Red Hat
Training and Certification

Student Workbook

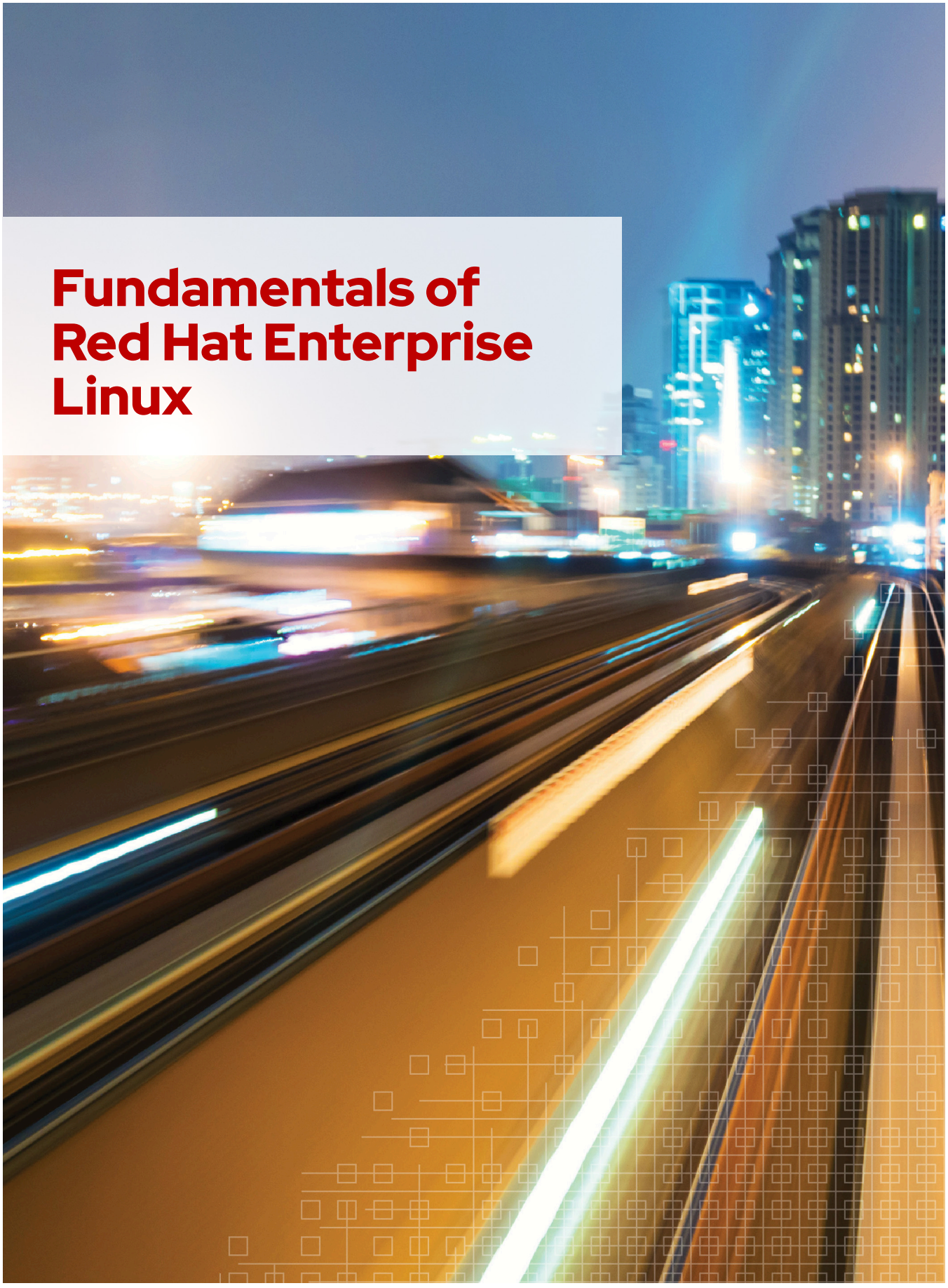
Red Hat Enterprise Linux 8 RH066

Fundamentals of Red Hat Enterprise Linux

Edition 1

RH066-RHEL8-en-1-cef4e50

Fundamentals of Red Hat Enterprise Linux



RH066-RHEL8-en-1-cef4e50

Red Hat Enterprise Linux 8 RH066
Fundamentals of Red Hat Enterprise Linux
Edition 1 cef4e50
Publication date 20200801

Authors: Susan Lauber, Philip Sweany, Rudolf Kastl, George Hacker
Editor: Steven Bonneville

Contributors: Rob Locke, Bowe Strickland, Scott McBrien, Wander Boessenkool, Forrest Taylor

Document Conventions

References

"References" describe where to find external documentation relevant to a subject.

Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.

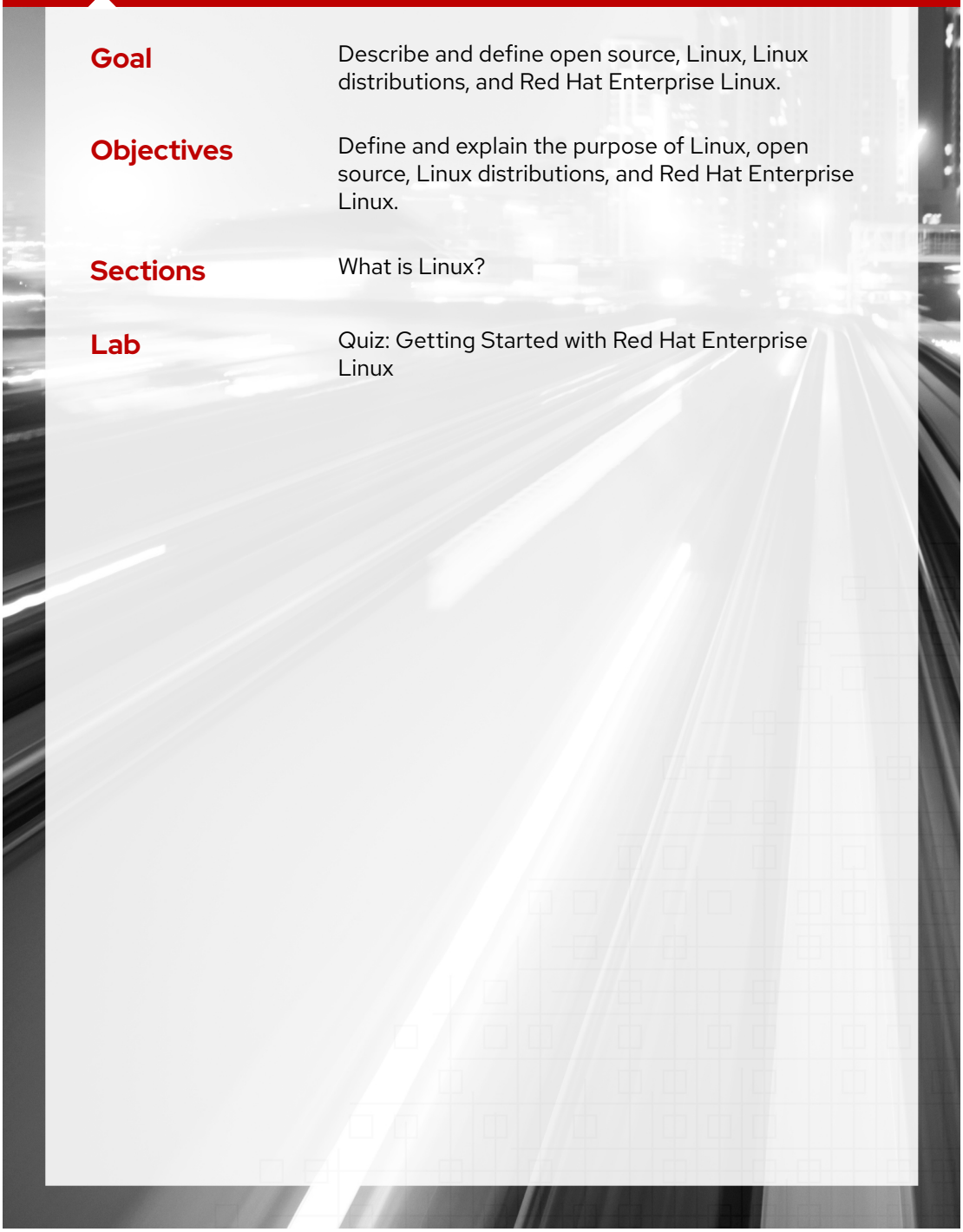
Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.

Chapter 1

Getting Started with Red Hat Enterprise Linux

Goal	Describe and define open source, Linux, Linux distributions, and Red Hat Enterprise Linux.
Objectives	Define and explain the purpose of Linux, open source, Linux distributions, and Red Hat Enterprise Linux.
Sections	What is Linux?
Lab	Quiz: Getting Started with Red Hat Enterprise Linux



What is Linux?

Objectives

After completing this section, students should be able to define and explain the purpose of Linux, open source, Linux distributions, and Red Hat Enterprise Linux.

Why Should You Learn about Linux?

The most important technology for IT professionals to understand may be Linux.

Linux is used everywhere. If you use the internet at all, you are already using Linux in your daily life. Perhaps the most obvious way in which you interact with Linux systems would be through browsing the worldwide web and using e-commerce sites to buy and sell products.

But Linux is used for much more than that. It manages point-of-sale systems and the world's stock markets. It is used to power smart TVs and in-flight entertainment systems. It powers most of the top 500 supercomputers in the world. Linux provides the foundational technologies powering the cloud revolution and the tools used to build the next generation of container-based microservices applications. Emerging software-based storage technologies and big-data solutions are being built on Linux.

In the modern datacenter, Linux and Microsoft Windows are the major players, and Linux is a growing segment in that space. Some of the many reasons to learn Linux:

- If you are a Windows person, you'll need to interoperate with Linux.
- If you are doing application development, it's likely your application or its runtime will be hosted on Linux.
- If you are working in the cloud, your cloud instances may be based on Linux, and your private or public cloud environment will probably be based on Linux.
- If you are working with mobile applications or the Internet of Things (IoT), the chances are great that the operating system of your device will be based on Linux.
- If you are looking for new opportunities in IT, Linux skills are in high demand.

What Makes Linux Great?

There are many different answers to the question "What makes Linux great?", but three of them are:

- Linux is *open source* software.

  Being open source does not just mean that you can see how the system works. You can also experiment with changes and share them freely for others to use. This means improvements are easier to make, enabling faster innovation.

- Easy access to *a powerful and scriptable command-line interface* (CLI).

  From the beginning, Linux has been built around the basic design philosophy that allows all administration to be done from the CLI. This enables easier automation, deployment, and provisioning, and simplifies both local and remote system administration. Unlike other operating

systems, these capabilities haven't had to be developed after the fact, and the assumptions of the system have always been to enable these important capabilities.

- Linux is modular and operating system *components can easily be replaced or removed*.

  Components of the system can be upgraded and updated as needed. A Linux system can be a general-purpose development workstation, or an extremely stripped-down software appliance.

What is Open Source Software?

Open source software is software with source code that anyone can use, study, modify, and share.

Source code is the set of human-readable instructions that are used to make a program. This may be interpreted as a script, or compiled into a binary executable which the computer runs directly. All source code is copyrighted from the moment it is created. Whether it can be distributed as source or binary executables is under the control of the copyright holder. Therefore, software is provided to users under a software license.

Some software has source code that only the person, team, or organization that created it can see, or change, or distribute. This is sometimes called "proprietary" or "closed source" software. Typically the license only allows the end user to run the program, and provides no access, or tightly limited access, to the source.

Open source software is different. When the copyright holder provides software under an open source license, they grant the user the right to run the program and *also* to view, modify, compile, and redistribute the source royalty-free to others.

Open source promotes collaboration, sharing, transparency and rapid innovation because it encourages people beyond the original developers to make modifications and improvements to the software and to share it with others.

Just because software is open source does not mean it is somehow not able to be used or provided commercially. Open source is a critical part of many organizations' commercial operations. Some open source licenses allow code to be reused in closed source products. Open source code can be sold, but the terms of true open source licenses generally allow the customer to re-distribute the source code. Most commonly, vendors like Red Hat can provide commercial help with deploying, supporting, and extending solutions based on open source products.

Open source has many benefits for the user:

- *Control*: See what the code does and change it to make it better.
- *Training*: Learn from real-world code and develop more effective applications.
- *Security*: Inspect sensitive code, fix with or without the original developers' help.
- *Stability*: Code can survive the loss of the original developer or distributor.

The bottom line is that we believe open source creates better software with a higher return on investment, and that we do things better when we do them together.

Types of Open Source Licenses

There is more than one way to be open source. The terms of the software license control how the source can be combined with other code or reused, and hundreds of different open source licenses exist. But in order to be open source, licenses must allow users to freely use, view, change, compile, and distribute the code.

There are two broad classes of open source license that are particularly important:

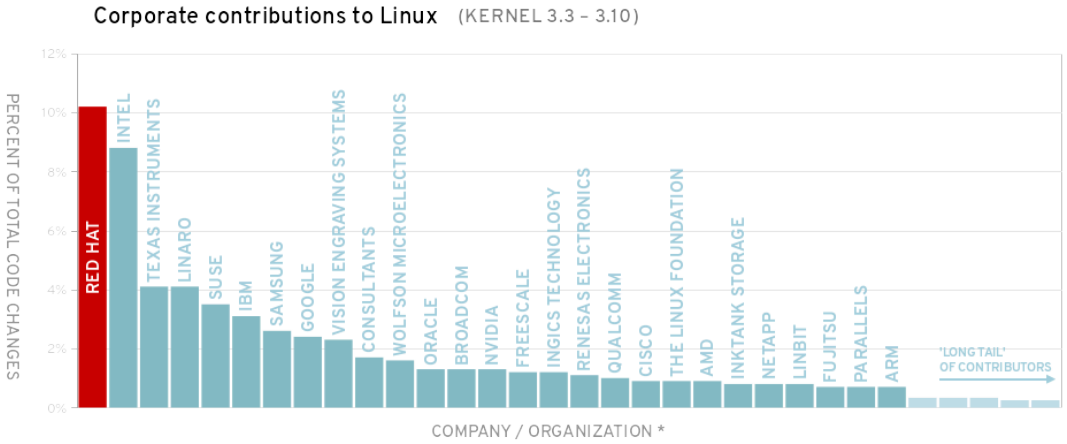
- *Copyleft* licenses that are designed to encourage keeping code open source.
- *Permissive* licenses that are designed to maximize code reusability.

Copyleft, or "share-alike" licenses, require that anyone who distributes the source code, with or without changes, must also pass along the freedom for others to also copy, change and distribute the code. The basic advantage of these licenses is that they help to keep existing code, and improvements to that code, open and add to the amount of open source code available. Common copyleft licenses include the GNU General Public License (GPL) and the Lesser GNU Public License (LGPL).

Permissive licenses are intended to maximize the reusability of source code. Users can use the source for any purpose as long as the copyright and license statements are preserved, including reusing that code under more restrictive or even proprietary licenses. This makes it very easy for this code to be reused, but at the risk of encouraging proprietary-only enhancements. Several commonly used permissive open source licenses include the MIT/X11 license, the Simplified BSD license, and the Apache Software License 2.0.

Who Develops Open Source Software?

It is a misconception to think that open source is developed solely by an "army of volunteers" or even an army of individuals plus Red Hat. In fact, open source development today is overwhelmingly professional. Many developers are paid by their organizations to work with open source projects to construct and contribute the enhancements they and their customers need.



* The developers who are known to be doing this work on their own, with no financial contribution happening from any company are not grouped together as 'None' and instead are considered part of the 'long tail,' as are contributors of academic or unknown sponsorship.

Source:
The Linux Foundation
Linux Kernel Development
September 2013
(page 9)

Volunteers and the academic community do play a very important role and can make vital contributions, especially in new technology areas. The combination of formal and informal development provides a highly dynamic and productive environment.

Who is Red Hat?

Red Hat is the world's leading provider of open source software solutions, using a community-powered approach to reliable and high-performing cloud, Linux, middleware, storage, and virtualization technologies. Red Hat's mission is to be the catalyst in communities of customers, contributors, and partners creating better technology the open source way.

Red Hat's role is to help customers connect with the open source community and our partners in order to effectively use open source software solutions. We actively participate in and support the

open source community, and many years of experience have convinced us of the importance of open source to the future of the IT industry.

Red Hat is most well-known for our participation in the Linux community and the Red Hat Enterprise Linux distribution. However, Red Hat is also very active in other open source communities including middleware projects centered on the JBoss developer community, virtualization solutions, cloud technologies such as OpenStack and OpenShift, and the Ceph and Gluster software-based storage projects, among others.

What is a Linux Distribution?

A *Linux distribution* is an installable operating system constructed from a Linux kernel and supporting user programs and libraries. A complete Linux operating system is not developed by a single organization, but by a collection of independent open source development communities working with individual software components. A distribution provides an easy way for users to install and manage a working Linux system.

In 1991, a young computer science student named Linus Torvalds developed a Unix-like kernel he named *Linux*, licensed as open source software under the GPL. The kernel is the core component of the operating system, which manages hardware, memory, and scheduling of running programs. This Linux kernel could then be supplemented with other open source software, such as the utilities and programs from the GNU Project, the graphical interface from MIT's X Window System, and many other open source components such as the Sendmail mail server or the Apache HTTPD web server, in order to build a complete open source Unix-like operating system.

However, one of the challenges for Linux users was to assemble all these pieces from many different sources. Very early in its history, Linux developers began working to provide a distribution of prebuilt and tested tools that users could download and use to quickly set up their Linux systems.

Many different Linux distributions exist, with differing goals and criteria for selecting and supporting the software provided by their distribution. But distributions generally have a number of common characteristics:

- A distribution consists of a Linux kernel and supporting user space programs.
- A distribution may be small and single-purpose or include thousands of open source programs.
- Some means of installing and updating the distribution and its components should be provided.
- The vendor should support that software, and ideally be participating directly in the community developing that software.

Red Hat Enterprise Linux is Red Hat's commercially-supported Linux distribution.

Red Hat Enterprise Linux

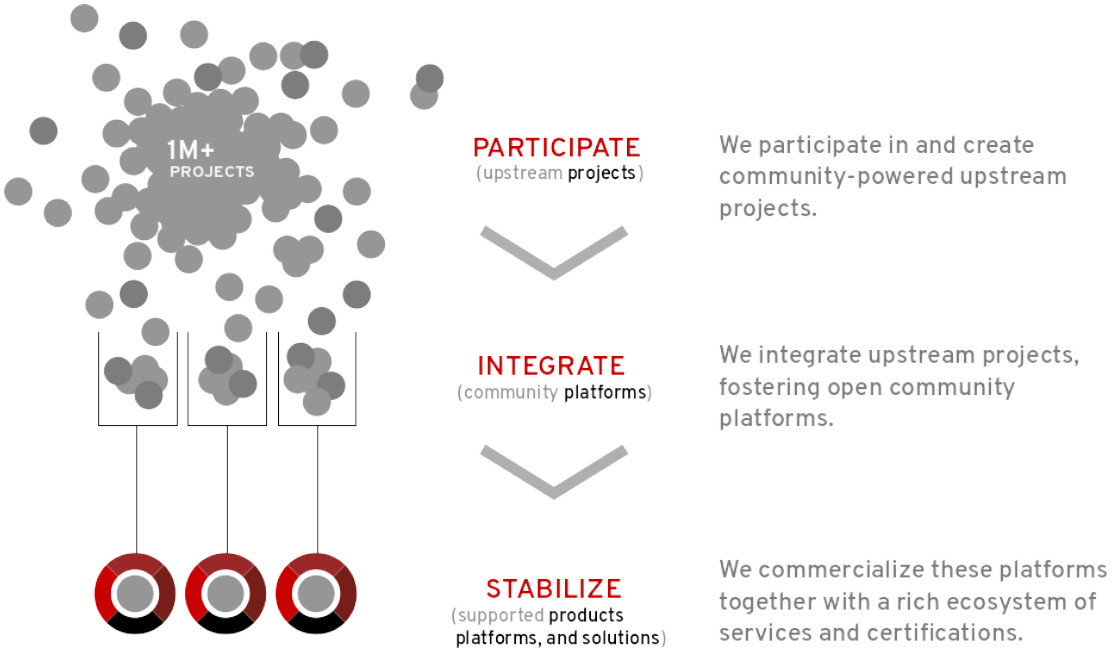
Development of Red Hat Enterprise Linux

Red Hat develops and integrates open source software into Red Hat Enterprise Linux through a multistage process.

