated in various locations
* The location of process stdout, known as the controlling terminal
* The current process state

### **Important**

The Linux version of **ps** supports three option formats:

* UNIX (POSIX) options, which may be grouped and must be preceded by a dash
* BSD options, which may be grouped and must not be used with a dash
* GNU long options, which are preceded by two dashes

For example, **ps -aux** is not the same as **ps aux**.

Perhaps the most common set of options, aux, displays all processes including processes without a controlling terminal. A long listing (options lax) provides more technical detail, but may display faster by avoiding user name lookups. The similar UNIX syntax uses the options -ef to display all processes.

**[user@host ~]$ ps aux**

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 0.1 0.1 51648 7504 ? Ss 17:45 0:03 /usr/lib/systemd/syst

root 2 0.0 0.0 0 0 ? S 17:45 0:00 [kthreadd]

root 3 0.0 0.0 0 0 ? S 17:45 0:00 [ksoftirqd/0]

root 5 0.0 0.0 0 0 ? S< 17:45 0:00 [kworker/0:0H]

root 7 0.0 0.0 0 0 ? S 17:45 0:00 [migration/0]

...output omitted...

**[user@host ~]$ ps lax**

F UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY TIME COMMAND

4 0 1 0 20 0 51648 7504 ep\_pol Ss ? 0:03 /usr/lib/systemd/

1 0 2 0 20 0 0 0 kthrea S ? 0:00 [kthreadd]

1 0 3 2 20 0 0 0 smpboo S ? 0:00 [ksoftirqd/0]

1 0 5 2 0 -20 0 0 worker S< ? 0:00 [kworker/0:0H]

1 0 7 2 -100 - 0 0 smpboo S ? 0:00 [migration/0]

...output omitted...

**[user@host ~]$ ps -ef**

UID PID PPID C STIME TTY TIME CMD

root 1 0 0 17:45 ? 00:00:03 /usr/lib/systemd/systemd --switched-ro

root 2 0 0 17:45 ? 00:00:00 [kthreadd]

root 3 2 0 17:45 ? 00:00:00 [ksoftirqd/0]

root 5 2 0 17:45 ? 00:00:00 [kworker/0:0H]

root 7 2 0 17:45 ? 00:00:00 [migration/0]

...output omitted...

By default, **ps** with no options selects all processes with the same effective user ID (EUID) as the current user, and which are associated with the same terminal where **ps** was invoked.

* Processes in brackets (usually at the top of the list) are scheduled kernel threads.
* Zombies are listed as exiting or defunct.
* The output of **ps** displays once. Use **top** for a process display that dynamically updates.
* **ps** can display in tree format so you can view relationships between parent and child processes.
* The default output is sorted by process ID number. At first glance, this may appear to be chronological order. However, the kernel reuses process IDs, so the order is less structured than it appears. To sort, use the -O or --sort options. Display order matches that of the system process table, which reuses table rows as processes die and new ones are created. Output may appear chronological, but is not guaranteed unless explicit -O or --sort options are used.

## References

**info libc signal** (GNU C Library Reference Manual)

* Section 24: Signal Handling

**info libc processes** (GNU C Library Reference Manual)

* Section 26: Processes

ps(1) and signal(7) man pages

# Quiz: Listing Processes

1. Which state represents a process that has been stopped or suspended?

D) T

2. Which state represents a process that has released all of its resources except its PID?

E) Z

3. Which process does a parent use to duplicate to create a new child process?

B) fork

4. Which state represents a process that is sleeping until some condition is met?

C) S

# Controlling Jobs

## Objectives

After completing this section, you should be able to use Bash job control to manage multiple processes started from the same terminal session.

## Describing Jobs and Sessions

*Job control* is a feature of the shell which allows a single shell instance to run and manage multiple commands.

A *job* is associated with each pipeline entered at a shell prompt. All processes in that pipeline are part of the job and are members of the same *process group*. If only one command is entered at a shell prompt, that can be considered to be a minimal “pipeline” of one command, creating a job with only one member.

Only one job can read input and keyboard generated signals from a particular terminal window at a time. Processes that are part of that job are *foreground* processes of that *controlling terminal*.

A *background* process of that controlling terminal is a member of any other job associated with that terminal. Background processes of a terminal cannot read input or receive keyboard generated interrupts from the terminal, but may be able to write to the terminal. A job in the background may be stopped (suspended) or it may be running. If a running background job tries to read from the terminal, it will be automatically suspended.

Each terminal is its own *session*, and can have a foreground process and any number of independent background processes. A job is part of exactly one session: the one belonging to its controlling terminal.

The **ps** command shows the device name of the controlling terminal of a process in the TTY column. Some processes, such as *system daemons*, are started by the system and not from a shell prompt. These processes do not have a controlling terminal, are not members of a job, and cannot be brought to the foreground. The **ps** command displays a question mark (?) in the TTY column for these processes.

## Running Jobs in the Background

Any command or pipeline can be started in the background by appending an ampersand (&) to the end of the command line. The Bash shell displays a *job number* (unique to the session) and the PID of the new child process. The shell does not wait for the child process to terminate, but rather displays the shell prompt.

**[user@host ~]$ sleep 10000 &**

[1] 5947

**[user@host ~]$**

## Note

When a command line containing a pipe is sent to the background using an ampersand, the PID of the last command in the pipeline is used as output. All processes in the pipeline are still members of that job.

**[user@host ~]$ example\_command | sort | mail -s "Sort output" &**

[1] 5998

You can display the list of jobs that Bash is tracking for a particular session with the **jobs** command.

**[user@host ~]$ jobs**

[1]+ Running sleep 10000 &

**[user@host ~]$**

A background job can be brought to the foreground by using the **fg** command with its job ID (%*job number*).