- We *participate* in supporting individual open source projects. We contribute code, developer time, resources, and other support, often collaborating with developers from other Linux distributions. This helps to improve the general quality of software for all of us.
- We sponsor and *integrate* open source projects into a community-driven Linux distribution, *Fedora*. Fedora provides a free working environment that can serve as a development lab

and proving ground for features that may be incorporated into our commercially-supported products.

- We *stabilize* the software to ensure that it's ready for long term support and standardization, and integrate it into our enterprise-ready distribution, Red Hat Enterprise Linux.



Fedora

Fedora is a community project that produces and release a complete, free Linux-based operating system. Red Hat sponsors the community and works with community representatives to integrate the latest upstream software into a fast-moving and secure distribution. The Fedora project contributes everything back to the free and open source world, and anyone can participate.

However, Fedora is focused on innovation and excellence, not long-term stability. New major updates happen every six months, and they can bring significant changes. Fedora only supports releases for about a year (two major updates). This can make Fedora less suitable for enterprise use.

Red Hat Enterprise Linux

Red Hat Enterprise Linux is Red Hat's enterprise-ready, commercially supported Linux distribution. The leading platform for open source computing, it's not just a collection of mature open source projects. Red Hat Enterprise Linux is extensively tested, with a large supporting ecosystem of partners, hardware and software certifications, consulting services, training, and multi-year support and maintenance guarantees.

Red Hat bases its major releases of Red Hat Enterprise Linux on Fedora. But after that, Red Hat can pick and choose which packages to include, make further enhancements (contributed back to the upstream projects and Fedora), and make configuration decisions that serve the needs of customers. Red Hat helps vendors and customers engage with the open source community, to work with upstream development to develop solutions and fix issues.

Red Hat Enterprise Linux is provided through a subscription-based model. Since this is open source software, this is not a license fee. Instead, it pays for support, maintenance, updates, security patches, access to the Knowledgebase at Red Hat Customer Portal (http://

access.redhat.com/), certifications, and so on. The customer is paying for long-term support and expertise, commitment, and assistance when they need it.

When major updates become available, customers can move to them at their convenience without paying more. This can simplify management of both the economic and practical aspects of system updates.

Trying out Red Hat Enterprise Linux

There are a number of different ways to try Red Hat Enterprise Linux. One is to download an evaluation copy from our website at https://access.redhat.com/products/red-hat-enterprise-linux/evaluation. A number of links to supplementary information are available from that page.

Red Hat also makes available free subscriptions of a number of our products for development purposes through the Red Hat Developer Program at https://developer.redhat.com. These subscriptions allow developers to easily develop, prototype, test, and demonstrate their software on the same enterprise products that it will eventually be deployed with.

Another approach is to deploy an instance of Red Hat Enterprise Linux made available through a cloud provider. For example, in this course the hands-on labs are designed to use an official Red Hat Enterprise Linux image provided through a major cloud provider's image catalog.

For more information, please visit the Red Hat Enterprise Linux "Get Started" page, referenced at the end of this section.

References

Red Hat Enterprise Linux: Get Started
https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux/get-started

The Open Source Way
https://opensource.com/open-source-way

Quiz

Getting Started with Red Hat Enterprise Linux

Choose the correct answer to the following questions:

1. Which of the following are benefits of open source software for the user? (Choose two.)

a. The code can survive the loss of the original developer or distributor.
b. The sensitive portions of the code are protected and only available to the original developer.
c. You can learn from real-world code and develop more effective applications.
d. The code remains open as long as it is in a public repository but the license may change when included with closed source software.

2. How does Red Hat develop their products for the future and interact with the community? (Choose two.)

a. Sponsor and integrate open source projects into the community driven Fedora project.
b. Develop specific integration tools only available in Red Hat distributions.
c. Participate in upstream projects.
d. Repackage and re-license community products.

3. Which statements describe benefits of Linux? (Choose two.)

a. Linux is developed entirely by volunteers making is a low cost operating system.
b. Linux is modular and can be configured as a full graphical desktop or a small appliance.
c. Linux is locked in a known state for a minimum of one year for each release making it easier to develop custom software.
d. Linux includes a powerful and scriptable command line interface enabling easier automation and provisioning.

Solution

Getting Started with Red Hat Enterprise Linux

Choose the correct answer to the following questions:

1. Which of the following are benefits of open source software for the user? (Choose two.)

a. The code can survive the loss of the original developer or distributor.

b. The sensitive portions of the code are protected and only available to the original developer.

c. You can learn from real-world code and develop more effective applications.

d. The code remains open as long as it is in a public repository but the license may change when included with closed source software.

2. How does Red Hat develop their products for the future and interact with the community? (Choose two.)

a. Sponsor and integrate open source projects into the community driven Fedora project.

b. Develop specific integration tools only available in Red Hat distributions.

c. Participate in upstream projects.

d. Repackage and re-license community products.

3. Which statements describe benefits of Linux? (Choose two.)

a. Linux is developed entirely by volunteers making is a low cost operating system.

b. Linux is modular and can be configured as a full graphical desktop or a small appliance.

c. Linux is locked in a known state for a minimum of one year for each release making it easier to develop custom software.

d. Linux includes a powerful and scriptable command line interface enabling easier automation and provisioning.

Chapter 2

Accessing the Command Line

Goal

To log into a Linux system and run simple commands using the shell.

Objectives

- Use Bash shell syntax to enter commands at a Linux console.
- Use Bash features to run commands from a shell prompt using fewer keystrokes.

Sections

- Accessing the Command Line (and Quiz)
- Executing Commands Using the Bash Shell (and Quiz)

Lab

Accessing the Command Line

Accessing the Command Line

Objectives

After completing this section, you should be able to log in to a Linux system and run simple commands using the shell.

Introduction to the Bash Shell

A *command line* is a text-based interface which can be used to input instructions to a computer system. The Linux command line is provided by a program called the *shell*. Various options for the shell program have been developed over the years, and different users can be configured to use different shells. Most users, however, stick with the current default.

The default shell for users in Red Hat Enterprise Linux is the GNU Bourne-Again Shell (**bash**). Bash is an improved version of one of the most successful shells used on UNIX-like systems, the Bourne Shell (**sh**).

When a shell is used interactively, it displays a string when it is waiting for a command from the user. This is called the *shell prompt*. When a regular user starts a shell, the default prompt ends with a **$** character, as shown below.

```
[user@host ~]$
```

The **$** character is replaced by a **#** character if the shell is running as the superuser, **root**. This makes it more obvious that it is a superuser shell, which helps to avoid accidents and mistakes which can affect the whole system. The superuser shell prompt is shown below.

```
[root@host ~]#
```

Using **bash** to execute commands can be powerful. The **bash** shell provides a scripting language that can support automation of tasks. The shell has additional capabilities that can simplify or make possible operations that are hard to accomplish efficiently with graphical tools.

Note

The **bash** shell is similar in concept to the command-line interpreter found in recent versions of Microsoft Windows, **cmd.exe**, although **bash** has a more sophisticated scripting language. It is also similar to Windows PowerShell in Windows 7 and Windows Server 2008 R2 and later. Administrators using the Apple Mac who use the Terminal utility may be pleased to note that **bash** is the default shell in macOS.

Shell Basics

Commands entered at the shell prompt have three basic parts:

- *Command* to run
- *Options* to adjust the behavior of the command

- *Arguments*, which are typically targets of the command

The *command* is the name of the program to run. It may be followed by one or more *options*, which adjust the behavior of the command or what it will do. Options normally start with one or two dashes (**`-a`** or **`--all`**, for example) to distinguish them from arguments. Commands may also be followed by one or more *arguments*, which often indicate a target that the command should operate upon.

For example, the command **`usermod -L user01`** has a command (**`usermod`**), an option (**`-L`**), and an argument (**`user01`**). The effect of this command is to lock the password of the **`user01`** user account.

Many commands have a **`--help`** option that displays a usage message and the list of available options.

Logging in to a Local Computer

To run the shell, you need to log in to the computer on a *terminal*. A terminal is a text-based interface used to enter commands into and print output from a computer system. There are several ways to do this.

The computer might have a hardware keyboard and display for input and output directly connected to it. This is the Linux machine's *physical console*. The physical console supports multiple *virtual consoles*, which can run separate terminals. Each virtual console supports an independent login session. You can switch between them by pressing **`Ctrl`**+**`Alt`** and a function key (**`F1`** through **`F6`**) at the same time. Most of these virtual consoles run a terminal providing a text login prompt, and if you enter your username and password correctly, you will log in and get a shell prompt.

The computer might provide a graphical login prompt on one of the virtual consoles. You can use this to log in to a *graphical environment*. The graphical environment also runs on a virtual console. To get a shell prompt you must start a terminal program in the graphical environment. The shell prompt is provided in an application window of your graphical terminal program.

Note

Many system administrators choose not to run a graphical environment on their servers. This allows resources which would be used by the graphical environment to be used by the server's services instead.

In Red Hat Enterprise Linux 8, if the graphical environment is available, the login screen will run on the first virtual console, called **`tty1`**. Five additional text login prompts are available on virtual consoles two through six.

If you log in using the graphical login screen, your graphical environment will start on the first virtual console that is not currently being used by a login session. Normally, your graphical session will replace the login prompt on the second virtual console (**`tty2`**). However, if that console is in use by an active text login session (not just a login prompt), the next free virtual console is used instead.

The graphical login screen continues to run on the first virtual console (**`tty1`**). If you are already logged in to a graphical session, and log in as another user on the graphical login screen or use the **Switch User** menu item to switch users in the graphical environment without logging out, another graphical environment will be started for that user on the next free virtual console.

When you log out of a graphical environment, it will exit and the physical console will automatically switch back to the graphical login screen on the first virtual console.

Note

In Red Hat Enterprise Linux 6 and 7, the graphical login screen runs on the first virtual console, but when you log in your initial graphical environment *replaces* the login screen on the first virtual console instead of starting on a new virtual console.

In Red Hat Enterprise Linux 5 and earlier, the first six virtual consoles always provided text login prompts. If the graphical environment is running, it is on virtual console *seven* (accessed through **Ctrl**+**Alt**+**F7**).

A *headless server* does not have a keyboard and display permanently connected to it. A data center may be filled with many racks of headless servers, and not providing each with a keyboard and display saves space and expense. To allow administrators to log in, a headless server might have a login prompt provided by its *serial console*, running on a serial port which is connected to a networked console server for remote access to the serial console.

The serial console would normally be used to fix the server if its own network card became misconfigured and logging in over its own network connection became impossible. Most of the time, however, headless servers are accessed by other means over the network.

Logging in over the Network

Linux users and administrators often need to get shell access to a remote system by connecting to it over the network. In a modern computing environment, many headless servers are actually virtual machines or are running as public or private cloud instances. These systems are not physical and do not have real hardware consoles. They might not even provide access to their (simulated) physical console or serial console.

In Linux, the most common way to get a shell prompt on a remote system is to use Secure Shell (SSH). Most Linux systems (including Red Hat Enterprise Linux) and macOS provide the OpenSSH command-line program **ssh** for this purpose.

In this example, a user with a shell prompt on the machine **host** uses **ssh** to log in to the remote Linux system **remotehost** as the user **remoteuser**:

```
[user@host ~]$ ssh remoteuser@remotehost
remoteuser@remotehost's password: password
[remoteuser@remotehost ~]$
```

The **ssh** command encrypts the connection to secure the communication against eavesdropping or hijacking of the passwords and content.

Some systems (such as new cloud instances) do not allow users to use a password to log in with **ssh** for tighter security. An alternative way to authenticate to a remote machine without entering a password is through *public key authentication*.

With this authentication method, users have a special identity file containing a *private key*, which is equivalent to a password, and which they keep secret. Their account on the server is configured with a matching *public key*, which does not have to be secret. When logging in, users can configure **ssh** to provide the private key and if their matching public key is installed in that account on that remote server, it will log them in without asking for a password.

In the next example, a user with a shell prompt on the machine **host** logs in to **remotehost** as **remoteuser** using **ssh**, using public key authentication. The **-i** option is used to specify the user's private key file, which is **mylab.pem**. The matching public key is already set up as an authorized key in the **remoteuser** account.

```
[user@host ~]$ ssh -i mylab.pem remoteuser@remotehost
[remoteuser@remotehost ~]$
```

For this to work, the private key file must be readable only by the user that owns the file. In the preceding example, where the private key is in the **mylab.pem** file, the command **chmod 600 mylab.pem** could be used to ensure this. How to set file permissions is discussed in more detail in a later chapter.

Users might also have private keys configured that are tried automatically, but that discussion is beyond the scope of this section. The References at the end of this section contain links to more information on this topic.

Note

The first time you log in to a new machine, you will be prompted with a warning from **ssh** that it cannot establish the authenticity of the host:

```
[user@host ~]$ ssh -i mylab.pem remoteuser@remotehost
The authenticity of host 'remotehost (192.0.2.42)' can't be established.
ECDSA key fingerprint is 47:bf:82:cd:fa:68:06:ee:d8:83:03:1a:bb:29:14:a3.
Are you sure you want to continue connecting (yes/no)? yes
[remoteuser@remotehost ~]$
```

Each time you connect to a remote host with **ssh**, the remote host sends **ssh** its *host key* to authenticate itself and to help set up encrypted communication. The **ssh** command compares that against a list of saved host keys to make sure it has not changed. If the host key has changed, this might indicate that someone is trying to pretend to be that host to hijack the connection which is also known as man-in-the-middle attack. In SSH, host keys protect against man-in-the-middle attacks, these host keys are unique for each server, and they need to be changed periodically and whenever a compromise is suspected.

You will get this warning if your local machine does not have a host key saved for the remote host. If you enter **yes**, the host key that the remote host sent will be accepted and saved for future reference. Login will continue, and you should not see this message again when connecting to this host. If you enter **no**, the host key will be rejected and the connection closed.

If the local machine does have a host key saved and it does not match the one actually sent by the remote host, the connection will automatically be closed with a warning.

Logging Out

When you are finished using the shell and want to quit, you can choose one of several ways to end the session. You can enter the **exit** command to terminate the current shell session. Alternatively, finish a session by pressing **Ctrl**+**D**.

The following is an example of a user logging out of an SSH session:

```
[remoteuser@remotehost ~]$ exit
logout
Connection to remotehost closed.
[user@host ~]$
```

References

intro(1), **bash**(1), **console**(4), **pts**(4), **ssh**(1), and **ssh-keygen**(1) man pages

*Note: Some details of the **console**(4) man page, involving **init**(8) and **inittab**(5), are outdated.*

For more information on OpenSSH and public key authentication, refer to the *Using secure communications between two systems with OpenSSH* chapter in the *Red Hat Enterprise Linux 8 Securing networks* guide at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/securing_networks/index#using-secure-communications-between-two-systems-with-openssh_securing-networks

Note

Instructions on how to read **man** pages and other online help documentation is included at the end of the next section.

Quiz

Local Console Access Terms

Choose the correct answer to the following questions:

1. Which term describes the interpreter that executes commands typed as strings?

a. Command
b. Console
c. Shell
d. Terminal

2. Which term describes the visual cue that indicates an interactive shell is waiting for the user to type a command?

a. Argument
b. Command
c. Option
d. Prompt

3. Which term describes the name of a program to run?

a. Argument
b. Command
c. Option
d. Prompt

4. Which term describes the part of the command line that adjusts the behavior of a command?

a. Argument
b. Command
c. Option
d. Prompt

5. Which term describes the part of the command line that specifies the target that the command should operate on?

a. Argument
b. Command
c. Option
d. Prompt

6. **Which term describes the hardware display and keyboard used to interact with a system?**
   a. Physical Console
   b. Virtual Console
   c. Shell
   d. Terminal

7. **Which term describes one of multiple logical consoles that can each support an independent login session?**
   a. Physical Console
   b. Virtual Console
   c. Shell
   d. Terminal

8. **Which term describes an interface that provides a display for output and a keyboard for input to a shell session?**
   a. Console
   b. Virtual Console
   c. Shell
   d. Terminal

Solution

Local Console Access Terms

Choose the correct answer to the following questions:

1. Which term describes the interpreter that executes commands typed as strings?

a. Command
b. Console
c. Shell
d. Terminal

2. Which term describes the visual cue that indicates an interactive shell is waiting for the user to type a command?

a. Argument
b. Command
c. Option
d. Prompt

3. Which term describes the name of a program to run?

a. Argument
b. Command
c. Option
d. Prompt

4. Which term describes the part of the command line that adjusts the behavior of a command?

a. Argument
b. Command
c. Option
d. Prompt

5. Which term describes the part of the command line that specifies the target that the command should operate on?

a. Argument
b. Command
c. Option
d. Prompt

6. Which term describes the hardware display and keyboard used to interact with a system?

a. Physical Console
b. Virtual Console
c. Shell
d. Terminal

7. Which term describes one of multiple logical consoles that can each support an independent login session?

a. Physical Console
b. Virtual Console
c. Shell
d. Terminal

8. Which term describes an interface that provides a display for output and a keyboard for input to a shell session?

a. Console
b. Virtual Console
c. Shell
d. Terminal

Executing Commands Using the Bash Shell

Objectives

After completing this section, you should be able to save time running commands from a shell prompt using Bash shortcuts.

Basic Command Syntax

The GNU Bourne-Again Shell (**bash**) is a program that interprets commands typed in by the user. Each string typed into the shell can have up to three parts: the command, options (which usually begin with a - or --), and arguments. Each word typed into the shell is separated from each other with spaces. Commands are the names of programs that are installed on the system. Each command has its own options and arguments.

When you are ready to execute a command, press the **Enter** key. Type each command on a separate line. The command output is displayed before the next shell prompt appears.

```
[user@host]$ whoami
user
[user@host]$
```

If you want to type more than one command on a single line, use the semicolon (**;**) as a command separator. A semicolon is a member of a class of characters called *metacharacters* that have special meanings for **bash**. In this case the output of both commands will be displayed before the next shell prompt appears.

The following example shows how to combine two commands (**command1** and **command2**) on the command line.

```
[user@host]$ command1;command2
```

Examples of Simple Commands

The **date** command displays the current date and time. It can also be used by the superuser to set the system clock. An argument that begins with a plus sign (+) specifies a format string for the date command.

```
[user@host ~]$ date
Sat Jan  26 08:13:50 IST 2019
[user@host ~]$ date +%R
08:13
[user@host ~]$ date +%x
01/26/2019
```

The **passwd** command changes a user's own password. The original password for the account must be specified before a change is allowed. By default, **passwd** is configured to require a strong password, consisting of lowercase letters, uppercase letters, numbers, and symbols, and is not

based on a dictionary word. The superuser can use the **passwd** command to change other users' passwords.

```
[user@host ~]$ passwd
Changing password for user user.
Current password: old_password
New password: new_password
Retype new password: new_password
passwd: all authentication tokens updated successfully.
```

Linux does not require file name extensions to classify files by type. The **file** command scans the beginning of a file's contents and displays what type it is. The files to be classified are passed as arguments to the command.

```
[user@host ~]$ file /etc/passwd
/etc/passwd: ASCII text
[user@host ~]$ file /bin/passwd
/bin/passwd: setuid ELF 64-bit LSB shared object, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 3.2.0, BuildID[sha1]=a3637110e27e9a48dced9f38b4ae43388d32d0e4,
 stripped
[user@host ~]$ file /home
/home: directory
```

Viewing the Contents of Files

One of the most simple and frequently used commands in Linux is **cat**. The **cat** command allows you to create single or multiple files, view the contents of files, concatenate the contents from multiple files, and redirect contents of the file to a terminal or files.

The example shows how to view the contents of the **/etc/passwd** file.

```
[user@host ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
...output omitted...
```

Use the following command to display the contents of multiple files.

```
[user@host ~]$ cat file1 file2
Hello World!!
Introduction to Linux commands.
```

Some files are very long and can take up more room to display than that provided by the terminal. The **cat** command does not display the contents of a file as pages. The **less** command displays one page of a file at a time and lets you scroll at your leisure.

The **less** command allows you to page forward and backward through files that are longer than can fit on one terminal window. Use the **UpArrow** key and the **DownArrow** key to scroll up and down. Press **q** to exit the command.