**[user@host ~]$ fg %1**

sleep 10000

In the preceding example, the **sleep** command is now running in the foreground on the controlling terminal. The shell itself is again asleep, waiting for this child process to exit.

To send a foreground process to the background, first press the keyboard generated *suspend* request (**Ctrl**+**z**) in the terminal.

sleep 10000

^Z

[1]+ Stopped sleep 10000

[user@host ~]$

The job is immediately placed in the background and is suspended.

The **ps j** command displays information relating to jobs. The PID is the unique *process ID* of the process. THe PPID is the PID of the *parent process* of this process, the process that started (forked) it. The PGID is the PID of the *process group leader*, normally the first process in the job's pipeline. The SID is the PID of the *session leader*, which (for a job) is normally the interactive shell that is running on its controlling terminal. Since the example **sleep** command is currently suspended, its process state is T.

**[user@host ~]$ ps j**

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

2764 2768 2768 2768 pts/0 6377 Ss 1000 0:00 /bin/bash

2768 5947 5947 2768 pts/0 6377 T 1000 0:00 sleep 10000

2768 6377 6377 2768 pts/0 6377 R+ 1000 0:00 ps j

To start the suspended process running in the background, use the **bg** command with the same job ID.

**[user@host ~]$ bg %1**

[1]+ sleep 10000 &

The shell will warn a user who attempts to exit a terminal window (session) with suspended jobs. If the user tries exiting again immediately, the suspended jobs are killed.

## Note

Note the + sign after the [1] in the examples above. The + sign indicates that this job is the current default job. That is, if a command is used that expects a %*job number* argument and a job number is not provided, then the action is taken on the job with the + indicator.

## References

Bash info page (*The GNU Bash Reference Manual*) <https://www.gnu.org/software/bash/manual>

* Section 7: Job Control

bash(1), builtins(1), ps(1), sleep(1) man pages

# Guided Exercise: Controlling Jobs

In this exercise, you will start, suspend, background, and foreground multiple processes using job control.

## Outcomes

You should be able to use job control to suspend and restart user processes.

Log in to workstation as student using student as the password.

On workstation, run the **lab processes-control start** command. This script ensures that servera is available.

**[student@workstation ~]$ lab processes-control start**

1. On workstation, open two terminal windows side by side. In this section, these two terminals are referred to as *left* and *right*. In each terminal, use the **ssh** command to log in to servera as the student user.

**[student@workstation ~]$ ssh student@servera**

*...output omitted...*

[student@servera ~]$

1. In the *left* window, create a new directory called /home/student/bin. In the new directory, create a shell script called control. Make the script executable.
   1. Use the **mkdir** command to create a new directory called /home/student/bin.

**[student@servera ~]$ mkdir /home/student/bin**

* 1. Use the **vim** command to create a script called control in the /home/student/bin directory. To enter Vim interactive mode, press the **i** key. Use the **:wq** command to save the file.

**[student@servera ~]$ vim /home/student/bin/control**

#!/bin/bash

while true; do

echo -n "$@ " >> ~/control\_outfile

sleep 1

done

Note

The *control* script runs until terminated. It appends command-line arguments to the file ~/control\_outfile once per second.

* 1. Use the **chmod** command to make the control file executable.

**[student@servera ~]$ chmod +x /home/student/bin/control**

1. Execute the **control** script. The script continuously appends the word "technical" and a space to the file ~/control\_outfile at one second intervals.

Note

You are able to execute your **control** script because it is located in your PATH, and has been made executable.

**[student@servera ~]$ control technical**

1. In the right terminal shell, use the **tail** command with the -f option to confirm that the new process is writing to the /home/student/control\_outfile file.

**[student@servera ~]$ tail -f ~/control\_outfile**

technical technical technical technical

*...output omitted...*

1. In the left terminal shell, press **Ctrl**+**z** to suspend the running process. The shell returns the job ID in square brackets. In the right window, confirm that the process output has stopped.

^Z

[1]+ Stopped control technical

[student@servera ~]$

technical technical technical technical

*...no further output...*

1. In the left terminal shell, view the **jobs** list. Remember that the + sign indicates the default job. Restart the job in the background. In the right terminal shell, confirm that the process output is again active.
   1. Using the **jobs** command, view the list of jobs.

**[student@servera ~]$ jobs**

[1]+ Stopped control technical

* 1. Using the **bg** command, restart the **control** job in the background.

**[student@servera ~]$ bg**

[1]+ control technical &

* 1. Use the **jobs** command to confirm that the **control** job is running again.

**[student@servera ~]$ jobs**

[1]+ Running control technical &

* 1. In the right terminal shell, confirm that the **tail** command is producing output.

*...output omitted...*

technical technical technical technical technical technical technical technical

1. In the left terminal shell, start two more **control** processes to append to the ~/output file. Use the ampersand (&) to start the processes in the background. Replace technical with documents and then with database. Replacing the arguments helps to differentiate between the three processes.

**[student@servera ~]$ control documents &**

[2] 6579

[student@servera ~]$

**[student@servera ~]$ control database &**

[3] 6654

Note

The job number of each new process is printed in square brackets. The second number is the unique system-wide process ID number (PID) for the process.

1. In the left terminal shell, use the **jobs** command to view the three running processes. In the right terminal shell, confirm that all three processes are appending to the file.

**[student@servera ~]$ jobs**

[1] Running control technical &

[2]- Running control documents &

[3]+ Running control database &

*...output omitted...*

technical documents database technical documents database technical documents database technical documents database

*...output omitted...*

1. Suspend the **control technical** process. Confirm that it has been suspended. Terminate the **control documents** process and confirm that it has been terminated.
   1. In the left terminal shell, use the **fg** command with the job ID to foreground the **control technical** process. Press **Ctrl**+**z** to suspend the process. Use the **jobs** command to confirm that the process is suspended.

**[student@servera ~]$ fg %1**

control technical

^Z

[1]+ Stopped control technical

**[student@servera ~]$ jobs**

[1]+ Stopped control technical

[2] Running control documents &

[3]- Running control database &

* 1. In the right terminal shell, confirm that the **control technical** process is no longer sending output.

database documents database documents database

*...no further output...*

* 1. In the left terminal shell, use the **fg** command with the job ID to foreground the **control documents** process. Press **Ctrl**+**c** to terminate the process. Use the **jobs** command to confirm that the process is terminated.

**[student@servera ~]$ fg %2**

control documents

^C

**[student@servera ~]$ jobs**

[1]+ Stopped control technical

[3]- Running control database &

* 1. In the right terminal shell, confirm that the **control documents** process is no longer sending output.

*...output omitted...*

database database database database database database database database

*...no further output...*

1. In the left window, use the **ps** command with the jT option to view the remaining jobs. The suspended jobs have a state of T. The other background jobs are sleeping (S).

**[student@servera ~]$ ps jT**

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

27277 27278 27278 27278 pts/1 28702 Ss 1000 0:00 -bash

27278 28234 28234 27278 pts/1 28702 T 1000 0:00 /bin/bash /home/student/bin/control technical

27278 28251 28251 27278 pts/1 28702 S 1000 0:00 /bin/bash /home/student/bin/control database

28234 28316 28234 27278 pts/1 28702 T 1000 0:00 sleep 1

28251 28701 28251 27278 pts/1 28702 S 1000 0:00 sleep 1

27278 28702 28702 27278 pts/1 28702 R+ 1000 0:00 ps jT

1. In the left window, use the **jobs** command to view the current jobs. Terminate the **control database** process and confirm that it has been terminated.

**[student@servera ~]$ jobs**

[1]+ Stopped control technical

[3]- Running control database &

Use the **fg** command with the job ID to foreground the **control database** process. Press **Ctrl**+**c** to terminate the process. Use the jobs command to confirm that the process is terminated.

**[student@servera ~]$ fg %3**

control database

^C

**[student@servera ~]$ jobs**

[1]+ Stopped control technical

1. In the right terminal shell, use the **Ctrl**+**c** command to stop the **tail** command. Using the **rm** command, delete the ~/control\_outfile file.

*...output omitted...*

**Ctrl**+**c**

**[student@servera ~]$ rm ~/control\_outfile**

1. Log out from servera on both terminals.

**[student@servera ~]$ exit**

logout

Connection to servera closed.

**[student@servera ~]$ exit**

logout

Connection to servera closed.

**Finish**

On workstation, run the **lab processes-control finish** script to complete this exercise.

**[student@workstation ~]$ lab processes-control finish**

This concludes the guided exercise.

# Killing Processes

## Objectives

After completing this section, you should be able to:

* Use commands to kill and communicate with processes.
* Define the characteristics of a daemon process.
* End user sessions and processes.

## Process control using signals

A signal is a software interrupt delivered to a process. Signals report events to an executing program. Events that generate a signal can be an error, external event (an I/O request or an expired timer), or by explicit use of a signal-sending command or keyboard sequence.

The following table lists the fundamental signals used by system administrators for routine process management. Refer to signals by either their short (HUP) or proper (SIGHUP) name.

## ****Table 8.2. Fundamental Process Management Signals****

| **Signal number** | **Short name** | **Definition** | **Purpose** |
| --- | --- | --- | --- |
| 1 | HUP | Hangup | Used to report termination of the controlling process of a terminal. Also used to request process reinitialization (configuration reload) without termination. |
| 2 | INT | Keyboard interrupt | Causes program termination. Can be blocked or handled. Sent by pressing INTR key sequence (**Ctrl**+**c**). |
| 3 | QUIT | Keyboard quit | Similar to SIGINT; adds a process dump at termination. Sent by pressing QUIT key sequence (**Ctrl**+**\**). |
| 9 | KILL | Kill, unblockable | Causes abrupt program termination. Cannot be blocked, ignored, or handled; always fatal. |
| 15default | TERM | Terminate | Causes program termination. Unlike SIGKILL, can be blocked, ignored, or handled. The “polite” way to ask a program to terminate; allows self-cleanup. |
| 18 | CONT | Continue | Sent to a process to resume, if stopped. Cannot be blocked. Even if handled, always resumes the process. |
| 19 | STOP | Stop, unblockable | Suspends the process. Cannot be blocked or handled. |
| 20 | TSTP | Keyboard stop | Unlike SIGSTOP, can be blocked, ignored, or handled. Sent by pressing SUSP key sequence (**Ctrl**+**z**). |

## Note

Signal numbers vary on different Linux hardware platforms, but signal names and meanings are standardized. For command use, it is advised to use signal names instead of numbers. The numbers discussed in this section are for x86\_64 systems.

Each signal has a default action, usually one of the following:

* Term - Cause a program to terminate (exit) at once.
* Core - Cause a program to save a memory image (core dump), then terminate.
* Stop - Cause a program to stop executing (suspend) and wait to continue (resume).

Programs can be prepared to react to expected event signals by implementing handler routines to ignore, replace, or extend a signal's default action.

## ****Commands for Sending Signals by Explicit Request****

You signal the current foreground process by pressing a keyboard control sequence to suspend (**Ctrl**+**z**), kill (**Ctrl**+**c**), or core dump (**Ctrl**+**\**) the process. However, you will use signal-sending commands to send signals to a background process or to processes in a different session.

Signals can be specified as options either by name (for example, -HUP or -SIGHUP) or by number (the related -1). Users may kill their own processes, but root privilege is required to kill processes owned by others.