The **head** and **tail** commands display the beginning and end of a file, respectively. By default these commands display 10 lines of the file, but they both have a **-n** option that allows a different number of lines to be specified. The file to display is passed as an argument to these commands.

```
[user@host ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
[user@host ~]$ tail -n 3 /etc/passwd
gdm:x:42:42::/var/lib/gdm:/sbin/nologin
gnome-initial-setup:x:977:977::/run/gnome-initial-setup/:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
```

The **wc** command counts lines, words, and characters in a file. It takes a **-l**, **-w**, or **-c** option to display only the number of lines, words, or characters, respectively.

```
[user@host ~]$ wc /etc/passwd
  45  102 2480 /etc/passwd
[user@host ~]$ wc -l /etc/passwd ; wc -l /etc/group
45 /etc/passwd
70 /etc/group
[user@host ~]$ wc -c /etc/group /etc/hosts
 966 /etc/group
 516 /etc/hosts
1482 total
```

Tab Completion

Tab completion allows a user to quickly complete commands or file names after they have typed enough at the prompt to make it unique. If the characters typed are not unique, pressing the **Tab** key twice displays all commands that begin with the characters already typed.

```
[user@host ~]$ pas(1)Tab+Tab
passwd       paste        pasuspender
[user@host ~]$ pass(2)Tab
[user@host ~]$ passwd
Changing password for user user.
Current password:
```

1. Press **Tab** twice.
2. Press **Tab** once.

Tab completion can be used to complete file names when typing them as arguments to commands. When **Tab** is pressed, it completes as much of the file name as possible. Pressing **Tab** a second time causes the shell to list all of the files that are matched by the current pattern. Type additional characters until the name is unique, then use tab completion to complete the command.

```
[user@host ~]$ ls /etc/pas(1)Tab
[user@host ~]$ ls /etc/passwd(2)Tab
passwd   passwd-
```

(1)(2) Press **Tab** once.

Arguments and options can be matched with tab completion for many commands. The **useradd** command is used by the superuser, **root**, to create additional users on the system. It has many options that can be used to control how that command behaves. Tab completion following a partial option can be used to complete the option without a lot of typing.

```
[root@host ~]# useradd --(1)Tab+Tab
--base-dir        --groups          --no-log-init     --shell
--comment         --help            --non-unique      --skel
--create-home     --home-dir        --no-user-group   --system
--defaults        --inactive        --password        --uid
--expiredate      --key             --root            --user-group
--gid             --no-create-home  --selinux-user
[root@host ~]# useradd --
```

(1) Press **Tab** twice.

Continuing a Long Command on Another Line

Commands with many options and arguments can quickly become long and are automatically wrapped by the command window when the the cursor reaches the right margin. Instead, to make command readability easier, you can type a long command using more than one line.

To do this, you will use a backslash character (\), referred to as the *escape* character, to ignore the meaning of the character immediately following the backslash. You have learned that entering a newline character, by pressing the **Enter** key, tells the shell that command entry is complete and to execute the command. By escaping the newline character, the shell is told to move to a new command line without performing command execution. The shell acknowledges the request by displaying a continuation prompt, referred to as the *secondary prompt*, using the greater-than character (>) by default, on an empty new line. Commands may be continued over many lines.

```
[user@host]$ head -n 3 \
> /usr/share/dict/words \
> /usr/share/dict/linux.words
==> /usr/share/dict/words <==
1080
10-point
10th

==> /usr/share/dict/linux.words <==
1080
10-point
10th
[user@host ~]$
```

Important

The previous screen example displays how a continued command appears to a typical user. However, protraying this realism in learning materials, such as this coursebook, commonly causes confusion. New learners might mistakenly insert the extra greater-than character as part of the typed command. The shell interprets a typed greater-than character as *process redirection*, which the user did not intend. Process redirection is discussed in an upcoming chapter.

To avoid this confusion, this coursebook will not show secondary prompts in screen outputs. A user still sees the secondary prompt in their shell window, but the course material intentionally displays only characters to be typed, as demonstrated in the example below. Compare with the previous screen example.

```
[user@host]$ head -n 3 \
/usr/share/dict/words \
/usr/share/dict/linux.words
==> /usr/share/dict/words <==
1080
10-point
10th

==> /usr/share/dict/linux.words <==
1080
10-point
10th
[user@host ~]$
```

Command History

The **`history`** command displays a list of previously executed commands prefixed with a command number.

The exclamation point character (!) is a metacharacter that is used to expand previous commands without having to retype them. The **`!`*`number`*** command expands to the command matching the number specified. The **`!`*`string`*** command expands to the most recent command that begins with the string specified.

```
[user@host ~]$ history
   ...output omitted...
   23  clear
   24  who
   25  pwd
   26  ls /etc
   27  uptime
   28  ls -l
   29  date
   30  history
[user@host ~]$ !ls
ls -l
total 0
drwxr-xr-x. 2 user user 6 Mar 29 21:16 Desktop
...output omitted...
```

```
[user@host ~]$ !26
ls /etc
abrt                    hosts                   pulse
adjtime                 hosts.allow             purple
aliases                 hosts.deny              qemu-ga
...output omitted...
```

The arrow keys can be used to navigate through previous commands in the shell's history. **UpArrow** edits the previous command in the history list. **DownArrow** edits the next command in the history list. **LeftArrow** and **RightArrow** move the cursor left and right in the current command from the history list, so that you can edit it before running it.

You can use either the **Esc**+**.** or **Alt**+**.** key combination to insert the last word of the previous command at the cursor's current location. Repeated use of the key combination will replace that text with the last word of even earlier commands in the history. The **Alt**+**.** key combination is particularly convenient because you can hold down **Alt** and press **.** repeatedly to easily go through earlier and earlier commands.

Editing the Command Line

When used interactively, **bash** has a command-line editing feature. This allows the user to use text editor commands to move around within and modify the current command being typed. Using the arrow keys to move within the current command and to step through the command history was introduced earlier in this session. More powerful editing commands are introduced in the following table.

Useful Command-line Editing Shortcuts

Shortcut	Description
Ctrl+**A**	Jump to the beginning of the command line.
Ctrl+**E**	Jump to the end of the command line.
Ctrl+**U**	Clear from the cursor to the beginning of the command line.
Ctrl+**K**	Clear from the cursor to the end of the command line.
Ctrl+**LeftArrow**	Jump to the beginning of the previous word on the command line.
Ctrl+**RightArrow**	Jump to the end of the next word on the command line.
Ctrl+**R**	Search the history list of commands for a pattern.

There are several other command-line editing commands available, but these are the most useful commands for new users. The other commands can be found in the **bash**(1) man page.

References

bash(1), **date**(1), **file**(1), **cat**(1), **more**(1), **less**(1), **head**(1), **passwd**(1), **tail**(1), and **wc**(1) man pages

Quiz

Bash Commands and Keyboard Shortcuts

Choose the correct answers to the following questions:

1. **Which Bash shortcut or command jumps to the beginning of the previous word on the command line?**
   a. Pressing **Ctrl**+**LeftArrow**
   b. Pressing **Ctrl**+**K**
   c. Pressing **Ctrl**+**A**
   d. **!*string***
   e. **!*number***

2. **Which Bash shortcut or command separates commands on the same line?**
   a. Pressing **Tab**
   b. **history**
   c. **;**
   d. **!*string***
   e. Pressing **Esc**+**.**

3. **Which Bash shortcut or command is used to clear characters from the cursor to the end of the command line?**
   a. Pressing **Ctrl**+**LeftArrow**
   b. Pressing **Ctrl**+**K**
   c. Pressing **Ctrl**+**A**
   d. **;**
   e. Pressing **Esc**+**.**

4. **Which Bash shortcut or command is used to re-execute a recent command by matching the command name?**
   a. Pressing **Tab**
   b. **!*number***
   c. **!*string***
   d. **history**
   e. Pressing **Esc**+**.**

5. **Which Bash shortcut or command is used to complete commands, file names, and options?**
   a. `;`
   b. `!number`
   c. `history`
   d. Pressing `Tab`
   e. Pressing `Esc+.`

6. **Which Bash shortcut or command re-executes a specific command in the history list?**
   a. Pressing `Tab`
   b. `!number`
   c. `!string`
   d. `history`
   e. Pressing `Esc+.`

7. **Which Bash shortcut or command jumps to the beginning of the command line?**
   a. `!number`
   b. `!string`
   c. Pressing `Ctrl+LeftArrow`
   d. Pressing `Ctrl+K`
   e. Pressing `Ctrl+A`

8. **Which Bash shortcut or command displays the list of previous commands?**
   a. Pressing `Tab`
   b. `!string`
   c. `!number`
   d. `history`
   e. Pressing `Esc+.`

9. **Which Bash shortcut or command copies the last argument of previous commands?**
   a. Pressing `Ctrl+K`
   b. Pressing `Ctrl+A`
   c. `!number`
   d. Pressing `Esc+.`

Solution

Bash Commands and Keyboard Shortcuts

Choose the correct answers to the following questions:

1. **Which Bash shortcut or command jumps to the beginning of the previous word on the command line?**
   a. Pressing **`Ctrl`**+**`LeftArrow`**
   b. Pressing `Ctrl`+`K`
   c. Pressing `Ctrl`+`A`
   d. `!string`
   e. `!number`

2. **Which Bash shortcut or command separates commands on the same line?**
   a. Pressing `Tab`
   b. `history`
   c. `;`
   d. `!string`
   e. Pressing `Esc`+`.`

3. **Which Bash shortcut or command is used to clear characters from the cursor to the end of the command line?**
   a. Pressing `Ctrl`+`LeftArrow`
   b. Pressing **`Ctrl`**+**`K`**
   c. Pressing `Ctrl`+`A`
   d. `;`
   e. Pressing `Esc`+`.`

4. **Which Bash shortcut or command is used to re-execute a recent command by matching the command name?**
   a. Pressing `Tab`
   b. `!number`
   c. **`!string`**
   d. `history`
   e. Pressing `Esc`+`.`

5. **Which Bash shortcut or command is used to complete commands, file names, and options?**
   a. `;`
   b. `!number`
   c. `history`
   d. Pressing **`Tab`**
   e. Pressing `Esc+.`

6. **Which Bash shortcut or command re-executes a specific command in the history list?**
   a. Pressing `Tab`
   b. **`!number`**
   c. `!string`
   d. `history`
   e. Pressing `Esc+.`

7. **Which Bash shortcut or command jumps to the beginning of the command line?**
   a. `!number`
   b. `!string`
   c. Pressing `Ctrl+LeftArrow`
   d. Pressing `Ctrl+K`
   e. Pressing **`Ctrl+A`**

8. **Which Bash shortcut or command displays the list of previous commands?**
   a. Pressing `Tab`
   b. `!string`
   c. `!number`
   d. **`history`**
   e. Pressing `Esc+.`

9. **Which Bash shortcut or command copies the last argument of previous commands?**
   a. Pressing `Ctrl+K`
   b. Pressing `Ctrl+A`
   c. `!number`
   d. Pressing **`Esc+.`**

Guided Exercise

Accessing the Command Line

In this exercise, you will use the Bash shell to execute commands.

Outcomes

- Successfully run simple programs using the Bash shell command line.
- Practice using some Bash command history "shortcuts" to more efficiently repeat commands or parts of commands.

Before You Begin

This exercise was prepared assuming that you have an Amazon EC2 account, have used it to launch an Amazon EC2 cloud instance based on a free-tier eligible version of Red Hat Enterprise Linux 8, and that you can connect to that instance using **ssh** key-based authentication as the regular user **ec2-user**.

See the setup document provided with the materials for this course for additional information on how to do this.

Steps

1. Use **ssh** to log into your Amazon EC2 system as **ec2-user**.

2. Use the **date** command to display the current time and date.

```
[ec2-user@ip-192-0-2-1 ~]$ date
Fri Jul 24 10:13:04 PDT 2020
```

In this example, **[ec2-user@ip-192-0-2-1 ~]$** is the shell's *command prompt*, how the shell prompts you to enter a command. It indicates the current user (**ec2-user**), the short hostname of the machine (**ip-192-0-2-1** in this example, although yours will be different), and information about the current directory (which will be discussed later). **date** is the command you typed. After you press **Enter**, the output of **date** is printed by the computer below your prompt (**Fri Jul 24 10:13:04 PDT 2020** in the example). You should get a new prompt after the command completes.

3. Use the **+%r** argument with the **date** command to display the current time in 12-hour clock time (for example, 11:42:11 AM).

```
[ec2-user@ip-192-0-2-1 ~]$ date +%r
10:14:07 AM
```

4. What kind of file is **/usr/bin/zcat**? Is it readable by humans? Use the **file** command to determine its file type.

```
[ec2-user@ip-192-0-2-1 ~]$ file /usr/bin/zcat
/usr/bin/zcat: POSIX shell script, ASCII text executable
```

5. The **wc** command can be used to display the number of lines, words, and bytes in the script **/usr/bin/zcat**. Instead of retyping the file name, use the Bash history shortcut **Esc+.** (the keys **Esc** and **.** pressed at the same time) to reuse the argument from the previous command.

```
[ec2-user@ip-192-0-2-1 ~]$ wc <Esc>.
[ec2-user@ip-192-0-2-1 ~]$ wc /usr/bin/zcat
  51  299 1983 /usr/bin/zcat
```

6. Use the **head** command to display the first 10 lines of **/usr/bin/zcat**. Try using the **Esc+.** shortcut again.

   The **head** command displays the beginning of the file. Did you use the **bash** shortcut again?

```
[ec2-user@ip-192-0-2-1 ~]$ head <Esc>.
[ec2-user@ip-192-0-2-1 ~]$ head /usr/bin/zcat
#!/bin/sh
# Uncompress files to standard output.

# Copyright (C) 2007, 2010-2018 Free Software Foundation, Inc.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.
```

7. Display the last 10 lines at the bottom of the **/usr/bin/zcat** file. Use the **tail** command.

```
[ec2-user@ip-192-0-2-1 ~]$ tail <Esc>.
[ec2-user@ip-192-0-2-1 ~]$ tail /usr/bin/zcat
With no FILE, or when FILE is -, read standard input.

Report bugs to <bug-gzip@gnu.org>."

case $1 in
--help)    printf '%s\n' "$usage"   || exit 1; exit;;
--version) printf '%s\n' "$version" || exit 1; exit;;
esac

exec gzip -cd "$@"
```

8. Repeat the previous command exactly. Either press the **UpArrow** key once to scroll back through the command history one command and press **Enter**, or run the shortcut command **!!** to run the most recent command in the command history. (Try both!)

```
[ec2-user@ip-192-0-2-1 ~]$ !!
tail /usr/bin/zcat
With no FILE, or when FILE is -, read standard input.

Report bugs to <bug-gzip@gnu.org>."

case $1 in
--help)    printf '%s\n' "$usage"   || exit 1; exit;;
--version) printf '%s\n' "$version" || exit 1; exit;;
esac

exec gzip -cd "$@"
```

9. Repeat the previous command again, but this time add the **-n 20** option to display the last 20 lines in the file. Use command line editing to accomplish this with a minimal amount of keystrokes.

   UpArrow displays the previous command. **Ctrl**+**a** makes the cursor jump to the beginning of the line. **Ctrl**+**Right Arrow** jumps to the next word, then add the **-n 20** option and press **Enter** to run the command.

```
[ec2-user@ip-192-0-2-1 ~]$ tail -n 20 /usr/bin/zcat
  -l, --list        list compressed file contents
  -q, --quiet       suppress all warnings
  -r, --recursive   operate recursively on directories
  -S, --suffix=SUF  use suffix SUF on compressed files
      --synchronous synchronous output (safer if system crashes, but slower)
  -t, --test        test compressed file integrity
  -v, --verbose     verbose mode
      --help        display this help and exit
      --version     display version information and exit

With no FILE, or when FILE is -, read standard input.

Report bugs to <bug-gzip@gnu.org>."

case $1 in
--help)    printf '%s\n' "$usage"   || exit 1; exit;;
--version) printf '%s\n' "$version" || exit 1; exit;;
esac

exec gzip -cd "$@"
```

10. Use the shell history to run the **`date +%r`** command again.

    Display the list of previous commands with the **`history`** command to identify the specific **`date`** command to be executed. Run the command with the **`!number`** history command.

    Note that your shell history may be different than the following example. Figure out the command number to use based on the output of your own **`history`** command.

```
[ec2-user@ip-192-0-2-1 ~]$ history
    1  date
    2  date +%r
    3  file /usr/bin/zcat
    4  wc /usr/bin/zcat
    5  head /usr/bin/zcat
    6  tail /usr/bin/zcat
    7  tail -n 20 /usr/bin/zcat
    8  history
[ec2-user@ip-192-0-2-1 ~]$ !2
date +%r
10:21:56 AM
```

11. Finish your session with the **`bash`** shell.

    Use either **`exit`** or the **`Ctrl`**+**`d`** key combination to close the shell and log out.

```
[ec2-user@ip-192-0-2-1 ~]$ exit
```

12. This concludes this exercise. Stop your Amazon EC2 instance.

Chapter 3

Managing Files From the Command Line

Goal

To copy, move, create, delete, and organize files while working from the Bash shell prompt.

Objectives

- Identify the purpose for important directories on a Linux system.
- Specify files using absolute and relative path names.
- Create, copy, move, and remove files and directories using command-line utilities.
- Match one or more file names using shell expansion as arguments to shell commands.

Sections

- The Linux File System Hierarchy (and Quiz)
- Locating Files by Name (and Quiz)
- Managing Files Using Command-Line Tools (and Guided Exercise)
- Matching File Names Using Path Name Expansion (and Quiz)

Lab

Managing Files with Shell Expansion

The Linux File System Hierarchy

Objectives

After completing this section, you should be able to describe how Linux organizes files, and the purposes of various directories in the file-system hierarchy.

The File-system Hierarchy

All files on a Linux system are stored on file systems, which are organized into a single *inverted* tree of directories, known as a *file-system hierarchy*. This tree is inverted because the root of the tree is said to be at the *top* of the hierarchy, and the branches of directories and subdirectories stretch *below* the root.

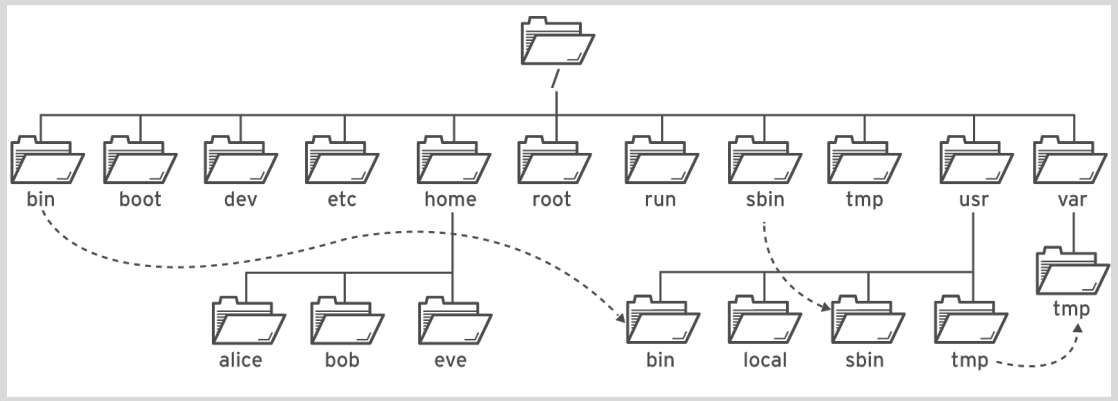


Figure 3.1: Significant file-system directories in Red Hat Enterprise Linux 8

The **/** directory is the root directory at the top of the file-system hierarchy. The **/** character is also used as a *directory separator* in file names. For example, if **etc** is a subdirectory of the **/** directory, you could refer to that directory as **/etc**. Likewise, if the **/etc** directory contained a file named **issue**, you could refer to that file as **/etc/issue**.

Subdirectories of **/** are used for standardized purposes to organize files by type and purpose. This makes it easier to find files. For example, in the root directory, the subdirectory **/boot** is used for storing files needed to boot the system.

Note

The following terms help to describe file-system directory contents:

- *static* content remains unchanged until explicitly edited or reconfigured.
- *dynamic* or *variable* content may be modified or appended by active processes.
- *persistent* content remains after a reboot, like configuration settings.
- *runtime* content is process- or system-specific content that is deleted by a reboot.

The following table lists some of the most important directories on the system by name and purpose.

Important Red Hat Enterprise Linux Directories

Location	Purpose
/usr	Installed software, shared libraries, include files, and read-only program data. Important subdirectories include: • **/usr/bin**: User commands. • **/usr/sbin**: System administration commands. • **/usr/local**: Locally customized software.
/etc	Configuration files specific to this system.
/var	Variable data specific to this system that should persist between boots. Files that dynamically change, such as databases, cache directories, log files, printer-spooled documents, and website content may be found under **/var**.
/run	Runtime data for processes started since the last boot. This includes process ID files and lock files, among other things. The contents of this directory are recreated on reboot. This directory consolidates **/var/run** and **/var/lock** from earlier versions of Red Hat Enterprise Linux.
/home	*Home directories* are where regular users store their personal data and configuration files.
/root	Home directory for the administrative superuser, **root**.
/tmp	A world-writable space for temporary files. Files which have not been accessed, changed, or modified for 10 days are deleted from this directory automatically. Another temporary directory exists, **/var/tmp**, in which files that have not been accessed, changed, or modified in more than 30 days are deleted automatically.
/boot	Files needed in order to start the boot process.
/dev	Contains special *device files* that are used by the system to access hardware.

Important

In Red Hat Enterprise Linux 7 and later, four older directories in **/** have identical contents to their counterparts located in **/usr**:

- **/bin** and **/usr/bin**
- **/sbin** and **/usr/sbin**
- **/lib** and **/usr/lib**
- **/lib64** and **/usr/lib64**

In earlier versions of Red Hat Enterprise Linux, these were distinct directories containing different sets of files.

In Red Hat Enterprise Linux 7 and later, the directories in **/** are symbolic links to the matching directories in **/usr**.

References

hier(7) man page

The UsrMove feature page from Fedora 17
https://fedoraproject.org/wiki/Features/UsrMove

Quiz

File System Hierarchy

Choose the correct answers to the following questions:

1. **Which directory contains persistent, system-specific configuration data?**
   a. `/etc`
   b. `/root`
   c. `/run`
   d. `/usr`

2. **Which directory is the top of the system's file system hierarchy?**
   a. `/etc`
   b. `/`
   c. `/home/root`
   d. `/root`

3. **Which directory contains user home directories?**
   a. `/`
   b. `/home`
   c. `/root`
   d. `/user`

4. **Which directory contains temporary files?**
   a. `/tmp`
   b. `/trash`
   c. `/run`
   d. `/var`

5. **Which directory contains dynamic data, such as for databases and websites?**
   a. `/etc`
   b. `/run`
   c. `/usr`
   d. `/var`

6. **Which directory is the administrative superuser's home directory?**
   a. `/etc`
   b. `/`
   c. `/home/root`
   d. `/root`

7. Which directory contains regular commands and utilities?

a. **`/commands`**
b. **`/run`**
c. **`/usr/bin`**
d. **`/usr/sbin`**

8. Which directory contains non-persistent process runtime data?

a. **`/tmp`**
b. **`/etc`**
c. **`/run`**
d. **`/var`**

9. Which directory contains installed software programs and libraries?

a. **`/etc`**
b. **`/lib`**
c. **`/usr`**
d. **`/var`**

Solution

File System Hierarchy

Choose the correct answers to the following questions:

1. **Which directory contains persistent, system-specific configuration data?**
   a. **`/etc`**
   b. `/root`
   c. `/run`
   d. `/usr`

2. **Which directory is the top of the system's file system hierarchy?**
   a. `/etc`
   b. **`/`**
   c. `/home/root`
   d. `/root`

3. **Which directory contains user home directories?**
   a. `/`
   b. **`/home`**
   c. `/root`
   d. `/user`

4. **Which directory contains temporary files?**
   a. **`/tmp`**
   b. `/trash`
   c. `/run`
   d. `/var`

5. **Which directory contains dynamic data, such as for databases and websites?**
   a. `/etc`
   b. `/run`
   c. `/usr`
   d. **`/var`**

6. **Which directory is the administrative superuser's home directory?**
   a. `/etc`
   b. `/`
   c. `/home/root`
   d. **`/root`**

7. Which directory contains regular commands and utilities?

a. `/commands`
b. `/run`
c. **`/usr/bin`**
d. `/usr/sbin`

8. Which directory contains non-persistent process runtime data?

a. `/tmp`
b. `/etc`
c. **`/run`**
d. `/var`

9. Which directory contains installed software programs and libraries?

a. `/etc`
b. `/lib`
c. **`/usr`**
d. `/var`

Locating Files by Name

Objectives

After completing this section, you should be able to specify the location of files relative to the current working directory and by absolute location, determine and change the working directory, and list the contents of directories.

Absolute Paths and Relative Paths

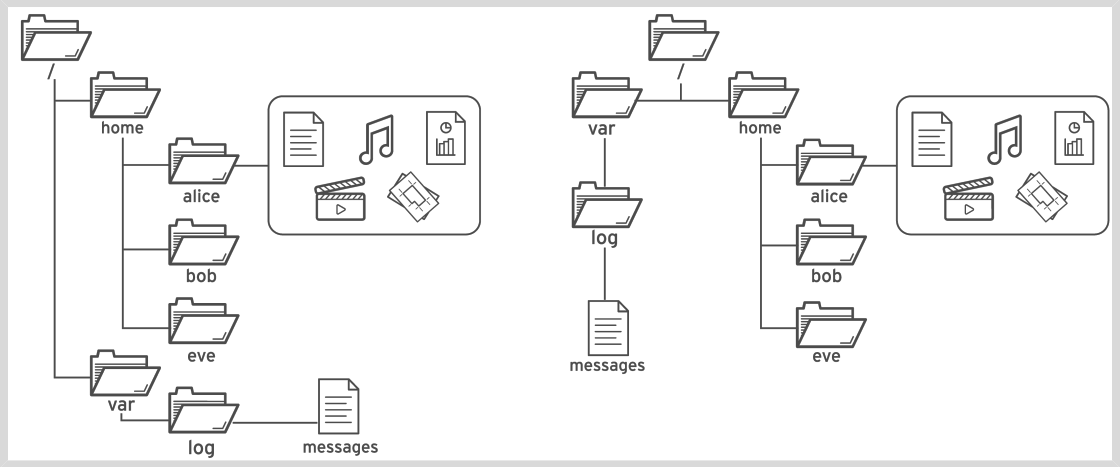


Figure 3.2: The common file browser view (left) is equivalent to the top-down view (right).

The *path* of a file or directory specifies its unique file system location. Following a file path traverses one or more named subdirectories, delimited by a forward slash (/), until the destination is reached. Directories, also called *folders*, contain other files and other subdirectories. They can be referenced in the same manner as files.

Important

A space character is acceptable as part of a Linux file name. However, spaces are also used by the shell to separate options and arguments on the command line. If you enter a command that includes a file that has a space in its name, the shell can misinterpret the command and assume that you want to start a new file name or other argument at the space.

It is possible to avoid this by putting file names in quotes. However, if you do not need to use spaces in file names, it can be simpler to simply avoid using them.

Absolute Paths

An *absolute path* is a *fully qualified* name, specifying the files exact location in the file system hierarchy. It begins at the root (/) directory and specifies each subdirectory that must be traversed to reach the specific file. Every file in a file system has a unique absolute path name, recognized with a simple rule: A path name with a forward slash (/) as the first character is an absolute path name. For example, the absolute path name for the system message log file is /

`var/log/messages`. Absolute path names can be long to type, so files may also be located *relative* to the current working directory for your shell prompt.

The Current Working Directory and Relative Paths

When a user logs in and opens a command window, the initial location is normally the user's home directory. System processes also have an initial directory. Users and processes navigate to other directories as needed; the terms *working directory* or *current working directory* refer to their *current* location.

Like an absolute path, a *relative path* identifies a unique file, specifying only the path necessary to reach the file from the working directory. Recognizing relative path names follows a simple rule: A path name with *anything other than* a forward slash as the first character is a relative path name. A user in the **`/var`** directory could refer to the message log file relatively as **`log/messages`**.

Linux file systems, including, but not limited to, ext4, XFS, GFS2, and GlusterFS, are case-sensitive. Creating **`FileCase.txt`** and **`filecase.txt`** in the same directory results in two unique files.

Non-Linux file systems might work differently. For example, VFAT, Microsoft's NTFS, and Apple's HFS+ have *case preserving* behavior. Although these file systems are *not* case-sensitive, they do display file names with the original capitalization used when the file was created. Therefore, if you tried to make the files in the preceding example on a VFAT file system, both names would be treated as pointing to the same file instead of two different files.

Navigating Paths

The **`pwd`** command displays the full path name of the current working directory for that shell. This can help you determine the syntax to reach files using relative path names. The **`ls`** command lists directory contents for the specified directory or, if no directory is given, for the current working directory.

```
[user@host ~]$ pwd
/home/user
[user@host ~]$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
[user@host ~]$
```

Use the **`cd`** command to change your shell's current working directory. If you do not specify any arguments to the command, it will change to your home directory.

In the following example, a mixture of absolute and relative paths are used with the **`cd`** command to change the current working directory for the shell.

```
[user@host ~]$ pwd
/home/user
[user@host ~]$ cd Videos
[user@host Videos]$ pwd
/home/user/Videos
[user@host Videos]$ cd /home/user/Documents
[user@host Documents]$ pwd
/home/user/Documents
[user@host Documents]$ cd
```

```
[user@host ~]$ pwd
/home/user
[user@host ~]$
```

As you can see in the preceding example, the default shell prompt also displays the last component of the absolute path to the current working directory. For example, for **/home/user/Videos**, only **Videos** displays. The prompt displays the tilde character (~) when your current working directory is your home directory.

The **touch** command normally updates a file's timestamp to the current date and time without otherwise modifying it. This is useful for creating empty files, which can be used for practice, because "touching" a file name that does not exist causes the file to be created. In the following example, the **touch** command creates practice files in the **Documents** and **Videos** subdirectories.

```
[user@host ~]$ touch Videos/blockbuster1.ogg
[user@host ~]$ touch Videos/blockbuster2.ogg
[user@host ~]$ touch Documents/thesis_chapter1.odf
[user@host ~]$ touch Documents/thesis_chapter2.odf
[user@host ~]$
```

The **ls** command has multiple options for displaying attributes on files. The most common and useful are **-l** (long listing format), **-a** (all files, including *hidden* files), and **-R** (recursive, to include the contents of all subdirectories).

```
[user@host ~]$ ls -l
total 15
drwxr-xr-x.  2 user user 4096 Feb  7 14:02 Desktop
drwxr-xr-x.  2 user user 4096 Jan  9 15:00 Documents
drwxr-xr-x.  3 user user 4096 Jan  9 15:00 Downloads
drwxr-xr-x.  2 user user 4096 Jan  9 15:00 Music
drwxr-xr-x.  2 user user 4096 Jan  9 15:00 Pictures
drwxr-xr-x.  2 user user 4096 Jan  9 15:00 Public
drwxr-xr-x.  2 user user 4096 Jan  9 15:00 Templates
drwxr-xr-x.  2 user user 4096 Jan  9 15:00 Videos
[user@host ~]$ ls -la
total 15
drwx------. 16 user user   4096 Feb  8 16:15 .
drwxr-xr-x.  6 root root   4096 Feb  8 16:13 ..
-rw-------.  1 user user  22664 Feb  8 00:37 .bash_history
-rw-r--r--.  1 user user     18 Jul  9  2013 .bash_logout
-rw-r--r--.  1 user user    176 Jul  9  2013 .bash_profile
-rw-r--r--.  1 user user    124 Jul  9  2013 .bashrc
drwxr-xr-x.  4 user user   4096 Jan 20 14:02 .cache
drwxr-xr-x.  8 user user   4096 Feb  5 11:45 .config
drwxr-xr-x.  2 user user   4096 Feb  7 14:02 Desktop
drwxr-xr-x.  2 user user   4096 Jan  9 15:00 Documents
drwxr-xr-x.  3 user user   4096 Jan 25 20:48 Downloads
drwxr-xr-x. 11 user user   4096 Feb  6 13:07 .gnome2
drwx------.  2 user user   4096 Jan 20 14:02 .gnome2_private
-rw-------.  1 user user  15190 Feb  8 09:49 .ICEauthority
drwxr-xr-x.  3 user user   4096 Jan  9 15:00 .local
drwxr-xr-x.  2 user user   4096 Jan  9 15:00 Music
drwxr-xr-x.  2 user user   4096 Jan  9 15:00 Pictures
```

```
drwxr-xr-x.  2 user user   4096 Jan  9 15:00 Public
drwxr-xr-x.  2 user user   4096 Jan  9 15:00 Templates
drwxr-xr-x.  2 user user   4096 Jan  9 15:00 Videos
[user@host ~]$
```

The two special directories at the top of the listing refer to the current directory (**.**) and the *parent* directory (**..**). These special directories exist in every directory on the system. You will discover their usefulness when you start using file management commands.

Important

File names beginning with a dot (.) indicate *hidden* files; you cannot see them in the normal view using **ls** and other commands. This is *not* a security feature. Hidden files keep necessary user configuration files from cluttering home directories. Many commands process hidden files only with specific command-line options, preventing one user's configuration from being accidentally copied to other directories or users.

To protect file *contents* from improper viewing requires the use of *file permissions*.

```
[user@host ~]$ ls -R
.:
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos

./Desktop:

./Documents:
thesis_chapter1.odf  thesis_chapter2.odf

./Downloads:

./Music:

./Pictures:

./Public:

./Templates:

./Videos:
blockbuster1.ogg  blockbuster2.ogg
[user@host ~]$
```

The **cd** command has many options. A few are so useful as to be worth practicing early and using often. The command **cd -** changes to the previous directory; where the user was *previously* to the current directory. The following example illustrates this behavior, alternating between two directories, which is useful when processing a series of similar tasks.

```
[user@host ~]$ cd Videos
[user@host Videos]$ pwd
/home/user/Videos
[user@host Videos]$ cd /home/user/Documents
[user@host Documents]$ pwd
/home/user/Documents
```

```
[user@host Documents]$ cd -
[user@host Videos]$ pwd
/home/user/Videos
[user@host Videos]$ cd -
[user@host Documents]$ pwd
/home/user/Documents
[user@host Documents]$ cd -
[user@host Videos]$ pwd
/home/user/Videos
[user@host Videos]$ cd
[user@host ~]$
```

The **cd ..** command uses the **..** hidden directory to move up one level to the *parent* directory, without needing to know the exact parent name. The other hidden directory (**.**) specifies the *current directory* on commands in which the current location is either the source or destination argument, avoiding the need to type out the directory's absolute path name.

```
[user@host Videos]$ pwd
/home/user/Videos
[user@host Videos]$ cd .
[user@host Videos]$ pwd
/home/user/Videos
[user@host Videos]$ cd ..
[user@host ~]$ pwd
/home/user
[user@host ~]$ cd ..
[user@host home]$ pwd
/home
[user@host home]$ cd ..
[user@host /]$ pwd
/
[user@host /]$ cd
[user@host ~]$ pwd
/home/user
[user@host ~]$
```

References

info libc 'file name resolution' (*GNU C Library Reference Manual*)

- Section 11.2.2 File name resolution

bash(1), **cd**(1), **ls**(1), **pwd**(1), **unicode**(7), and **utf-8**(7) man pages

UTF-8 and Unicode
http://www.utf-8.com/

Quiz

Locating Files and Directories

Choose the correct answers to the following questions:

1. **Which command is used to return to the current user's home directory, assuming the current working directory is `/tmp` and their home directory is `/home/user`?**
   a. **`cd`**
   b. **`cd ..`**
   c. **`cd .`**
   d. **`cd *`**
   e. **`cd /home`**

2. **Which command displays the absolute path name of the current location?**
   a. **`cd`**
   b. **`pwd`**
   c. **`ls ~`**
   d. **`ls -d`**

3. **Which command will always return you to the working directory used prior to the current working directory?**
   a. **`cd -`**
   b. **`cd -p`**
   c. **`cd ~`**
   d. **`cd ..`**

4. **Which command will always change the working directory up two levels from the current location?**
   a. **`cd ~`**
   b. **`cd ../`**
   c. **`cd ../..`**
   d. **`cd -u2`**

5. **Which command lists files in the current location, using a long format, and including hidden files?**
   a. **`llong ~`**
   b. **`ls -a`**
   c. **`ls -l`**
   d. **`ls -al`**

6. **Which command will always change the working directory to `/bin`?**
   a. **`cd bin`**
   b. **`cd /bin`**
   c. **`cd ~bin`**
   d. **`cd -bin`**

7. **Which command will always change the working directory to the parent of the current location?**
   a. **`cd ~`**
   b. **`cd ..`**
   c. **`cd ../..`**
   d. **`cd -u1`**

8. **Which command will change the working directory to `/tmp` if the current working directory is `/home/student`?**
   a. **`cd tmp`**
   b. **`cd ..`**
   c. **`cd ../../tmp`**
   d. **`cd ~tmp`**

Solution

Locating Files and Directories

Choose the correct answers to the following questions:

1. **Which command is used to return to the current user's home directory, assuming the current working directory is `/tmp` and their home directory is `/home/user`?**
   a. **`cd`**
   b. `cd ..`
   c. `cd .`
   d. `cd *`
   e. `cd /home`

2. **Which command displays the absolute path name of the current location?**
   a. `cd`
   b. **`pwd`**
   c. `ls ~`
   d. `ls -d`

3. **Which command will always return you to the working directory used prior to the current working directory?**
   a. **`cd -`**
   b. `cd -p`
   c. `cd ~`
   d. `cd ..`

4. **Which command will always change the working directory up two levels from the current location?**
   a. `cd ~`
   b. `cd ../`
   c. **`cd ../..`**
   d. `cd -u2`

5. **Which command lists files in the current location, using a long format, and including hidden files?**
   a. `llong ~`
   b. `ls -a`
   c. `ls -l`
   d. **`ls -al`**

6. **Which command will always change the working directory to `/bin`?**
   a. `cd bin`
   b. **`cd /bin`**
   c. `cd ~bin`
   d. `cd -bin`

7. **Which command will always change the working directory to the parent of the current location?**
   a. `cd ~`
   b. **`cd ..`**
   c. `cd ../..`
   d. `cd -u1`

8. **Which command will change the working directory to `/tmp` if the current working directory is `/home/student`?**
   a. `cd tmp`
   b. `cd ..`
   c. **`cd ../../tmp`**
   d. `cd ~tmp`

Managing Files Using Command-line Tools

Objectives

After completing this section, you should be able to create, copy, move, and remove files and directories.

Command-line File Management

To manage files, you need to be able to create, remove, copy, and move them. You also need to organize them logically into directories, which you also need to be able to create, remove, copy, and move.

The following table summarizes some of the most common file management commands. The remainder of this section will discuss ways to use these commands in more detail.

Common file management commands

Activity	Command Syntax
Create a directory	**mkdir *directory***
Copy a file	**cp *file new-file***
Copy a directory and its contents	**cp -r *directory new-directory***
Move or rename a file or directory	**mv *file new-file***
Remove a file	**rm *file***
Remove a directory containing files	**rm -r *directory***
Remove an empty directory	**rmdir *directory***

Creating Directories

The **mkdir** command creates one or more directories or subdirectories. It takes as arguments a list of paths to the directories you want to create.

The **mkdir** command will fail with an error if the directory already exists, or if you are trying to create a subdirectory in a directory that does not exist. The **-p** (*parent*) option creates missing parent directories for the requested destination. Use the **mkdir -p** command with caution, because spelling mistakes can create unintended directories without generating error messages.

In the following example, pretend that you are trying to create a directory in the **Videos** directory named **Watched**, but you accidentally left off the letter "s" in **Videos** in your **mkdir** command.

```
[user@host ~]$ mkdir Video/Watched
mkdir: cannot create directory `Video/Watched': No such file or directory
```

The **mkdir** command failed because **Videos** was misspelled and the directory **Video** does not exist. If you had used the **mkdir** command with the **-p** option, the directory **Video** would be created, which was not what you had intended, and the subdirectory **Watched** would be created in that incorrect directory.