The **kill** command sends a signal to a process by PID number. Despite its name, the kill command can be used to send any signal, not just those for terminating programs. You can use the **kill -l** command to list the names and numbers of all available signals.

**[user@host ~]$ kill -l**

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

...output omitted...

**[user@host ~]$ ps aux | grep job**

5194 0.0 0.1 222448 2980 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job1

5199 0.0 0.1 222448 3132 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job2

5205 0.0 0.1 222448 3124 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job3

5430 0.0 0.0 221860 1096 pts/1 S+ 16:41 0:00 grep --color=auto job

**[user@host ~]$ kill 5194**

**[user@host ~]$ ps aux | grep job**

user 5199 0.0 0.1 222448 3132 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job2

user 5205 0.0 0.1 222448 3124 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job3

user 5783 0.0 0.0 221860 964 pts/1 S+ 16:43 0:00 grep --color=auto job

[1] Terminated control job1

**[user@host ~]$ kill -9 5199**

**[user@host ~]$ ps aux | grep job**

user 5205 0.0 0.1 222448 3124 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job3

user 5930 0.0 0.0 221860 1048 pts/1 S+ 16:44 0:00 grep --color=auto job

[2]- Killed control job2

**[user@host ~]$ kill -SIGTERM 5205**

user 5986 0.0 0.0 221860 1048 pts/1 S+ 16:45 0:00 grep --color=auto job

[3]+ Terminated control job3

The **killall** command can signal multiple processes, based on their command name.

**[user@host ~]$ ps aux | grep job**

5194 0.0 0.1 222448 2980 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job1

5199 0.0 0.1 222448 3132 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job2

5205 0.0 0.1 222448 3124 pts/1 S 16:39 0:00 /bin/bash /home/user/bin/control job3

5430 0.0 0.0 221860 1096 pts/1 S+ 16:41 0:00 grep --color=auto job

**[user@host ~]$ killall control**

[1] Terminated control job1

[2]- Terminated control job2

[3]+ Terminated control job3

**[user@host ~]$**

Use **pkill** to send a signal to one or more processes which match selection criteria. Selection criteria can be a command name, a process owned by a specific user, or all system-wide processes. The **pkill** command includes advanced selection criteria:

* Command - Processes with a pattern-matched command name.
* UID - Processes owned by a Linux user account, effective or real.
* GID - Processes owned by a Linux group account, effective or real.
* Parent - Child processes of a specific parent process.
* Terminal - Processes running on a specific controlling terminal.

**[user@host ~]$ ps aux | grep pkill**

user 5992 0.0 0.1 222448 3040 pts/1 S 16:59 0:00 /bin/bash /home/user/bin/control pkill1

user 5996 0.0 0.1 222448 3048 pts/1 S 16:59 0:00 /bin/bash /home/user/bin/control pkill2

user 6004 0.0 0.1 222448 3048 pts/1 S 16:59 0:00 /bin/bash /home/user/bin/control pkill3

**[user@host ~]$ pkill control**

[1] Terminated control pkill1

[2]- Terminated control pkill2

**[user@host ~]$ ps aux | grep pkill**

user 6219 0.0 0.0 221860 1052 pts/1 S+ 17:00 0:00 grep --color=auto pkill

[3]+ Terminated control pkill3

**[user@host ~]$ ps aux | grep test**

user 6281 0.0 0.1 222448 3012 pts/1 S 17:04 0:00 /bin/bash /home/user/bin/control test1

user 6285 0.0 0.1 222448 3128 pts/1 S 17:04 0:00 /bin/bash /home/user/bin/control test2

user 6292 0.0 0.1 222448 3064 pts/1 S 17:04 0:00 /bin/bash /home/user/bin/control test3

user 6318 0.0 0.0 221860 1080 pts/1 S+ 17:04 0:00 grep --color=auto test

**[user@host ~]$ pkill -U user**

**[user@host ~]$ ps aux | grep test**

user 6870 0.0 0.0 221860 1048 pts/0 S+ 17:07 0:00 grep --color=auto test

**[user@host ~]$**

## Logging Users Out Administratively

You may need to log other users off for any of a variety of reasons. To name a few of the many possibilities: the user committed a security violation; the user may have overused resources; the user may have an unresponsive system; or the user has improper access to materials. In these cases, you may need to administratively terminate their session using signals.

To log off a user, first identify the login session to be terminated. Use the **w** command to list user logins and current running processes. Note the TTY and FROM columns to determine the sessions to close.

All user login sessions are associated with a terminal device (TTY). If the device name is of the form pts/*N*, it is a pseudo-terminal associated with a graphical terminal window or remote login session. If it is of the form tty*N*, the user is on a system console, alternate console, or other directly connected terminal device.

**[user@host ~]$ w**

12:43:06 up 27 min, 5 users, load average: 0.03, 0.17, 0.66

USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT

root tty2 12:26 14:58 0.04s 0.04s -bash

bob tty3 12:28 14:42 0.02s 0.02s -bash

user pts/1 desk.example.com 12:41 2.00s 0.03s 0.03s w

**[user@host ~]$**

Discover how long a user has been on the system by viewing the session login time. For each session, CPU resources consumed by current jobs, including background tasks and child processes, are in the JCPU column. Current foreground process CPU consumption is in the PCPU column.

Processes and sessions can be individually or collectively signaled. To terminate all processes for one user, use the **pkill** command. Because the initial process in a login session (session leader) is designed to handle session termination requests and ignore unintended keyboard signals, killing all of a user's processes and login shells requires using the SIGKILL signal.

## Important

SIGKILL is commonly used too quickly by administrators.

Since the SIGKILL signal cannot be handled or ignored, it is always fatal. However, it forces termination without allowing the killed process to run self-cleanup routines. It is recommended to send SIGTERM first, then try SIGINT, and only if both fail retry with SIGKILL.

First identify the PID numbers to be killed using **pgrep**, which operates much like **pkill**, including using the same options, except that **pgrep** lists processes rather than killing them.

**[root@host ~]# pgrep -l -u bob**

6964 bash

6998 sleep

6999 sleep

7000 sleep

**[root@host ~]# pkill -SIGKILL -u bob**

**[root@host ~]# pgrep -l -u bob**

**[root@host ~]#**

When processes requiring attention are in the same login session, it may not be necessary to kill all of a user's processes. Determine the controlling terminal for the session using the **w** command, then kill only processes referencing the same terminal ID. Unless SIGKILL is specified, the session leader (here, the Bash login shell) successfully handles and survives the termination request, but all other session processes are terminated.

**[root@host ~]# pgrep -l -u bob**

7391 bash

7426 sleep

7427 sleep

7428 sleep

**[root@host ~]# w -h -u bob**

bob tty3 18:37 5:04 0.03s 0.03s -bash

**[root@host ~]# pkill -t tty3**

**[root@host ~]# pgrep -l -u bob**

7391 bash

**[root@host ~]# pkill -SIGKILL -t tty3**

**[root@host ~]# pgrep -l -u bob**

**[root@host ~]#**

The same selective process termination can be applied using parent and child process relationships. Use the **pstree** command to view a process tree for the system or a single user. Use the parent process's PID to kill all children they have created. This time, the parent Bash login shell survives because the signal is directed only at its child processes.

**[root@host ~]# pstree -p bob**

bash(8391)─┬─sleep(8425)

├─sleep(8426)

└─sleep(8427)

**[root@host ~]# pkill -P 8391**

**[root@host ~]# pgrep -l -u bob**

bash(8391)

**[root@host ~]# pkill -SIGKILL -P 8391**

**[root@host ~]# pgrep -l -u bob**

bash(8391)

**[root@host ~]#**

## References

**info libc signal** (GNU C Library Reference Manual)

* Section 24: Signal Handling

**info libc processes** (GNU C Library Reference Manual)

* Section 26: Processes

kill(1), killall(1), pgrep(1), pkill(1), pstree(1), signal(7), and w(1) man pages

# Guided Exercise: Killing Processes

In this exercise, you will use signals to manage and stop processes.

## Outcomes

You should be able to start and stop multiple shell processes.

Log in to workstation as student using student as the password.

On workstation, run the **lab processes-kill start** command. The command runs a start script that determines whether the host, servera, is reachable on the network.

**[student@workstation ~]$ lab processes-kill start**

1. On workstation, open two terminal windows side by side. In this section, these terminals are referred to as *left* and *right*. In each terminal, use the **ssh** command to log in to servera as the student user.

**[student@workstation ~]$ ssh student@servera**

*...output omitted...*

[student@servera ~]$

1. In the *left* window, create a new directory called /home/student/bin. In the new directory, create a shell script called killing. Make the script executable.
   1. Use the **mkdir** command to create a new directory called /home/student/bin.

**[student@servera ~]$ mkdir /home/student/bin**

* 1. Use the **vim** command to create a script called killing in the /home/student/bin directory. Press the **i** key to enter Vim interactive mode. Use the **:wq** command to save the file.

**[student@servera ~]$ vim /home/student/bin/killing**

#!/bin/bash

while true; do

echo -n "$@ " >> ~/killing\_outfile

sleep 5

done

Note

The killing script runs until terminated. It appends command line arguments to the ~/killing\_outfile once every 5 seconds.

* 1. Use the **chmod** command to make the killing file executable.

**[student@servera ~]$ chmod +x /home/student/bin/killing**

1. In the left terminal shell, use the **cd** command to change into the /home/student/bin/ directory. Start three **killing** processes with the arguments network, interface, and connection, respectively. Start three processes called network, interface, and connection. Use the ampersand (&) to start the processes in the background.

**[student@servera ~]$ cd /home/student/bin**

**[student@servera bin]$ killing network &**

[1] 3460

**[student@servera bin]$ killing interface &**

[2] 3482

**[student@servera bin]$ killing connection &**

[3] 3516

Your processes will have different PID numbers.

1. In the right terminal shell, use the **tail** command with the -f option to confirm that all three processes are appending to the /home/student/killing\_outfile file.

**[student@servera ~]$ tail -f ~/killing\_outfile**

network interface network connection interface network connection interface network

*...output omitted...*

1. In the left terminal shell, use the **jobs** command to list jobs.

**[student@servera bin]$ jobs**

[1] Running killing network &

[2]- Running killing interface &

[3]+ Running killing connection &

1. Use signals to suspend the network process. Confirm that the network process is Stopped. In the right terminal shell, confirm that the network process is no longer appending output to the ~/killing\_output.
   1. Use the **kill** with the -SIGSTOP option to stop the network process. Run the **jobs** to confirm it is stopped.

**[student@servera bin]$ kill -SIGSTOP %1**

[1]+ Stopped killing network

**[student@servera bin]$ jobs**

[1]+ Stopped killing network

[2] Running killing interface &

[3]- Running killing connection &

* 1. In the right terminal shell, look at the output from the **tail** command. Confirm that the word network is no longer being appended to the ~/killing\_outfile file.

*...output omitted...*

interface connection interface connection interface connection interface

1. In the left terminal shell, terminate the interface process using signals. Confirm that the interface process has disappeared. In the right terminal shell, confirm that interface process output is no longer appended to the ~/killing\_outfile file.
   1. Use the **kill** command with the -SIGTERM option to terminate the interface process. Run the **jobs** command to confirm that it has been terminated.