After correctly spelling the **Videos** parent directory, creating the **Watched** subdirectory will succeed.

```
[user@host ~]$ mkdir Videos/Watched
[user@host ~]$ ls -R Videos
Videos/:
blockbuster1.ogg  blockbuster2.ogg  Watched

Videos/Watched:
```

In the following example, files and directories are organized beneath the **/home/user/Documents** directory. Use the **mkdir** command and a space-delimited list of the directory names to create multiple directories.

```
[user@host ~]$ cd Documents
[user@host Documents]$ mkdir ProjectX ProjectY
[user@host Documents]$ ls
ProjectX  ProjectY
```

Use the **mkdir -p** command and space-delimited relative paths for each of the subdirectory names to create multiple parent directories with subdirectories.

```
[user@host Documents]$ mkdir -p Thesis/Chapter1 Thesis/Chapter2 Thesis/Chapter3
[user@host Documents]$ cd
[user@host ~]$ ls -R Videos Documents
Documents:
ProjectX  ProjectY  Thesis

Documents/ProjectX:

Documents/ProjectY:

Documents/Thesis:
Chapter1  Chapter2  Chapter3

Documents/Thesis/Chapter1:

Documents/Thesis/Chapter2:

Documents/Thesis/Chapter3:

Videos:
blockbuster1.ogg  blockbuster2.ogg  Watched

Videos/Watched:
```

The last **mkdir** command created three Chapter*N* subdirectories with one command. The **-p** option created the missing parent directory **Thesis**.

Copying Files

The **cp** command copies a file, creating a new file either in the current directory or in a specified directory. It can also copy multiple files to a directory.

Warning
If the destination file already exists, the **cp** command overwrites the file.

```
[user@host ~]$ cd Videos
[user@host Videos]$ cp blockbuster1.ogg blockbuster3.ogg
[user@host Videos]$ ls -l
total 0
-rw-rw-r--. 1 user user    0 Feb  8 16:23 blockbuster1.ogg
-rw-rw-r--. 1 user user    0 Feb  8 16:24 blockbuster2.ogg
-rw-rw-r--. 1 user user    0 Feb  8 16:34 blockbuster3.ogg
drwxrwxr-x. 2 user user 4096 Feb  8 16:05 Watched
[user@host Videos]$
```

When copying multiple files with one command, the last argument must be a directory. Copied files retain their original names in the new directory. If a file with the same name exists in the target directory, the existing file is overwritten. By default, the **cp** does not copy directories; it ignores them.

In the following example, two directories are listed, **Thesis** and **ProjectX**. Only the last argument, **ProjectX** is valid as a destination. The **Thesis** directory is ignored.

```
[user@host Videos]$ cd ../Documents
[user@host Documents]$ cp thesis_chapter1.odf thesis_chapter2.odf Thesis ProjectX
cp: omitting directory `Thesis'
[user@host Documents]$ ls Thesis ProjectX
ProjectX:
thesis_chapter1.odf  thesis_chapter2.odf

Thesis:
Chapter1  Chapter2  Chapter3
```

In the first **cp** command, the **Thesis** directory failed to copy, but the **thesis_chapter1.odf** and **thesis_chapter2.odf** files succeeded.

If you want to copy a file to the current working directory, you can use the special **.** directory:

```
[user@host ~]$ cp /etc/hostname .
[user@host ~]$ cat hostname
host.example.com
[user@host ~]$
```

Use the copy command with the **-r** (*recursive*) option, to copy the **Thesis** directory and its contents to the **ProjectX** directory.

```
[user@host Documents]$ cp -r Thesis ProjectX
[user@host Documents]$ ls -R ProjectX
ProjectX:
Thesis  thesis_chapter1.odf  thesis_chapter2.odf

ProjectX/Thesis:
Chapter1  Chapter2  Chapter3

ProjectX/Thesis/Chapter1:

ProjectX/Thesis/Chapter2:
thesis_chapter2.odf

ProjectX/Thesis/Chapter3:
```

Moving Files

The **mv** command moves files from one location to another. If you think of the absolute path to a file as its full name, moving a file is effectively the same as renaming a file. File contents remain unchanged.

Use the **mv** command to *rename* a file.

```
[user@host Videos]$ cd ../Documents
[user@host Documents]$ ls -l thesis*
-rw-rw-r--. 1 user user 0 Feb  6 21:16 thesis_chapter1.odf
-rw-rw-r--. 1 user user 0 Feb  6 21:16 thesis_chapter2.odf
[user@host Documents]$ mv thesis_chapter2.odf thesis_chapter2_reviewed.odf
[user@host Documents]$ ls -l thesis*
-rw-rw-r--. 1 user user 0 Feb  6 21:16 thesis_chapter1.odf
-rw-rw-r--. 1 user user 0 Feb  6 21:16 thesis_chapter2_reviewed.odf
```

Use the **mv** command to *move* a file to a different directory.

```
[user@host Documents]$ ls Thesis/Chapter1
[user@host Documents]$
[user@host Documents]$ mv thesis_chapter1.odf Thesis/Chapter1
[user@host Documents]$ ls Thesis/Chapter1
thesis_chapter1.odf
[user@host Documents]$ ls -l thesis*
-rw-rw-r--. 1 user user 0 Feb  6 21:16 thesis_chapter2_reviewed.odf
```

Removing Files and Directories

The **rm** command removes files. By default, **rm** will not remove directories that contain files, unless you add the **-r** or **--recursive** option.

Important

There is no command-line undelete feature, nor a "trash bin" from which you can restore files staged for deletion.

It is a good idea to verify your current working directory before removing a file or directory.

```
[user@host Documents]$ pwd
/home/student/Documents
```

Use the **rm** command to remove a single file from your working directory.

```
[user@host Documents]$ ls -l thesis*
-rw-rw-r--. 1 user user 0 Feb  6 21:16 thesis_chapter2_reviewed.odf
[user@host Documents]$ rm thesis_chapter2_reviewed.odf
[user@host Documents]$ ls -l thesis*
ls: cannot access 'thesis*': No such file or directory
```

If you attempt to use the **rm** command to remove a directory without using the **-r** option, the command will fail.

```
[user@host Documents]$ rm Thesis/Chapter1
rm: cannot remove `Thesis/Chapter1': Is a directory
```

Use the **rm -r** command to remove a subdirectory and its contents.

```
[user@host Documents]$ ls -R Thesis
Thesis/:
Chapter1  Chapter2  Chapter3

Thesis/Chapter1:
thesis_chapter1.odf

Thesis/Chapter2:
thesis_chapter2.odf

Thesis/Chapter3:
[user@host Documents]$ rm -r Thesis/Chapter1
[user@host Documents]$ ls -l Thesis
total 8
drwxrwxr-x. 2 user user 4096 Feb 11 12:47 Chapter2
drwxrwxr-x. 2 user user 4096 Feb 11 12:48 Chapter3
```

The **rm -r** command traverses each subdirectory first, individually removing their files before removing each directory. You can use the **rm -ri** command to interactively prompt for confirmation before deleting. This is essentially the opposite of using the **-f** option, which forces the removal without prompting the user for confirmation.

```
[user@host Documents]$ rm -ri Thesis
rm: descend into directory `Thesis'? y
rm: descend into directory `Thesis/Chapter2'? y
rm: remove regular empty file `Thesis/Chapter2/thesis_chapter2.odf'? y
rm: remove directory `Thesis/Chapter2'? y
rm: remove directory `Thesis/Chapter3'? y
rm: remove directory `Thesis'? y
[user@host Documents]$
```

Warning

If you specify both the **-i** and **-f** options, the **-f** option takes priority and you will not be prompted for confirmation before **rm** deletes files.

In the following example, the **rmdir** command only removes the directory that is empty. Just like the earlier example, you must use the **rm -r** command to remove a directory that contains content.

```
[user@host Documents]$ pwd
/home/student/Documents
[user@host Documents]$ rmdir ProjectY
[user@host Documents]$ rmdir ProjectX
rmdir: failed to remove `ProjectX': Directory not empty
[user@host Documents]$ rm -r ProjectX
[user@host Documents]$ ls -lR
.:
total 0
[user@host Documents]$
```

Note

The **rm** command with no options cannot remove an empty directory. You must use the **rmdir** command, **rm -d** (which is equivalent to **rmdir**), or **rm -r**.

References

cp(1), **mkdir**(1), **mv**(1), **rm**(1), and **rmdir**(1) man pages

Guided Exercise

Command Line File Management

In this exercise, you will practice efficient techniques for creating and organizing files using directories, file copies, and links.

Outcomes:

- Successfully create, move, copy, and delete files using the command line.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**.

Steps

1. Make sure you're in the home directory for your current user by entering the **cd** command. The **pwd** command should report **/home/ec2-user**, and your shell prompt should show **~** as the current directory.

```
[ec2-user@ip-192-0-2-1 ~]$ cd
[ec2-user@ip-192-0-2-1 ~]$ pwd
/home/ec2-user
```

2. In the home directory for **ec2-user**, create sets of empty practice files to use for the remainder of this lab. In each set, replace *X* with the numbers 1 through 6.

   Create six "song" files with names of the form **songX.mp3**.

   Create six "snapshot" files with names of the form **snapX.jpg**.

   Create six "movie" files with names of the form **filmX.avi**.

```
[ec2-user@ip-192-0-2-1 ~]$ touch song1.mp3 song2.mp3 song3.mp3 song4.mp3 song5.mp3
 song6.mp3
[ec2-user@ip-192-0-2-1 ~]$ touch snap1.jpg snap2.jpg snap3.jpg snap4.jpg snap5.jpg
 snap6.jpg
[ec2-user@ip-192-0-2-1 ~]$ touch film1.avi film2.avi film3.avi film4.avi film5.avi
 film6.avi
[ec2-user@ip-192-0-2-1 ~]$ ls -l
...output omitted...
```

3. In the **ec2-user** home directory, create three subdirectories, **Music**, **Pictures**, and **Videos**.

```
[ec2-user@ip-192-0-2-1 ~]$ mkdir Music
[ec2-user@ip-192-0-2-1 ~]$ mkdir Pictures
[ec2-user@ip-192-0-2-1 ~]$ mkdir Videos
```

4. From the **ec2-user** home directory, move the song files into the **Music** subdirectory, the snapshot files into the **Pictures** subdirectory, and the movie files into the **Videos** subdirectory.

   Once you are finished, use the **ls -l** command to list the contents of each of the subdirectories in order to check your work.

```
[ec2-user@ip-192-0-2-1 ~]$ mv song1.mp3 song2.mp3 song3.mp3 song4.mp3 song5.mp3
 song6.mp3 Music
[ec2-user@ip-192-0-2-1 ~]$ mv snap1.jpg snap2.jpg snap3.jpg snap4.jpg snap5.jpg
 snap6.jpg Pictures
[ec2-user@ip-192-0-2-1 ~]$ mv film1.avi film2.avi film3.avi film4.avi film5.avi
 film6.avi Videos
[ec2-user@ip-192-0-2-1 ~]$ ls -l Music Pictures Videos
...output omitted...
```

5. In the **ec2-user** home directory, create three additional subdirectories to organize the files into projects. Call these directories **friends**, **family**, and **work**. Create all three with one command.

   Check your work with the **ls -l** command.

```
[ec2-user@ip-192-0-2-1 ~]$ mkdir friends family work
[ec2-user@ip-192-0-2-1 ~]$ ls -l
drwxrwxr-x. 2 ec2-user ec2-user 6 Apr 17 17:51 family
drwxrwxr-x. 2 ec2-user ec2-user 6 Apr 17 17:51 friends
drwxrwxr-x. 2 ec2-user ec2-user 6 Apr 17 17:47 Music
drwxrwxr-x. 2 ec2-user ec2-user 6 Apr 17 17:47 Pictures
drwxrwxr-x. 2 ec2-user ec2-user 6 Apr 17 17:47 Videos
drwxrwxr-x. 2 ec2-user ec2-user 6 Apr 17 17:51 work
```

6. Next, copy some files from the **Music**, **Pictures**, and **Videos** subdirectories to the **friends** and **family** subdirectories.

   Copy files in the **Music**, **Pictures**, and **Videos** subdirectories containing numbers **1** and **2** to the **friends** folder.

   Copy files in the **Music**, **Pictures**, and **Videos** subdirectories containing numbers **3** and **4** to the **family** folder.

   Before each copy, the example also changes directory to the *destination* directory that files are to be copied to. Note that the example uses the special relative directory name **.** to refer to the shell prompt's current directory for the copy commands. You may find it quicker

to enter commands if you remember that you can use **Tab** to complete file names quickly. This is not the only possible solution to this step.

Use **ls** to check your work.

```
[ec2-user@ip-192-0-2-1 ~]$ cd friends
[ec2-user@ip-192-0-2-1 friends]$ cp ~/Music/song1.mp3 ~/Music/song2.mp3 ~/
Pictures/snap1.jpg ~/Pictures/snap2.jpg ~/Videos/film1.avi ~/Videos/film2.avi .
[ec2-user@ip-192-0-2-1 friends]$ ls
film1.mp3  film2.mp3  snap1.mp3  snap2.mp3  song1.mp3  song2.mp3
[ec2-user@ip-192-0-2-1 friends]$ cd ../family
[ec2-user@ip-192-0-2-1 family]$ cp ~/Music/song3.mp3 ~/Music/song4.mp3 ~/Pictures/
snap3.jpg ~/Pictures/snap4.jpg ~/Videos/film3.avi ~/Videos/film4.avi .
[ec2-user@ip-192-0-2-1 family]$ ls
film3.mp3  film4.mp3  snap3.mp3  snap4.mp3  song3.mp3  song4.mp3
```

7. Now copy files in the **Music**, **Pictures**, and **Videos** subdirectories containing numbers **5** and **6** to the **work** subdirectory.

```
[ec2-user@ip-192-0-2-1 family]$ cd ../work
[ec2-user@ip-192-0-2-1 work]$ cp ~/Music/song5.mp3 ~/Music/song6.mp3 ~/Pictures/
snap5.jpg ~/Pictures/snap6.jpg ~/Videos/film5.avi ~/Videos/film6.avi .
[ec2-user@ip-192-0-2-1 work]$ ls
film5.mp3  film6.mp3  snap5.mp3  snap6.mp3  song5.mp3  song6.mp3
```

8. Finally, it's time to delete the **friends**, **family**, and **work** directories.

   Change to the **ec2-user** home directory. Attempt to delete both the **family** and **friends** directories with a single **rmdir** command.

```
[ec2-user@ip-192-0-2-1 work]$ cd
[ec2-user@ip-192-0-2-1 ~]$ rmdir family friends
rmdir: failed to remove `family': Directory not empty
rmdir: failed to remove `friends': Directory not empty
```

   Using the **rmdir** command should fail since both directories have files in them.

9. Use the **rm** command with the **-r** (recursive) option to delete the **family** and **friends** folders and their copies of the files.

```
[ec2-user@ip-192-0-2-1 ~]$ rm -r family friends
[ec2-user@ip-192-0-2-1 ~]$ ls
Music  Pictures  Videos  work
```

10. Delete all the files in the **work** subdirectory, but do not delete the **work** directory itself.

```
[ec2-user@ip-192-0-2-1 ~]$ cd work
[ec2-user@ip-192-0-2-1 work]$ rm song5.mp3 song6.mp3 snap5.jpg snap6.jpg film5.avi
 film6.avi
[ec2-user@ip-192-0-2-1 work]$ ls -l
total 0
[ec2-user@ip-192-0-2-1 work]$
```

11. Finally, from the home directory, use the **`rmdir`** command to delete the **`work`** directory. The command should succeed now that **`work`** is empty.

```
[ec2-user@ip-192-0-2-1 work]$ cd
[ec2-user@ip-192-0-2-1 ~]$ rmdir work
[ec2-user@ip-192-0-2-1 ~]$ ls
Music  Pictures  Videos
```

12. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Matching File Names Using Path Name Expansion

Objectives

After completing this section, you should be able to efficiently run commands affecting many files by using pattern matching features of the Bash shell.

Command-line Expansions

The Bash shell has multiple ways of expanding a command line including *pattern matching*, home directory expansion, string expansion, and variable substitution. Perhaps the most powerful of these is the path name-matching capability, historically called *globbing*. The Bash globbing feature, sometimes called "wildcards", makes managing large numbers of files easier. Using *metacharacters* that "expand" to match file and path names being sought, commands perform on a focused set of files at once.

Pattern Matching

Globbing is a shell command-parsing operation that expands a wildcard pattern into a list of matching path names. Command-line metacharacters are replaced by the match list prior to command execution. Patterns that do not return matches display the original pattern request as literal text. The following are common metacharacters and pattern classes.

Table of Metacharacters and Matches

Pattern	Matches
*	Any string of zero or more characters.
?	Any single character.
[*abc...*]	Any one character in the enclosed class (between the square brackets).
[!*abc...*]	Any one character *not* in the enclosed class.
[^*abc...*]	Any one character *not* in the enclosed class.
[[:alpha:]]	Any alphabetic character.
[[:lower:]]	Any lowercase character.
[[:upper:]]	Any uppercase character.
[[:alnum:]]	Any alphabetic character or digit.
[[:punct:]]	Any printable character not a space or alphanumeric.
[[:digit:]]	Any single digit from 0 to 9.
[[:space:]]	Any single white space character. This may include tabs, newlines, carriage returns, form feeds, or spaces.

For the next few examples, pretend that you have run the following commands to create some sample files.

```
[user@host ~]$ mkdir glob; cd glob
[user@host glob]$ touch alfa bravo charlie delta echo able baker cast dog easy
[user@host glob]$ ls
able  alfa  baker  bravo  cast  charlie  delta  dog  easy  echo
[user@host glob]$
```

The first example will use simple pattern matches with the asterisk (*) and question mark (?) characters, and a class of characters, to match some of those file names.

```
[user@host glob]$ ls a*
able  alfa
[user@host glob]$ ls *a*
able  alfa  baker  bravo  cast  charlie  delta  easy
[user@host glob]$ ls [ac]*
able  alfa  cast  charlie
[user@host glob]$ ls ????
able  alfa  cast  easy  echo
[user@host glob]$ ls ?????
baker  bravo  delta
[user@host glob]$
```

Tilde Expansion

The tilde character (~), matches the current user's home directory. If it starts a string of characters other than a slash (/), the shell will interpret the string up to that slash as a user name, if one matches, and replace the string with the absolute path to that user's home directory. If no user name matches, then an actual tilde followed by the string of characters will be used instead.

In the following example the **echo** command is used to display the value of the tilde character. The **echo** command can also be used to display the values of brace and variable expansion characters, and others.

```
[user@host glob]$ echo ~root
/root
[user@host glob]$ echo ~user
/home/user
[user@host glob]$ echo ~/glob
/home/user/glob
[user@host glob]$
```

Brace Expansion

Brace expansion is used to generate discretionary strings of characters. Braces contain a comma-separated list of strings, or a sequence expression. The result includes the text preceding or following the brace definition. Brace expansions may be nested, one inside another. Also double-dot syntax (..) expands to a sequence such that **{m..p}** will expand to **m n o p**.

```
[user@host glob]$ echo {Sunday,Monday,Tuesday,Wednesday}.log
Sunday.log Monday.log Tuesday.log Wednesday.log
[user@host glob]$ echo file{1..3}.txt
```

```
file1.txt file2.txt file3.txt
[user@host glob]$ echo file{a..c}.txt
filea.txt fileb.txt filec.txt
[user@host glob]$ echo file{a,b}{1,2}.txt
filea1.txt filea2.txt fileb1.txt fileb2.txt
[user@host glob]$ echo file{a{1,2},b,c}.txt
filea1.txt filea2.txt fileb.txt filec.txt
[user@host glob]$
```

A practical use of brace expansion is to quickly create a number of files or directories.

```
[user@host glob]$ mkdir ../RHEL{6,7,8}
[user@host glob]$ ls ../RHEL*
RHEL6 RHEL7 RHEL8
[user@host glob]$
```

Variable Expansion

A variable acts like a named container that can store a value in memory. Variables make it easy to access and modify the stored data either from the command line or within a shell script.

You can assign data as a value to a variable using the following syntax:

```
[user@host ~]$ VARIABLENAME=value
```

You can use variable expansion to convert the variable name to its value on the command line. If a string starts with a dollar sign ($), then the shell will try to use the rest of that string as a variable name and replace it with whatever value the variable has.

```
[user@host ~]$ USERNAME=operator
[user@host ~]$ echo $USERNAME
operator
```

To help avoid mistakes due to other shell expansions, you can put the name of the variable in curly braces, for example **${VARIABLENAME}**.

```
[user@host ~]$ USERNAME=operator
[user@host ~]$ echo ${USERNAME}
operator
```

Shell variables and ways to use them will be covered in more depth later in this course.