**[student@servera bin]$ kill -SIGTERM %2**

**[student@servera bin]$ jobs**

[1]+ Stopped killing network

[2] Terminated killing interface

[3]- Running killing connection &

* 1. In the right terminal shell, look at the output from the **tail** command. Confirm that the word interface is no longer being appended to the ~/killing\_outfile file.

*...output omitted...*

connection connection connection connection connection connection connection connection

1. In the left terminal shell, resume the network process using signals. Confirm that the network process is Running. In the right window, confirm that network process output is being appended to the ~/killing\_outfile file.
   1. Use the **kill** command with the -SIGCONT to resume the network process. Run the **jobs** command to confirm that the process is Running.

**[student@servera bin]$ kill -SIGCONT %1**

**[student@servera bin]$ jobs**

[1]+ Running killing network &

[3]- Running killing connection &

* 1. In the right terminal shell, look at the output from the **tail** command. Confirm that the word network is being appended to the ~/killing\_outfile file.

*...output omitted...*

network connection network connection network connection network connection network connection

1. In the left terminal shell, terminate the remaining two jobs. Confirm that no jobs remain and that output has stopped.
   1. Use the **kill** command with the -SIGTERM option to terminate the network process. Use the same command to terminate the connection process.

**[student@servera bin]$ kill -SIGTERM %1**

**[student@servera bin]$ kill -SIGTERM %3**

[1]+ Terminated killing network

**[student@servera bin]$ jobs**

[3]+ Terminated killing connection

1. In the left terminal shell, list tail processes running in all open terminal shells. Terminate running tail commands. Confirm that the process is no longer running.
   1. Use the **ps** command with the -ef option to list all running tail processes. Refine the search using the **grep** command.

**[student@servera bin]$ ps -ef | grep tail**

student 4581 31358 0 10:02 pts/0 00:00:00 tail -f killing\_outfile

student 4869 2252 0 10:33 pts/1 00:00:00 grep --color=auto tail

* 1. Use the **pkill** command with the -SIGTERM option to kill the tail process. Use the **ps** to confirm it is no longer present.

**[student@servera bin]$ pkill -SIGTERM tail**

**[student@servera bin]$ ps -ef | grep tail**

student 4874 2252 0 10:36 pts/1 00:00:00 grep --color=auto tail

* 1. In the right terminal shell, confirm that the **tail** command is no longer running.

*...output omitted...*

network connection network connection network connection Terminated

[student@servera ~]$

1. Exit from both terminal windows. Failure to exit from all sessions causes the finish script to fail.

**[student@servera bin]$ exit**

logout

Connection to servera closed.

[student@workstation ~]$

**[student@servera ~]$ exit**

logout

Connection to servera closed.

[student@workstation ~]$

## Finish

On workstation, run the **lab processes-kill finish** script to complete this exercise.

**[student@workstation ~]$ lab processes-kill finish**

This concludes the guided exercise.

# Monitoring Process Activity

## Objectives

After completing this section, you should be able to describe what load average is and determine processes responsible for high resource use on a server.

## Describing Load Average

*Load average* is a measurement provided by the Linux kernel that is a simple way to represent the perceived system load over time. It can be used as a rough gauge of how many system resource requests are pending, and to determine whether system load is increasing or decreasing over time.

Every five seconds, the kernel collects the current *load number*, based on the number of processes in runnable and uninterruptible states. This number is accumulated and reported as an exponential moving average over the most recent 1, 5, and 15 minutes.

## Understanding the Linux Load Average Calculation

The load average represents the perceived system load over a time period. Linux determines this by reporting how many processes are ready to run on a CPU, and how many processes are waiting for disk or network I/O to complete.

* The load number is a running average of the number of processes that are ready to run (in process state R) or are waiting for I/O to complete (in process state D).
* Some UNIX systems only consider CPU utilization or run queue length to indicate system load. Linux also includes disk or network utilization because that can have as significant an impact on system performance as CPU load. When experiencing high load averages with minimal CPU activity, examine disk and network activity.

Load average is a rough measurement of how many processes are currently waiting for a request to complete before they can do anything else. The request might be for CPU time to run the process. Alternatively, the request might be for a critical disk I/O operation to complete, and the process cannot be run on the CPU until the request completes, even if the CPU is idle. Either way, system load is impacted and the system appears to run more slowly because processes are waiting to run.

## Interpreting Displayed Load Average Values

The **uptime** command is one way to display the current load average. It prints the current time, how long the machine has been up, how many user sessions are running, and the current load average.

**[user@host ~]$ uptime**

15:29:03 up 14 min, 2 users, load average: 2.92, 4.48, 5.20

The three values for the load average represent the load over the last 1, 5, and 15 minutes. A quick glance indicates whether system load appears to be increasing or decreasing.

If the main contribution to load average is from processes waiting for the CPU, you can calculate the approximate *per CPU* load value to determine whether the system is experiencing significant waiting.

The **lscpu** command can help you determine how many CPUs a system has.

In the following example, the system is a dual-core single socket system with two hyperthreads per core. Roughly speaking, Linux will treat this as a four CPU system for scheduling purposes.

**[user@host ~]$ lscpu**

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 4

On-line CPU(s) list: 0-3

Thread(s) per core: 2

Core(s) per socket: 2

Socket(s): 1

NUMA node(s): 1

*...output omitted...*

For a moment, imagine that the only contribution to the load number is from processes that need CPU time. Then you can divide the displayed load average values by the number of logical CPUs in the system. A value below 1 indicates satisfactory resource utilization and minimal wait times. A value above 1 indicates resource saturation and some amount of processing delay.

# From **lscpu**, the system has four logical CPUs, so divide by 4:

# load average: 2.92, 4.48, 5.20

# divide by number of logical CPUs: 4 4 4

# ---- ---- ----

# per-CPU load average: 0.73 1.12 1.30

#

# This system's load average appears to be decreasing.

# With a load average of 2.92 on four CPUs, all CPUs were in use ~73% of the time.

# During the last 5 minutes, the system was overloaded by ~12%.

# During the last 15 minutes, the system was overloaded by ~30%.

An idle CPU queue has a load number of 0. Each process waiting for a CPU adds a count of 1 to the load number. If one process is running on a CPU, the load number is one, the resource (the CPU) is in use, but there are no requests waiting. If that process is running for a full minute, its contribution to the one-minute load average will be 1.

However, processes uninterruptibly sleeping for critical I/O due to a busy disk or network resource are also included in the count and increase the load average. While not an indication of CPU utilization, these processes are added to the queue count because they are waiting for resources and cannot run on a CPU until they get them. This is still system load due to resource limitations that is causing processes not to run.

Until resource saturation, a load average remains below 1, since tasks are seldom found waiting in queue. Load average only increases when resource saturation causes requests to remain queued and are counted by the load calculation routine. When resource utilization approaches 100%, each additional request starts experiencing service wait time.

A number of additional tools report load average, including **w** and **top**.

## Real-time Process Monitoring

The **top** program is a dynamic view of the system's processes, displaying a summary header followed by a process or thread list similar to **ps** information. Unlike the static **ps** output, **top** continuously refreshes at a configurable interval, and provides capabilities for column reordering, sorting, and highlighting. User configurations can be saved and made persistent.

Default output columns are recognizable from other resource tools:

* The process ID (PID).
* User name (USER) is the process owner.
* Virtual memory (VIRT) is all memory the process is using, including the resident set, shared libraries, and any mapped or swapped memory pages. (Labeled VSZ in the **ps** command.)
* Resident memory (RES) is the physical memory used by the process, including any resident shared objects. (Labeled RSS in the **ps** command.)
* Process state (S) displays as:
  + D = Uninterruptible Sleeping
  + R = Running or Runnable
  + S = Sleeping
  + T = Stopped or Traced
  + Z = Zombie
* CPU time (TIME) is the total processing time since the process started. May be toggled to include cumulative time of all previous children.
* The process command name (COMMAND).

## Table 8.3. Fundamental Keystrokes in top

| **Key** | **Purpose** |
| --- | --- |
| **?** *or* **h** | Help for interactive keystrokes. |
| **l**, **t**, **m** | Toggles for load, threads, and memory header lines. |
| **1** | Toggle showing individual CPUs or a summary for all CPUs in header. |
| **s** (1) | Change the refresh (screen) rate, in decimal seconds (e.g., 0.5, 1, 5). |
| **b** | Toggle reverse highlighting for Running processes; default is bold only. |
| **Shift**+**b** | Enables use of bold in display, in the header, and for *Running* processes. |
| **Shift**+**h** | Toggle threads; show process summary or individual threads. |
| **u**, **Shift**+**u** | Filter for any user name (effective, real). |
| **Shift**+**m** | Sorts process listing by memory usage, in descending order. |
| **Shift**+**p** | Sorts process listing by processor utilization, in descending order. |
| **k** (1) | Kill a process. When prompted, enter PID, then signal. |
| **r** (1) | Renice a process. When prompted, enter PID, then nice\_value. |
| **Shift**+**w** | Write (save) the current display configuration for use at the next **top** restart. |
| **q** | Quit. |
| **f** | Manage the columns by enabling or disabling fields. Also allows you to set the sort field for **top**. |
| Note: | (1) Not available if top started in secure mode. See **top**(1). |

## References

ps(1), top(1), uptime(1), and w(1) man pages

# Guided Exercise: Monitoring Process Activity

In this exercise, you will use the **top** command to dynamically examine running processes and control them.

## Outcomes

You should be able to manage processes in real time.

Log in to workstation as student using student as the password.

On workstation, run the **lab processes-monitor start** command. The command runs a start script that determines if the host, servera, is reachable on the network.

**[student@workstation ~]$ lab processes-monitor start**

1. On workstation open two terminal windows side by side. These terminals are referred to as *left* and *right*. On each terminal, use the **ssh** command to log in to servera as the student user.

**[student@workstation ~]$ ssh student@servera**

*...output omitted...*

**[student@servera ~]$**

1. In the *left* terminal shell, create a new directory called /home/student/bin. In the new directory create a shell script called monitor, which generates artificial CPU load. Ensure the script is executable.
   1. Use the **mkdir** command to create a new directory called /home/student/bin.

**[student@servera ~]$ mkdir /home/student/bin**

* 1. Create a script named monitor in the /home/student/bin directory with the following content:

#!/bin/bash

while true; do

var=1

while [[ var -lt 60000 ]]; do

var=$(($var+1))

done

sleep 1

done

The monitor script runs until terminated. It generates artificial CPU load by performing sixty thousand addition problems. It then sleeps for one second, resets the variable, and repeats.

* 1. Use the **chmod** command to make the monitor file executable.