Command Substitution

Command substitution allows the output of a command to replace the command itself on the command line. Command substitution occurs when a command is enclosed in parentheses, and preceded by a dollar sign ($). The **$(*command*)** form can nest multiple command expansions inside each other.

```
[user@host glob]$ echo Today is $(date +%A).
Today is Wednesday.
[user@host glob]$ echo The time is $(date +%M) minutes past $(date +%l%p).
The time is 26 minutes past 11AM.
[user@host glob]$
```

Note

An older form of command substitution uses backticks: `` `command` ``. Disadvantages to the backticks form include: 1) it can be easy to visually confuse backticks with single quote marks, and 2) backticks cannot be nested.

Protecting Arguments from Expansion

Many characters have special meaning in the Bash shell. To keep the shell from performing shell expansions on parts of your command line, you can *quote* and *escape* characters and strings.

The backslash (\) is an escape character in the Bash shell. It will protect the character immediately following it from expansion.

```
[user@host glob]$ echo The value of $HOME is your home directory.
The value of /home/user is your home directory.
[user@host glob]$ echo The value of \$HOME is your home directory.
The value of $HOME is your home directory.
[user@host glob]$
```

In the preceding example, protecting the dollar sign from expansion caused Bash to treat it as a regular character and it did not perform variable expansion on **$HOME**.

To protect longer character strings, single quotes (') or double quotes (") are used to enclose strings. They have slightly different effects. Single quotes stop all shell expansion. Double quotes stop *most* shell expansion.

Use double quotation marks to suppress globbing and shell expansion, but still allow command and variable substitution.

```
[user@host glob]$ myhost=$(hostname -s); echo $myhost
host
[user@host glob]$ echo "***** hostname is ${myhost} *****"
***** hostname is host *****
[user@host glob]$
```

Use single quotation marks to interpret *all* text literally.

```
[user@host glob]$ echo "Will variable $myhost evaluate to $(hostname -s)?"
Will variable myhost evaluate to host?
[user@host glob]$ echo 'Will variable $myhost evaluate to $(hostname -s)?'
Will variable $myhost evaluate to $(hostname -s)?
[user@host glob]$
```

Important

The single quote (') and the command substitution backtick (`) can be easy to confuse, both on the screen and on the keyboard. Using one when you mean to use the other will lead to unexpected shell behavior.

References

bash(1), **cd**(1), **glob**(7), **isalpha**(3), **ls**(1), **path_resolution**(7), and **pwd**(1) man pages

Quiz

Path Name Expansion

Choose the correct answers to the following questions:

1. **Which pattern will match only filenames ending with "b"?**
   a. `b*`
   b. `*b`
   c. `*b*`
   d. `[!b]*`

2. **Which pattern will match only filenames beginning with "b"?**
   a. `b*`
   b. `*b`
   c. `*b*`
   d. `[!b]*`

3. **Which pattern will match only filenames where the first character is not "b"?**
   a. `b*`
   b. `*b`
   c. `*b*`
   d. `[!b]*`

4. **Which pattern will match all filenames containing a "b"?**
   a. `b*`
   b. `*b`
   c. `*b*`
   d. `[!b]*`

5. **Which pattern will match only filenames that contain a number?**
   a. `*#*`
   b. `*[[:digit:]]*`
   c. `*[digit]*`
   d. `[0-9]`

6. **Which pattern will match only filenames that begin with an uppercase letter?**
   a. `^?*`
   b. `^*`
   c. `[upper]*`
   d. `[[:upper:]]*`
   e. `[[CAP]]*`

7. Which pattern will match only filenames at least three characters in length?

a. **???***

b. **???**

c. **\3***

d. **+++***

e. **. . .***

Solution

Path Name Expansion

Choose the correct answers to the following questions:

1. **Which pattern will match only filenames ending with "b"?**
   a. `b*`
   b. **`*b`**
   c. `*b*`
   d. `[!b]*`

2. **Which pattern will match only filenames beginning with "b"?**
   a. **`b*`**
   b. `*b`
   c. `*b*`
   d. `[!b]*`

3. **Which pattern will match only filenames where the first character is not "b"?**
   a. `b*`
   b. `*b`
   c. `*b*`
   d. **`[!b]*`**

4. **Which pattern will match all filenames containing a "b"?**
   a. `b*`
   b. `*b`
   c. **`*b*`**
   d. `[!b]*`

5. **Which pattern will match only filenames that contain a number?**
   a. `*#*`
   b. **`*[[:digit:]]*`**
   c. `*[digit]*`
   d. `[0-9]`

6. **Which pattern will match only filenames that begin with an uppercase letter?**
   a. `^?*`
   b. `^*`
   c. `[upper]*`
   d. **`[[:upper:]]*`**
   e. `[[CAP]]*`

7. Which pattern will match only filenames at least three characters in length?

a. **???***

b. ???

c. \3*

d. +++*

e. . . .*

Lab

Managing Files with Shell Expansion

In this lab, you will create, move, and remove files and folders using a variety of file name matching shortcuts.

Outcomes:

- Increased familiarity and practice with using shell wildcards to locate and reference files.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**.

Important

This is the first Lab exercise in this course. Unlike a Practice, which is a step-by-step guided exercise, a Lab will instruct you to perform a set of tasks and it is up to you to determine how to accomplish each of them based on what you've learned so far.

However, a complete step-by-step solution to the Lab is also provided after the last task. This can be used to check your work, but it can also be useful if you are finding the Lab confusing and just want a step-by-step walkthrough.

Steps

1. Create an initial set of empty practice files to use in this lab.

   This should consist of 12 "recorded video files" with names of the form **tv_season*X*_episode*Y*.ogg**. Replace *X* with the season number (**1** or **2**). Replace *Y* with an episode number (**1** through **6**). This should result in two "seasons" of six "episodes" each.

   Use the **ls** command with a wildcard to check your work by listing all the files with names that start with **tv**.

2. As the author of a successful series of mystery novels, your next bestseller's chapters are being edited for publishing. Create a total of eight files with names **mystery_chapter*X*.odf**. Replace *X* with the numbers **1** through **8**.

   Check your work by listing all the files with names that start with **mys**.

3. From the home directory, create two subdirectories named **season1** and **season2** under the **Videos** directory. Use one command. (Hint: if **Videos** does not exist from an earlier exercise, the **-p** option for **mkdir** will create any missing parent directories if necessary.)

4. Move the appropriate **tv_season*X*_episode*Y*.ogg** files into the **season1** and **season2** subdirectories. Use only two commands, specifying destinations using relative paths.

5. To organize the mystery book chapters, create a two-level directory hierarchy with one command. Create the directory **my_bestseller** under the **Documents** directory, and the directory **chapters** beneath the new **my_bestseller** directory.

6. Using one command, create three more subdirectories directly under the **my_bestseller** directory. Name these subdirectories **editor**, **changes**, and **vacation**. The *create parent* (**-p**) option is not needed since the **my_bestseller** parent directory already exists.

7. Change to the **chapters** directory. Using the tilde (**~**) home directory shortcut to specify the source files, move all book chapters into the **chapters** directory, which is now your current directory. What is the simplest syntax to specify the destination directory?

8. The first two chapters are sent to the editor for review. To remember to not modify these chapters during the review, move those two chapters only to the **editor** directory. Starting from the **chapters** subdirectory, use brace expansion with a range to specify the chapter file names to copy and a relative path for the destination directory.

9. Chapters 7 and 8 will be written while on vacation. Move the files from **chapters** to **vacation**. Use one command, specifying the chapter file names using brace expansion with a list of strings and without using wildcard characters.

10. Change your working directory to **~/Videos/season2**, then copy the first episode of the season to the **vacation** directory.

11. With one **cd** command, change your working directory to **~/Documents/my_bestseller/vacation**. List its files. Return to the **season2** directory using **cd** with its *previous working directory* argument. (This will succeed if the last directory change with **cd** was accomplished with one command rather than several **cd** commands.) Copy the episode 2 file into **vacation**. Return to **vacation** using the **cd** shortcut again.

12. Chapters 5 and 6 may need a plot change. To prevent these changes from modifying original files, copy both files into **changes**. Move up one directory to the parent directory of **vacation**, then use one command from there. Try using square bracket pattern matching to specify which chapter numbers to match in the file name with the copy command.

13. Change your current directory to the **changes** directory.

    Use the **date +%F** command with command substitution to copy **mystery_chapter5.odf** to a new file name which includes the full date. The name should have the form **mystery_chapter5_*YYYY-MM-DD*.odf**.

    Make another copy of **mystery_chapter5.odf**, appending the current timestamp (as the number of seconds since the *epoch*, 1970-01-01 00:00 UTC) to ensure a unique file name. Use command substitution with the **date +%s** command to accomplish this.

14. Delete the **changes** directory in the following way.

    First, delete all of the files in the **changes** directory. The current working directory should be **changes** at this point. Change to its parent directory, because a directory cannot be deleted while it is the current working directory. Try to delete the empty directory using the **rm** command *without* the *recursive* option. This attempt should fail. (If you had used the option, it would succeed.) Finally, use the **rmdir** command to delete the empty directory, which will succeed.

15. When the vacation is over, the **vacation** directory is no longer needed. Delete it using the **rm** command with the *recursive* option.

    When finished, return to the **ec2-user** home directory.

16. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Solution

Managing Files with Shell Expansion

In this lab, you will create, move, and remove files and folders using a variety of file name matching shortcuts.

Outcomes:

- Increased familiarity and practice with using shell wildcards to locate and reference files.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**.

Important

This is the first Lab exercise in this course. Unlike a Practice, which is a step-by-step guided exercise, a Lab will instruct you to perform a set of tasks and it is up to you to determine how to accomplish each of them based on what you've learned so far.

However, a complete step-by-step solution to the Lab is also provided after the last task. This can be used to check your work, but it can also be useful if you are finding the Lab confusing and just want a step-by-step walkthrough.

Steps

1. Create an initial set of empty practice files to use in this lab.

   This should consist of 12 "recorded video files" with names of the form **tv_season*X*_episode*Y*.ogg**. Replace *X* with the season number (**1** or **2**). Replace *Y* with an episode number (**1** through **6**). This should result in two "seasons" of six "episodes" each.

   Use the **ls** command with a wildcard to check your work by listing all the files with names that start with **tv**.

```
[ec2-user@ip-192-0-2-1 ~]$ touch tv_season{1..2}_episode{1..6}.ogg
[ec2-user@ip-192-0-2-1 ~]$ ls tv*
tv_season1_episode1.ogg  tv_season1_episode5.ogg  tv_season2_episode3.ogg
tv_season1_episode2.ogg  tv_season1_episode6.ogg  tv_season2_episode4.ogg
tv_season1_episode3.ogg  tv_season2_episode1.ogg  tv_season2_episode5.ogg
tv_season1_episode4.ogg  tv_season2_episode2.ogg  tv_season2_episode6.ogg
```

2. As the author of a successful series of mystery novels, your next bestseller's chapters are being edited for publishing. Create a total of eight files with names **mystery_chapter*X*.odf**. Replace *X* with the numbers **1** through **8**.

   Check your work by listing all the files with names that start with **mys**.

```
[ec2-user@ip-192-0-2-1 ~]$ touch mystery_chapter{1..8}.odf
[ec2-user@ip-192-0-2-1 ~]$ ls mys*
mystery_chapter1.odf  mystery_chapter4.odf  mystery_chapter7.odf
mystery_chapter2.odf  mystery_chapter5.odf  mystery_chapter8.odf
mystery_chapter3.odf  mystery_chapter6.odf
```

3. From the home directory, create two subdirectories named **season1** and **season2** under the **Videos** directory. Use one command. (Hint: if **Videos** does not exist from an earlier exercise, the **-p** option for **mkdir** will create any missing parent directories if necessary.)

```
[ec2-user@ip-192-0-2-1 ~]$ mkdir -p Videos/season{1..2}
[ec2-user@ip-192-0-2-1 ~]$ ls Videos
season1  season2
```

4. Move the appropriate **tv_season*X*_episode*Y*.ogg** files into the **season1** and **season2** subdirectories. Use only two commands, specifying destinations using relative paths.

```
[ec2-user@ip-192-0-2-1 ~]$ mv tv_season1* Videos/season1
[ec2-user@ip-192-0-2-1 ~]$ mv tv_season2* Videos/season2
[ec2-user@ip-192-0-2-1 ~]$ ls -R Videos
Videos/:
season1  season2

Videos/season1:
tv_season1_episode1.ogg  tv_season1_episode3.ogg  tv_season1_episode5.ogg
tv_season1_episode2.ogg  tv_season1_episode4.ogg  tv_season1_episode6.ogg

Videos/season2:
tv_season2_episode1.ogg  tv_season2_episode3.ogg  tv_season2_episode5.ogg
tv_season2_episode2.ogg  tv_season2_episode4.ogg  tv_season2_episode6.ogg
```

5. To organize the mystery book chapters, create a two-level directory hierarchy with one command. Create the directory **my_bestseller** under the **Documents** directory, and the directory **chapters** beneath the new **my_bestseller** directory.

```
[ec2-user@ip-192-0-2-1 ~]$ mkdir -p Documents/my_bestseller/chapters
[ec2-user@ip-192-0-2-1 ~]$ ls -R Documents
Documents/:
my_bestseller

Documents/my_bestseller:
chapters

Documents/my_bestseller/chapters:
```

6. Using one command, create three more subdirectories directly under the **my_bestseller** directory. Name these subdirectories **editor**, **changes**, and **vacation**. The *create parent* (**-p**) option is not needed since the **my_bestseller** parent directory already exists.

```
[ec2-user@ip-192-0-2-1 ~]$ mkdir Documents/my_bestseller/{editor,changes,vacation}
[ec2-user@ip-192-0-2-1 ~]$ ls -R Documents
Documents/:
my_bestseller

Documents/my_bestseller:
changes  chapters  editor  vacation

Documents/my_bestseller/changes:

Documents/my_bestseller/chapters:

Documents/my_bestseller/editor:

Documents/my_bestseller/vacation:
```

7. Change to the **chapters** directory. Using the tilde (**~**) home directory shortcut to specify the source files, move all book chapters into the **chapters** directory, which is now your current directory. What is the simplest syntax to specify the destination directory?

```
[ec2-user@ip-192-0-2-1 ~]$ cd Documents/my_bestseller/chapters
[ec2-user@ip-192-0-2-1 chapters]$ mv ~/mystery_chapter* .
[ec2-user@ip-192-0-2-1 chapters]$ ls
mystery_chapter1.odf  mystery_chapter4.odf  mystery_chapter7.odf
mystery_chapter2.odf  mystery_chapter5.odf  mystery_chapter8.odf
mystery_chapter3.odf  mystery_chapter6.odf
```

8. The first two chapters are sent to the editor for review. To remember to not modify these chapters during the review, move those two chapters only to the **editor** directory. Starting from the **chapters** subdirectory, use brace expansion with a range to specify the chapter file names to copy and a relative path for the destination directory.

```
[ec2-user@ip-192-0-2-1 chapters]$ mv mystery_chapter{1..2}.odf ../editor
[ec2-user@ip-192-0-2-1 chapters]$ ls
mystery_chapter3.odf  mystery_chapter5.odf  mystery_chapter7.odf
mystery_chapter4.odf  mystery_chapter6.odf  mystery_chapter8.odf
[ec2-user@ip-192-0-2-1 chapters]$ ls ../editor
mystery_chapter1.odf  mystery_chapter2.odf
```

9. Chapters 7 and 8 will be written while on vacation. Move the files from **chapters** to **vacation**. Use one command, specifying the chapter file names using brace expansion with a list of strings and without using wildcard characters.

```
[ec2-user@ip-192-0-2-1 chapters]$ mv mystery_chapter{7,8}.odf ../vacation
[ec2-user@ip-192-0-2-1 chapters]$ ls
mystery_chapter3.odf  mystery_chapter5.odf
mystery_chapter4.odf  mystery_chapter6.odf
[ec2-user@ip-192-0-2-1 chapters]$ ls ../vacation
mystery_chapter7.odf  mystery_chapter8.odf
```

10. Change your working directory to **~/Videos/season2**, then copy the first episode of the season to the **vacation** directory.

```
[ec2-user@ip-192-0-2-1 chapters]$ cd ~/Videos/season2
[ec2-user@ip-192-0-2-1 season2]$ cp *episode1.ogg ~/Documents/my_bestseller/
vacation
```

11. With one **cd** command, change your working directory to **~/Documents/my_bestseller/vacation**. List its files. Return to the **season2** directory using **cd** with its *previous working directory* argument. (This will succeed if the last directory change with **cd** was accomplished with one command rather than several **cd** commands.) Copy the episode 2 file into **vacation**. Return to **vacation** using the **cd** shortcut again.

```
[ec2-user@ip-192-0-2-1 season2]$ cd ~/Documents/my_bestseller/vacation
[ec2-user@ip-192-0-2-1 vacation]$ ls
mystery_chapter7.odf  mystery_chapter8.odf  tv_season2_episode1.ogg
[ec2-user@ip-192-0-2-1 vacation]$ cd -
/home/ec2-user/Videos/season2
[ec2-user@ip-192-0-2-1 season2]$ cp *episode2.ogg ~/Documents/my_bestseller/
vacation
[ec2-user@ip-192-0-2-1 season2]$ cd -
/home/ec2-user/Documents/my_bestseller/vacation
[ec2-user@ip-192-0-2-1 vacation]$ ls
mystery_chapter7.odf  tv_season2_episode1.ogg
mystery_chapter8.odf  tv_season2_episode2.ogg
```

12. Chapters 5 and 6 may need a plot change. To prevent these changes from modifying original files, copy both files into **changes**. Move up one directory to the parent directory of **vacation**, then use one command from there. Try using square bracket pattern matching to specify which chapter numbers to match in the file name with the copy command.

```
[ec2-user@ip-192-0-2-1 vacation]$ cd ..
[ec2-user@ip-192-0-2-1 my_bestseller]$ cp chapters/mystery_chapter[56].odf changes
[ec2-user@ip-192-0-2-1 my_bestseller]$ ls chapters
mystery_chapter3.odf  mystery_chapter5.odf
mystery_chapter4.odf  mystery_chapter6.odf
[ec2-user@ip-192-0-2-1 my_bestseller]$ ls changes
mystery_chapter5.odf  mystery_chapter6.odf
```

13. Change your current directory to the **changes** directory.

    Use the **date +%F** command with command substitution to copy **mystery_chapter5.odf** to a new file name which includes the full date. The name should have the form **mystery_chapter5_*YYYY-MM-DD*.odf**.

    Make another copy of **mystery_chapter5.odf**, appending the current timestamp (as the number of seconds since the *epoch*, 1970-01-01 00:00 UTC) to ensure a unique file name. Use command substitution with the **date +%s** command to accomplish this.

Important

The following example shows **ec2-user** using the **** escape followed by a **Return** on the command line to start a new line in the terminal without submitting the command to the shell. The **>** prompt will appear to indicate that the next line is part of the same command. This is being done to make it more clear what the command line arguments should look like (to ensure they don't wrap in the middle of the argument on screen.) You could also type the whole command after each prompt without using the **** escape followed by **Return** in the middle of the command.

```
[ec2-user@ip-192-0-2-1 my_bestseller]$ cd changes
[ec2-user@ip-192-0-2-1 changes]$ cp mystery_chapter5.odf \
> mystery_chapter5_$(date +%F).odf
[ec2-user@ip-192-0-2-1 changes]$ cp mystery_chapter5.odf \
> mystery_chapter5_$(date +%s).odf
[ec2-user@ip-192-0-2-1 changes]$ ls
mystery_chapter5_1595594872.odf  mystery_chapter5.odf
mystery_chapter5_2020-07-24.odf  mystery_chapter6.odf
```

14. Delete the **changes** directory in the following way.

    First, delete all of the files in the **changes** directory. The current working directory should be **changes** at this point. Change to its parent directory, because a directory cannot be deleted while it is the current working directory. Try to delete the empty directory using the **rm** command *without* the *recursive* option. This attempt should fail. (If you had used the option, it would succeed.) Finally, use the **rmdir** command to delete the empty directory, which will succeed.

```
[ec2-user@ip-192-0-2-1 changes]$ rm mystery*
[ec2-user@ip-192-0-2-1 changes]$ cd ..
[ec2-user@ip-192-0-2-1 my_bestseller]$ rm changes
rm: cannot remove 'changes': Is a directory
[ec2-user@ip-192-0-2-1 my_bestseller]$ rmdir changes
[ec2-user@ip-192-0-2-1 my_bestseller]$ ls
chapters  editor  vacation
```

15. When the vacation is over, the **`vacation`** directory is no longer needed. Delete it using the **`rm`** command with the *recursive* option.

    When finished, return to the **`ec2-user`** home directory.

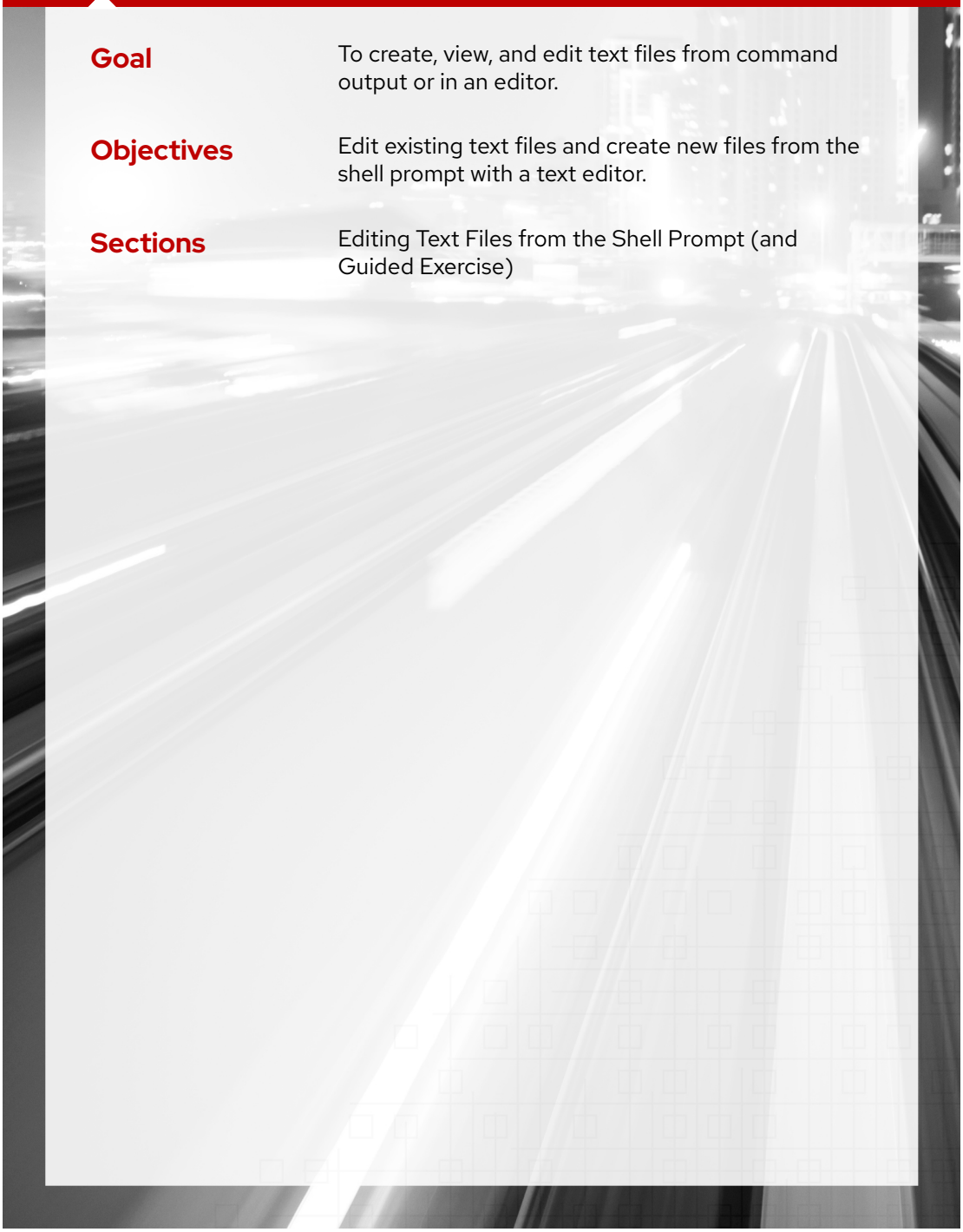
```
[ec2-user@ip-192-0-2-1 my_bestseller]$ rm -r vacation
[ec2-user@ip-192-0-2-1 my_bestseller]$ ls
chapters  editor
[ec2-user@ip-192-0-2-1 my_bestseller]$ cd
[ec2-user@ip-192-0-2-1 ~]$
```

16. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Chapter 4

Creating, Viewing, and Editing Text Files

Goal	To create, view, and edit text files from command output or in an editor.
Objectives	Edit existing text files and create new files from the shell prompt with a text editor.
Sections	Editing Text Files from the Shell Prompt (and Guided Exercise)



Editing Text Files from the Shell Prompt

Objectives

After completing this section, you should be able to create and edit text files from the command line using the **vim** editor.

Editing Files with Vim

A key design principle of Linux is that information and configuration settings are commonly stored in text-based files. These files can be structured in various ways, as lists of settings, in INI-like formats, as structured XML or YAML, and so on. However, the advantage of text files is that they can be viewed and edited using any simple text editor.

Vim is an improved version of the **vi** editor distributed with Linux and UNIX systems. Vim is highly configurable and efficient for practiced users, including such features as split screen editing, color formatting, and highlighting for editing text.

Why Learn Vim?

You should know how to use at least one text editor that can be used from a text-only shell prompt. If you do, you can edit text-based configuration files from a terminal window, or from remote logins through **ssh** or the Web Console. Then you do not need access to a graphical desktop in order to edit files on a server, and in fact that server might not need to run a graphical desktop environment at all.

But then, why learn Vim instead of other possible options? The key reason is that Vim is almost always installed on a server, if any text editor is present. This is because **vi** was specified by the POSIX standard that Linux and many other UNIX-like operating systems comply with in large part.

In addition, Vim is often used as the **vi** implementation on other common operating systems or distributions. For example, macOS currently includes a lightweight installation of Vim by default. So Vim skills learned for Linux might also help you get things done elsewhere.

Starting Vim

Vim may be installed in Red Hat Enterprise Linux in two different ways. This can affect the features and Vim commands available to you.

Your server might only have the *vim-minimal* package installed. This is a very lightweight installation that includes only the core feature set and the basic **vi** command. In this case, you can open a file for editing with **vi *filename***, and all the core features discussed in this section will be available to you.

Alternatively, your server might have the *vim-enhanced* package installed. This provides a much more comprehensive set of features, an on-line help system, and a tutorial program. In order to start Vim in this enhanced mode, you use the **vim** command.

```
[user@host ~]$ vim filename
```

Either way, the core features that we will discuss in this section will work with both commands.

Note

If *vim-enhanced* is installed, regular users will have a shell alias set so that if they run the **`vi`** command, they will automatically get the **`vim`** command instead. This does not apply to **`root`** and other users with UIDs below 200 (which are used by system services).

If you are editing files as the **`root`** user and you expect **`vi`** to run in enhanced mode, this can be a surprise. Likewise, if *vim-enhanced* is installed and a regular user wants the simple **`vi`** for some reason, they might need to use **`\vi`** to override the alias temporarily.

Advanced users can use **`\vi --version`** and **`vim --version`** to compare the feature sets of the two commands.

Vim Operating Modes

An unusual characteristic of Vim is that it has several *modes* of operation, including *command mode*, *extended command mode*, *edit mode*, and *visual mode*. Depending on the mode, you may be issuing commands, editing text, or working with blocks of text. As a new Vim user, you should always be aware of your current mode as keystrokes have different effects in different modes.

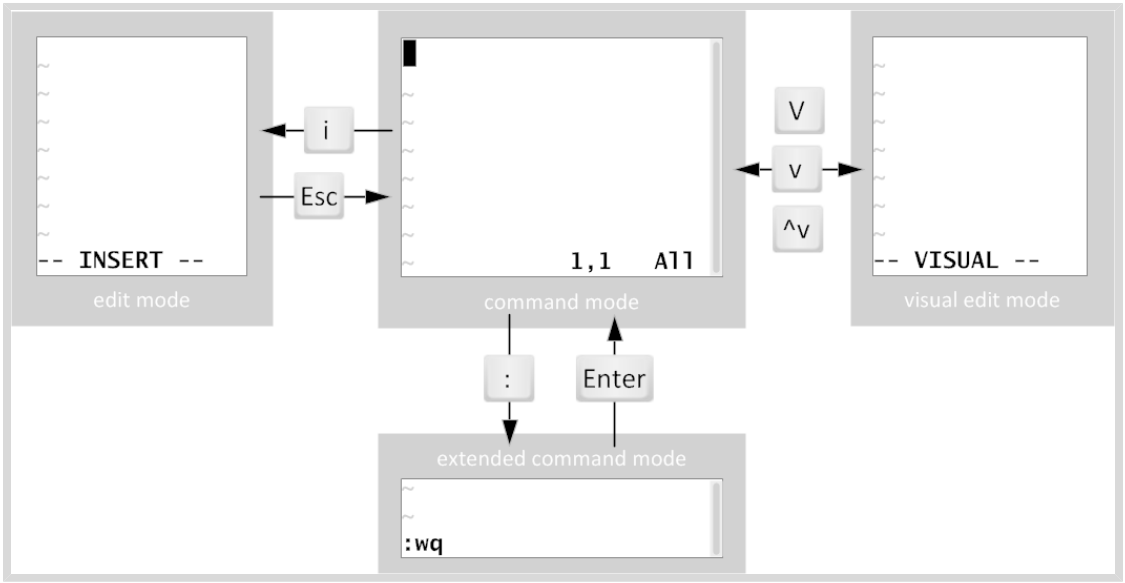


Figure 4.1: Moving between Vim modes

When you first open Vim, it starts in *command mode*, which is used for navigation, cut and paste, and other text manipulation. Enter each of the other modes with single character keystrokes to access specific editing functionality:

- An **`i`** keystroke enters *insert mode*, where all text typed becomes file content. Pressing **`Esc`** returns to command mode.
- A **`v`** keystroke enters *visual mode*, where multiple characters may be selected for text manipulation. Use **`Shift`**+**`V`** for multiline and **`Ctrl`**+**`V`** for block selection. The same keystroke used to enter visual mode (**`v`**, **`Shift`**+**`V`** or **`Ctrl`**+**`V`**) is used to exit.
- The **`:`** keystroke begins *extended command mode* for tasks such as writing the file (to save it), and quitting the Vim editor.

Note

If you are not sure what mode Vim is in, you can try pressing **Esc** a few times to get back into command mode. Pressing **Esc** in command mode is harmless, so a few extra key presses are okay.

The Minimum, Basic Vim Workflow

Vim has efficient, coordinated keystrokes for advanced editing tasks. Although considered useful with practice, Vim's capabilities can overwhelm new users.

The **i** key puts Vim into insert mode. All text entered after this is treated as file contents until you exit insert mode. The **Esc** key exits insert mode and returns Vim to command mode. The **u** key will undo the most recent edit. Press the **x** key to delete a single character. The **:w** command writes (saves) the file and remains in command mode for more editing. The **:wq** command writes (saves) the file and quits Vim. The **:q!** command quits Vim, discarding all file changes since the last write. The Vim user must learn these commands to accomplish any editing task.

Rearranging Existing Text

In Vim, copy and paste is known as *yank and put*, using command characters **y** and **p**. Begin by positioning the cursor on the first character to be selected, and then enter visual mode. Use the arrow keys to expand the visual selection. When ready, press **y** to *yank* the selection into memory. Position the cursor at the new location, and then press **p** to *put* the selection at the cursor.

Visual Mode in Vim

Visual mode is a great way to highlight and manipulate text. There are three keystrokes:

- Character mode: **v**
- Line mode: **Shift**+**v**
- Block mode: **Ctrl**+**v**

Character mode highlights sentences in a block of text. The word **VISUAL** will appear at the bottom of the screen. Press **v** to enter visual character mode. **Shift**+**v** enters line mode. **VISUAL LINE** will appear at the bottom of the screen.

Visual block mode is perfect for manipulating data files. From the cursor, press the **Ctrl**+**v** to enter visual block. **VISUAL BLOCK** will appear at the bottom of the screen. Use the arrow keys to highlight the section to change.

Note

Vim has a lot of capabilities, but you should master the basic workflow first. You do not need to quickly understand the entire editor and its capabilities. Get comfortable with those basics through practice and then you can expand your Vim vocabulary by learning additional Vim commands (keystrokes).

The exercise for this section will introduce you to the **vimtutor** command. This tutorial, which ships with *vim-enhanced*, is an excellent way to learn the core functionality of Vim.

References

vim(1) man page

The **:help** command in **vim** (if the *vim-enhanced* package is installed).

Vim the editor
http://www.vim.org/

Getting Started with Vim visual mode
https://opensource.com/article/19/2/getting-started-vim-visual-mode

Guided Exercise

Editing Files with Vim

In this exercise, you will use the Vim tutorial bundled with the editor to practice entry-level **vim** editor techniques.

Outcomes:

Basic competency in using the **vim** text editor, and knowledge of the **vimtutor** tutorial for future learning and practice.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**.

Steps

1. This exercise will use the existing Vim tutorial bundled with the Vim editor. Your Amazon EC2 instance probably has a basic version of **vim** installed, but not the fully enhanced version of **vim** which includes the tutorial and help files. Run the command **sudo yum -y install vim-enhanced** to make sure that it is installed. Software installation will be discussed later in this course.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo yum -y install vim-enhanced
```

2. Run **vimtutor**. Read the Welcome screen and perform *Lesson 1.1*.

```
[ec2-user@ip-192-0-2-1 ~]$ vimtutor
```

   In the lecture, only keyboard arrow keys were used for navigation. In **vi**'s early years, users could not rely on working keyboard mappings for arrow keys. Therefore, **vi** was designed with commands using only standard character keys, such as the conveniently grouped **h**, **j**, **k**, and **l**. Here is one way to remember them:

   hang *back*, **j**ump *down*, **k**ick *up*, **l**eap *forward*.

3. Return to the **vimtutor** window. Perform *Lesson 1.2*.

   This early lesson teaches how to quit without having to keep an unwanted file change. All changes are lost, but this is better than leaving a critical file in an incorrect state.

4. Return to the **vimtutor** window. Perform *Lesson 1.3*.

   Vim has faster, more efficient keystrokes to delete an exact amount of words, lines, sentences, and paragraphs. However, any editing job *can* be accomplished using only **x** for single-character deletion.

5. Return to the **vimtutor** window. Perform *Lesson 1.4*.

   The minimum required keystrokes are for entering and leaving edit mode, arrow keys, and deleting. For most edit tasks, the first key pressed is **i**.

6. (*Optional*) Return to the **`vimtutor`** window. Perform *Lesson 1.5*.

   In the lecture, only the **`i`** (*insert*) command was taught as the keystroke to enter edit mode. This vimtutor lesson demonstrates that other keystrokes are available to change the cursor placement when insert mode is entered. However, once in insert mode, all text typed is still file content.

7. Return to the **`vimtutor`** window. Perform *Lesson 1.6*.

   Save the file by **`w`**riting and **`q`**uitting. This is the last lesson for the minimum *required* keystrokes to be able to accomplish any editing task.

8. Return to the **`vimtutor`** window. Finish by reading the *Lesson 1 Summary*.

   There are six more multi-step lessons in **`vimtutor`**. None are assigned as further lessons for this course, but feel free to use **`vimtutor`** on your own to learn more about Vim.

9. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Chapter 5

Managing Local Linux Users and Groups

Goal

To manage local Linux users and groups and administer local password policies.

Objectives

- Explain the role of users and groups on a Linux system and how they are understood by the computer.
- Run commands as the superuser to administer a Linux system.
- Create, modify, lock, and delete locally defined user accounts.
- Create, modify, and delete locally defined group accounts.

Sections

- Users and Groups (and Quiz)
- Gaining Superuser Access (and Guided Exercise)
- Managing Local User Accounts (and Guided Exercise)
- Managing Local Group Accounts (and Guided Exercise)

Lab

Managing Local Linux Users and Groups

Users and Groups

Objectives

After completing this section, you should be able to describe the purpose of users and groups on a Linux system.

What is a User?

A *user* account is used to provide security boundaries between different people and programs that can run commands.

Users have *user names* to identify them to human users and make them easier to work with. Internally, the system distinguishes user accounts by the unique identification number assigned to them, the *user ID* or *UID*. If a user account is used by humans, it will generally be assigned a secret *password* that the user will use to prove that they are the actual authorized user when logging in.

User accounts are fundamental to system security. Every process (running program) on the system runs as a particular user. Every file has a particular user as its owner. File ownership helps the system enforce access control for users of the files. The user associated with a running process determines the files and directories accessible to that process.

There are three main types of user account: the *superuser*, *system users*, and *regular users*.

- The *superuser* account is for administration of the system. The name of the superuser is **`root`** and the account has UID 0. The superuser has full access to the system.
- The system has *system user* accounts which are used by processes that provide supporting services. These processes, or *daemons*, usually do not need to run as the superuser. They are assiged non-privileged accounts that allow them to secure their files and other resources from each other and from regular users on the system. Users do not interactively log in using a system user account.
- Most users have *regular user* accounts which they use for their day-to-day work. Like system users, regular users have limited access to the system.

You can use the **`id`** command to show information about the currently logged-in user.

```
[user01@host ~]$ id
uid=1000(user01) gid=1000(user01) groups=1000(user01)
 context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

To view basic information about another user, pass the username to the **`id`** command as an argument.

```
[user01@host]$ id user02
uid=1002(user02) gid=1001(user02) groups=1001(user02)
 context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

To view the owner of a file use the **`ls -l`** command. To view the owner of a directory use the **`ls -ld`** command. In the following output, the third column shows the username.

```
[user01@host ~]$ ls -l file1
-rw-rw-r--. 1 user01 user01 0 Feb  5 11:10 file1
[user01@host]$ ls -ld dir1
drwxrwxr-x. 2 user01 user01 6 Feb  5 11:10 dir1
```

To view process information, use the **ps** command. The default is to show only processes in the current shell. Add the **a** option to view all processes with a terminal. To view the user associated with a process, include the **u** option. In the following output, the first column shows the username.

```
[user01@host]$ ps -au
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       777  0.0  0.0 225752  1496 tty1     Ss+  11:03   0:00 /sbin/agetty -o -
p -- \u --noclear tty1 linux
root       780  0.0  0.1 225392  2064 ttyS0    Ss+  11:03   0:00 /sbin/agetty -o -
p -- \u --keep-baud 115200,38400,9600
user01      1207  0.0  0.2 234044  5104 pts/0    Ss   11:09   0:00 -bash
user01      1319  0.0  0.2 266904  3876 pts/0    R+   11:33   0:00 ps au
```

The output of the preceding command displays users by name, but internally the operating system uses the UIDs to track users. The mapping of usernames to UIDs is defined in databases of account information. By default, systems use the **/etc/passwd** file to store information about local users.

Each line in the **/etc/passwd** file contains information about one user. It is divided up into seven colon-separated fields. Here is an example of a line from **/etc/passwd**:

```
❶user01:❷x:❸1000:❹1000:❺User One:❻/home/user01:❼/bin/bash
```

❶ Username for this user (**user01**).
❷ The user's password used to be stored here in encrypted format. That has been moved to the **/etc/shadow** file, which will be covered later. This field should always be **x**.
❸ The UID number for this user account (**1000**).
❹ The GID number for this user account's primary group (**1000**). Groups will be discussed later in this section.
❺ The real name for this user (**User One**).
❻ The home directory for this user (**/home/user01**). This is the initial working directory when the shell starts and contains the user's data and configuration settings.
❼ The default shell program for this user, which runs on login (**/bin/bash**). For a regular user, this is normally the program that provides the user's command-line prompt. A system user might use **/sbin/nologin** if interactive logins are not allowed for that user.

What is a Group?

A group is a collection of users that need to share access to files and other system resources. Groups can be used to grant access to files to a set of users instead of just a single user.

Like users, groups have *group names* to make them easier to work with. Internally, the system distinguishes groups by the unique identification number assigned to them, the *group ID* or *GID*.

The mapping of group names to GIDs is defined in databases of group account information. By default, systems use the **/etc/group** file to store information about local groups.

Each line in the **/etc/group** file contains information about one group. Each group entry is divided into four colon-separated fields. Here is an example of a line from **/etc/group**:

```
1group01:2x:310000:4user01,user02,user03
```

1. Group name for this group (**group01**).
2. Obsolete group password field. This field should always be **x**.
3. The GID number for this group (**10000**).
4. A list of users who are members of this group as a supplementary group (**user01**, **user02**, **user03**). Primary (or default) and supplementary groups are discussed later in this section.

Primary Groups and Supplementary Groups

Every user has exactly one primary group. For local users, this is the group listed by GID number in the **/etc/passwd** file. By default, this is the group that will own new files created by the user.

Normally, when you create a new regular user, a new group with the same name as that user is created. That group is used as the primary group for the new user, and that user is the only member of this *User Private Group*. It turns out that this helps make management of file permissions simpler, which will be discussed later in this course.

Users may also have *supplementary groups*. Membership in supplementary groups is determined by the **/etc/group** file. Users are granted access to files based on whether any of their groups have access. It doesn't matter if the group or groups that have access are primary or supplementary for the user.

For example, if the user **user01** has a primary group **user01** and supplementary groups **wheel** and **webadmin**, then that user can read files readable by any of those three groups.

The **id** command can also be used to find out about group membership for a user.