**[student@servera ~]$ chmod a+x /home/student/bin/monitor**

1. In the right terminal shell, run the **top** utility. Size the window to be as tall as possible.

**[student@servera ~]$ top**

top - 12:13:03 up 11 days, 58 min, 3 users, load average: 0.00, 0.00, 0.00

Tasks: 113 total, 2 running, 111 sleeping, 0 stopped, 0 zombie

%Cpu(s): 0.2 us, 0.0 sy, 0.0 ni, 99.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

MiB Mem : 1829.4 total, 1377.3 free, 193.9 used, 258.2 buff/cache

MiB Swap: 1024.0 total, 1024.0 free, 0.0 used. 1476.1 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

5861 root 20 0 0 0 0 I 0.3 0.0 0:00.71 kworker/1:3-events

6068 student 20 0 273564 4300 3688 R 0.3 0.2 0:00.01 top

1 root 20 0 178680 13424 8924 S 0.0 0.7 0:04.03 systemd

2 root 20 0 0 0 0 S 0.0 0.0 0:00.03 kthreadd

3 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 rcu\_gp

*...output omitted...*

1. In the left terminal shell use the **lscpu** command to determine the number of logical CPUs on this virtual machine.

**[student@servera ~]$ lscpu**

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

**CPU(s): 2**

*...output omitted...*

1. In the left terminal shell, run a single instance of the **monitor** executable. Use the ampersand (&) to run the process in the background.

**[student@servera ~]$ monitor &**

[1] 6071

1. In the right terminal shell, observe the **top** display. Use the single keystrokes **l**, **t**, and **m** to toggle the load, threads, and memory header lines. After observing this behavior, ensure that all headers are displaying.
2. Note the process ID (PID) for **monitor**. View the CPU percentage for the process, which is expected to hover around 15% to 25%.

**[student@servera ~]$ top**

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

071 student 20 0 222448 2964 2716 S 18.7 0.2 0:27.35 monitor

*...output omitted...*

View the load averages. The one minute load average is currently less than a value of 1. The value observed may be affected by resource contention from another virtual machine or the virtual host.

top - 12:23:45 up 11 days, 1:09, 3 users, load average: 0.21, 0.14, 0.05

1. In the left terminal shell, run a second instance of **monitor**. Use the ampersand (&) to run the process in the background.

**[student@servera ~]$ monitor &**

[2] 6498

1. In the right terminal shell, note the process ID (PID) for the second monitor process. View the CPU percentage for the process, also expected to hover between 15% and 25%.

**[student@servera ~]$ top**

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

6071 student 20 0 222448 2964 2716 S 19.0 0.2 1:36.53 monitor

6498 student 20 0 222448 2996 2748 R 15.7 0.2 0:16.34 monitor

*...output omitted...*

View the one minute load average again, which is still less than 1. It is important to wait for at least one minute to allow the calculation to adjust to the new workload.

top - 12:27:39 up 11 days, 1:13, 3 users, load average: 0.36, 0.25, 0.11

1. In the left terminal shell, run a third instance of **monitor**. Use the ampersand (&) to run the process in the background.

**[student@servera ~]$ monitor &**

[3] 6881

1. In the right terminal shell, note the process ID (PID) for the third monitor process. View the CPU percentage for the process, again expected to hover between 15% and 25%.

**[student@servera ~]$ top**

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

6881 student 20 0 222448 3032 2784 S 18.6 0.2 0:11.48 monitor

6498 student 20 0 222448 2996 2748 S 15.6 0.2 0:47.86 monitor

6071 student 20 0 222448 2964 2716 S 18.1 0.2 2:07.86 monitor

To push the load average above 1, you must start more monitor processes. The classroom setup has 2 CPUs so only 3 processes are not enough to stress it. Start three more monitor processes. View the one minute load average again, which now is expected to be above 1. It is important to wait for at least one minute to allow the calculation to adjust to the new workload.

**[student@servera ~]$ monitor &**

[4] 10708

**[student@servera ~]$ monitor &**

[5] 11122

**[student@servera ~]$ monitor &**

[6] 11338

top - 12:42:32 up 11 days, 1:28, 3 users, load average: 1.23, 2.50, 1.54

1. When finished observing the load average values, terminate each of the monitor processes from within **top**.
   1. In the right terminal shell, press **k**. Observe the prompt below the headers and above the columns.

*...output omitted...*

PID to signal/kill [default pid = 11338]

* 1. The prompt has chosen the monitor processes at the top of the list. Press **Enter** to kill the process.

*...output omitted...*

Send pid 11338 signal [15/sigterm]

* 1. Press **Enter** again to confirm the SIGTERM signal 15.

Confirm that the selected process is no longer observed in **top**. If the PID still remains, repeat these terminating steps, substituting SIGKILL signal 9 when prompted.

6498 student 20 0 222448 2996 2748 R 22.9 0.2 5:31.47 monitor

6881 student 20 0 222448 3032 2784 R 21.3 0.2 4:54.47 monitor

11122 student 20 0 222448 2984 2736 R 15.3 0.2 2:32.48 monitor

6071 student 20 0 222448 2964 2716 S 15.0 0.2 6:50.90 monitor

10708 student 20 0 222448 3032 2784 S 14.6 0.2 2:53.46 monitor

1. Repeat the previous step for each remaining **monitor** instance. Confirm that no monitor processes remain in **top**.
2. In the right terminal shell, press **q** to exit **top**. Exit from servera on both terminal windows.

**[student@servera ~]$ exit**

logout

Connection to servera closed.

[student@workstation ~]$

**[student@servera ~]$ exit**

logout

Connection to servera closed.

[student@workstation ~]$

## Finish

On workstation, run the **lab processes-monitor finish** script to complete this exercise.

**[student@workstation ~]$ lab processes-monitor finish**

This concludes the guided exercise.

# Summary

In this chapter, you learned:

* A process is a running instance of an executable program. Processes are assigned a state, which can be running, sleeping, stopped, or zombie. The **ps** command is used to list processes.
* Each terminal is its own session and can have foreground process and independent background processes. The **jobs** command displays processes within a terminal session.
* A signal is a software interrupt that reports events to an executing program. The **kill**, **pkill**, and **killall** commands use signals to control processes.
* Load average is an estimate of how busy the system is. To display load average values, you can use the **top**, **uptime**, or **w** command.