```
[user03@host ~]$ id
uid=1003(user03) gid=1003(user03) groups=1003(user03),10(wheel),10000(group01)
 context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

In the preceding example, **user03** has the group **user03** as their primary group (**gid**). The **groups** item lists all groups for this user, and other than the primary group **user03**, the user has groups **wheel** and **group01** as supplementary groups.

References

id(1), **passwd**(5), and **group**(5) man pages

info libc (*GNU C Library Reference Manual*)

- Section 30: Users and groups

(Note that the *glibc-devel* package must be installed for this info node to be available.)

Quiz

User and Group Concepts

Choose the correct answer to the following questions:

1. Which item represents a number that identifies the user at the most fundamental level?

a. primary user
b. UID
c. GID
d. username

2. Which item represents the program that provides the user's command-line prompt?

a. primary shell
b. home directory
c. login shell
d. command name

3. Which item or file represents the location of the local group information?

a. home directory
b. **`/etc/passwd`**
c. **`/etc/GID`**
d. **`/etc/group`**

4. Which item or file represents the location of the user's personal files?

a. home directory
b. login shell
c. **`/etc/passwd`**
d. **`/etc/group`**

5. Which item represents a number that identifies the group at the most fundamental level?

a. primary group
b. UID
c. GID
d. groupid

6. Which item or file represents the location of the local user account information?

a. home directory
b. **`/etc/passwd`**
c. **`/etc/UID`**
d. **`/etc/group`**

7. What is the fourth field of the `/etc/passwd` file?

a. home directory
b. UID
c. login shell
d. primary group

Solution

User and Group Concepts

Choose the correct answer to the following questions:

1. Which item represents a number that identifies the user at the most fundamental level?

a. primary user
b. UID
c. GID
d. username

2. Which item represents the program that provides the user's command-line prompt?

a. primary shell
b. home directory
c. login shell
d. command name

3. Which item or file represents the location of the local group information?

a. home directory
b. `/etc/passwd`
c. `/etc/GID`
d. **`/etc/group`**

4. Which item or file represents the location of the user's personal files?

a. home directory
b. login shell
c. `/etc/passwd`
d. `/etc/group`

5. Which item represents a number that identifies the group at the most fundamental level?

a. primary group
b. UID
c. GID
d. groupid

6. Which item or file represents the location of the local user account information?

a. home directory
b. **`/etc/passwd`**
c. `/etc/UID`
d. `/etc/group`

7. What is the fourth field of the `/etc/passwd` file?

a. home directory

b. UID

c. login shell

d. primary group

Gaining Superuser Access

Objectives

After completing this section, you will be able to switch to the superuser account to manage a Linux system, and grant other users superuser access through the **sudo** command.

The Superuser

Most operating systems have some sort of *superuser*, a user that has all power over the system. In Red Hat Enterprise Linux this is the **root** user. This user has the power to override normal privileges on the file system, and is used to manage and administer the system. To perform tasks such as installing or removing software and to manage system files and directories, users must escalate their privileges to the **root** user.

The **root** user only among normal users can control most devices, but there are a few exceptions. For example, normal users can control removable devices, such as USB devices. Thus, normal users can add and remove files and otherwise manage a removable device, but only **root** can manage "fixed" hard drives by default.

This unlimited privilege, however, comes with responsibility. The **root** user has unlimited power to damage the system: remove files and directories, remove user accounts, add back doors, and so on. If the **root** user's account is compromised, someone else would have administrative control of the system. Throughout this course, administrators are encouraged to log in as a normal user and escalate privileges to **root** only when needed.

The **root** account on Linux is roughly equivalent to the local Administrator account on Microsoft Windows. In Linux, most system administrators log in to the system as an unprivileged user and use various tools to temporarily gain **root** privileges.

Warning

One common practice on Microsoft Windows in the past was for the local **Administrator** user to log in directly to perform system administrator duties. Although this is possible on Linux, Red Hat recommends that system administrators do not log in directly as **root**. Instead, system administrators should log in as a normal user and use other mechanisms (**su**, **sudo**, or PolicyKit, for example) to temporarily gain superuser privileges.

By logging in as the superuser, the entire desktop environment unnecessarily runs with administrative privileges. In that situation, any security vulnerability which would normally only compromise the user account has the potential to compromise the entire system.

Switching Users

The **su** command allows users to switch to a different user account. If you run **su** from a regular user account, you will be prompted for the password of the account to which you want to switch. When **root** runs **su**, you do not need to enter the user's password.

```
[user01@host ~]$ su - user02
Password:
[user02@host ~]$
```

If you omit the user name, the **su** or **su -** command attempts to switch to **root** by default.

```
[user01@host ~]$ su -
Password:
[root@host ~]#
```

The command **su** starts a *non-login shell*, while the command **su -** (with the dash option) starts a *login shell*. The main distinction between the two commands is that **su -** sets up the shell environment as if it were a new login as that user, while **su** just starts a shell as that user, but uses the original user's environment settings.

In most cases, administrators should run **su -** to get a shell with the target user's normal environment settings. For more information, see the **bash**(1) man page.

Note

The **su** command is most frequently used to get a command-line interface (shell prompt) which is running as another user, typically **root**. However, with the **-c** option, it can be used like the Windows utility **runas** to run an arbitrary program as another user. Run **info su** to view more details.

Running Commands with Sudo

In some cases, the **root** user's account may not have a valid password at all for security reasons. In this case, users cannot log in to the system as **root** directly with a password, and **su** cannot be used to get an interactive shell. One tool that can be used to get **root** access in this case is **sudo**.

Unlike **su**, **sudo** normally requires users to enter their own password for authentication, not the password of the user account they are trying to access. That is, users who use **sudo** to run commands as **root** do not need to know the **root** password. Instead, they use their own passwords to authenticate access.

Additionally, **sudo** can be configured to allow specific users to run any command as some other user, or only some commands as that user.

For example, when **sudo** is configured to allow the **user01** user to run the command **usermod** as **root**, **user01** could run the following command to lock or unlock a user account:

```
[user01@host ~]$ sudo usermod -L user02
[sudo] password for user01:
[user01@host ~]$ su - user02
Password:
su: Authentication failure
[user01@host ~]$
```

If a user tries to run a command as another user, and the **sudo** configuration does not permit it, the command will be blocked, the attempt will be logged, and by default an email will be sent to the **root** user.

```
[user02@host ~]$ sudo tail /var/log/secure
[sudo] password for user02:
user02 is not in the sudoers file.  This incident will be reported.
[user02@host ~]$
```

One additional benefit to using **sudo** is that all commands executed are logged by default to **/var/log/secure**.

```
[user01@host ~]$ sudo tail /var/log/secure
...output omitted...
Feb  6 20:45:46 host sudo[2577]:  user01 : TTY=pts/0 ; PWD=/home/user01 ;
 USER=root ; COMMAND=/sbin/usermod -L user02
...output omitted...
```

In Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux 8, all members of the **wheel** group can use **sudo** to run commands as any user, including **root**. The user is prompted for their own password. This is a change from Red Hat Enterprise Linux 6 and earlier, where users who were members of the **wheel** group did not get this administrative access by default.

Warning

RHEL 6 did not grant the **wheel** group any special privileges by default. Sites that have been using this group for a non-standard purpose might be surprised when RHEL 7 and RHEL 8 automatically grants all members of **wheel** full **sudo** privileges. This could lead to unauthorized users getting administrative access to RHEL 7 and RHEL 8 systems.

Historically, UNIX-like systems use membership in the **wheel** group to grant or control superuser access.

Getting an Interactive Root Shell with Sudo

If there is a nonadministrative user account on the system that can use **sudo** to run the **su** command, you can run **sudo su -** from that account to get an interactive **root** user shell. This works because **sudo** will run **su -** as **root**, and **root** does not need to enter a password to use **su**.

Another way to access the **root** account with **sudo** is to use the **sudo -i** command. This will switch to the **root** account and run that user's default shell (usually **bash**) and associated shell login scripts. If you just want to run the shell, you can use the **sudo -s** command.

For example, an administrator might get an interactive shell as **root** on an AWS EC2 instance by using SSH public-key authentication to log in as the normal user **ec2-user**, and then by running **sudo -i** to get the **root** user's shell.

```
[ec2-user@host ~]$ sudo -i
[sudo] password for ec2-user:
[root@host ~]#
```

The **sudo su -** command and **sudo -i** do not behave exactly the same. This will be discussed briefly at the end of the section.

Configuring Sudo

The main configuration file for **sudo** is **/etc/sudoers**. To avoid problems if multiple administrators try to edit it at the same time, it should only be edited with the special **visudo** command.

For example, the following line from the **/etc/sudoers** file enables **sudo** access for members of group **wheel**.

```
%wheel        ALL=(ALL)       ALL
```

In this line, **%wheel** is the user or group to whom the rule applies. A **%** specifies that this is a group, group **wheel**. The **ALL=(ALL)** specifies that on any host that might have this file, **wheel** can run any command. The final **ALL** specifies that **wheel** can run those commands as any user on the system.

By default, **/etc/sudoers** also includes the contents of any files in the **/etc/sudoers.d** directory as part of the configuration file. This allows an administrator to add **sudo** access for a user simply by putting an appropriate file in that directory.

Note

Using supplementary files under the **/etc/sudoers.d** directory is convenient and simple. You can enable or disable **sudo** access simply by copying a file into the directory or removing it from the directory.

In this course, you will create and remove files in the **/etc/sudoers.d** directory to configure **sudo** access for users and groups.

To enable full **sudo** access for the user **user01**, you could create **/etc/sudoers.d/user01** with the following content:

```
user01  ALL=(ALL)  ALL
```

To enable full **sudo** access for the group **group01**, you could create **/etc/sudoers.d/group01** with the following content:

```
%group01  ALL=(ALL)  ALL
```

It is also possible to set up **sudo** to allow a user to run commands as another user without entering their password:

```
ansible  ALL=(ALL)  NOPASSWD:ALL
```

While there are obvious security risks to granting this level of access to a user or group, it is frequently used with cloud instances, virtual machines, and provisioning systems to help configure servers. The account with this access must be carefully protected and might require SSH public-key authentication in order for a user on a remote system to access it at all.

For example, the official AMI for Red Hat Enterprise Linux in the Amazon Web Services Marketplace ships with the **root** and the **ec2-user** users' passwords locked. The **ec2-user** user account is set up to allow remote interactive access through SSH public-key authentication. The

user **ec2-user** can also run any command as **root** without a password because the last line of the AMI's **/etc/sudoers** file is set up as follows:

```
ec2-user  ALL=(ALL)  NOPASSWD: ALL
```

The requirement to enter a password for **sudo** can be re-enabled or other changes may be made to tighten security as part of the process of configuring the system.

Note

In this course, you may see **sudo su -** used instead of **sudo -i**. Both commands work, but there are some subtle differences between them.

The **sudo su -** command sets up the **root** environment exactly like a normal login because the **su -** command ignores the settings made by **sudo** and sets up the environment from scratch.

The default configuration of the **sudo -i** command actually sets up some details of the **root** user's environment differently than a normal login. For example, it sets the **PATH** environment variable slightly differently. This affects where the shell will look to find commands.

You can make **sudo -i** behave more like **su -** by editing **/etc/sudoers** with **visudo**. Find the line

```
Defaults    secure_path = /sbin:/bin:/usr/sbin:/usr/bin
```

and replace it with the following two lines:

```
Defaults      secure_path = /usr/local/bin:/usr/bin
Defaults>root secure_path = /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
```

For most purposes, this is not a major difference. However, for consistency of **PATH** settings on systems with the default **/etc/sudoers** file, the authors of this course use **sudo -i** in examples and exercises.

References

su(1), **sudo**(8), **visudo**(8) and **sudoers**(5) man pages

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

- Section 30.2: The Persona of a Process

(Note that the *glibc-devel* package must be installed for this info node to be available.)

Guided Exercise

Running Commands as root

In this exercise, you will practice running commands as **root**.

Outcomes

- Use the **sudo** command to run other commands as **root**.
- Use **sudo** to run **su** to get an interactive shell as **root** when the superuser does not have a valid password.
- Explain how **su** and **su -** can affect the shell environment through not running or running login scripts.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**.

It is assumed that the AMI that you are using is pre-configured to allow the **ec2-user** to run any command as any user without a password using **sudo**.

Steps

1. Explore the characteristics of the **ec2-user** shell environment.

1.1. View the current user and group information and display the current working directory.

```
[ec2-user@ip-192-0-2-1 ~]$ id
uid=1000(ec2-user) gid=1000(ec2-user) groups=1000(ec2-user),4(adm),190(systemd-
journal) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[ec2-user@ip-192-0-2-1 ~]$ pwd
/home/ec2-user
```

1.2. View the environment variables which specify the user's home directory and the locations searched for executable files.

```
[ec2-user@ip-192-0-2-1 ~]$ echo $HOME
/home/ec2-user
[ec2-user@ip-192-0-2-1 ~]$ echo $PATH
/home/ec2-user/.local/bin:/home/ec2-user/bin:/usr/local/bin:/usr/bin:/usr/local/
sbin:/usr/sbin
```

2. Switch to **root** by using **sudo** to run **su** without the dash, and explore the characteristics of the new shell environment.

2.1. Become the **root** user at the shell prompt.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su
[root@ip-192-0-2-1 ec2-user]#
```

2.2. View the current user and group information and display the current working directory. Note that the user identity changed, but not the current working directory.

```
[root@ip-192-0-2-1 ec2-user]# id
uid=0(root) gid=0(root) groups=0(root)
 context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[root@ip-192-0-2-1 ec2-user]# pwd
/home/ec2-user
```

2.3. View the environment variables which specify the home directory and the locations searched for executable files. Look for references to the **ec2-user** and **root** accounts.

```
[root@ip-192-0-2-1 ec2-user]# echo $HOME
/root
[root@ip-192-0-2-1 ec2-user]# echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin
```

Important

If you already have some experience with Linux and the **su** command, you may have expected that using **su** without the **-** option to become **root** would cause you to keep the current **PATH** of **ec2-user** (as seen in the previous step). That did not happen! But as you'll see in the next step, this isn't the normal **PATH** for **root** either.

What happened? The difference is that you did not run **su** directly (because **root** doesn't have a valid password, and since you are not **root** you need to use a password to use **su**). Instead, you ran **su** as **root** using the **sudo** command so you did not need a **root** password.

It turns out that **sudo** initially overrides the **PATH** from the initial environment for security reasons. That variable can still be updated by the command that was run after that initial override, which is what **su -** will do in a moment.

2.4. Exit the shell to return to the **ec2-user** user.

```
[root@ip-192-0-2-1 ec2-user]# exit
exit
[ec2-user@ip-192-0-2-1 ~]$
```

3. Switch to **root** by using **sudo** to run **su -**, and compare the characteristics of the new shell environment with those of the preceding example.

3.1. Become the **root** user at the shell prompt. Be sure all the login scripts are also executed.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su -
Last login: Fri Jul 24 17:16:01 EDT 2020 on pts/0
[root@ip-192-0-2-1 ~]#
```

Note the differences: the **Last login** message and the difference in the shell prompt.

3.2. View the current user and group information and display the current working directory.

```
[root@ip-192-0-2-1 ~]# id
uid=0(root) gid=0(root) groups=0(root)
 context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[root@ip-192-0-2-1 ~]# pwd
/root
```

3.3. View the environment variables which specify the home directory and the locations searched for executable files. Look for references to the **ec2-user** and **root** accounts.

```
[root@ip-192-0-2-1 ~]# echo $HOME
/root
[root@ip-192-0-2-1 ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

Important

In this example, after **sudo** reset the **PATH** environment variable from the settings in the **ec2-user** shell environment, the **su -** command ran the shell login scripts for **root** and set **PATH** again, to yet another value. The **su** command without the **-** option (in the previous example in this exercise) did not do that.

3.4. Exit the **root** shell to return to the **ec2-user** shell.

```
[root@ip-192-0-2-1 ~]# exit
logout
[ec2-user@ip-192-0-2-1 ~]$
```

4. Next, run several commands as **ec2-user** which require **root** access. Then, rather than using **sudo su -** to run them as **root** from an interactive **root** shell, use **sudo** to run those commands as **root** from the **ec2-user** shell as one-off commands.

4.1. View the last 5 lines of the **/var/log/messages**. Your output may vary from this example.

```
[ec2-user@ip-192-0-2-1 ~]$ tail -n 5 /var/log/messages
tail: cannot open '/var/log/messages' for reading: Permission denied
[ec2-user@ip-192-0-2-1 ~]$ sudo tail -n 5 /var/log/messages
Jul 24 17:10:10 ip-192-0-2-1 systemd: Started Network Manager Script Dispatcher
 Service.
```

```
Jul 24 17:10:10 ip-192-0-2-1 nm-dispatcher: req:1 'dhcp4-change' [eth0]: new
 request (3 scripts)
Jul 24 17:10:10 ip-192-0-2-1 nm-dispatcher: req:1 'dhcp4-change' [eth0]: start
 running ordered scripts...
Jul 24 17:16:01 ip-192-0-2-1 su: (to root) ec2-user on pts/0
Jul 24 17:18:12 ip-192-0-2-1 su: (to root) ec2-user on pts/0
[ec2-user@ip-192-0-2-1 ~]$
```

4.2. Make a backup of a configuration file in the **/etc** directory.

```
[ec2-user@ip-192-0-2-1 ~]$ cp /etc/issue /etc/issueOLD
cp: cannot create regular file '/etc/issueOLD': Permission denied
[ec2-user@ip-192-0-2-1 ~]$ sudo cp /etc/issue /etc/issueOLD
[ec2-user@ip-192-0-2-1 ~]$
```

4.3. Remove the **/etc/issueOLD** file that was just created.

```
[ec2-user@ip-192-0-2-1 ~]$ rm /etc/issueOLD
rm: remove write-protected regular empty file '/etc/issueOLD'? y
rm: cannot remove '/etc/issueOLD': Permission denied
[ec2-user@ip-192-0-2-1 ~]$ sudo rm /etc/issueOLD
[ec2-user@ip-192-0-2-1 ~]$
```

5. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Managing Local User Accounts

Objectives

After completing this section, you should be able to create, modify, and delete local user accounts.

Managing Local Users

A number of command-line tools can be used to manage local user accounts.

Creating Users from the Command Line

- The **useradd *username*** command creates a new user named **username**. It sets up the user's home directory and account information, and creates a private group for the user named **username**. At this point the account does not have a valid password set, and the user cannot log in until a password is set.
- The **useradd --help** command displays the basic options that can be used to override the defaults. In most cases, the same options can be used with the **usermod** command to modify an existing user.
- Some defaults, such as the range of valid UID numbers and default password aging rules, are read from the **/etc/login.defs** file. Values in this file are only used when creating new users. A change to this file does not affect existing users.

Modifying Existing Users from the Command Line

- The **usermod --help** command displays the basic options that can be used to modify an account. Some common options include:

usermod options:	Usage
-c, --comment COMMENT	Add the user's real name to the comment field.
-g, --gid GROUP	Specify the primary group for the user account.
-G, --groups GROUPS	Specify a comma-separated list of supplementary groups for the user account.
-a, --append	Used with the **-G** option to add the supplementary groups to the user's current set of group memberships instead of replacing the set of supplementary groups with a new set.
-d, --home HOME_DIR	Specify a particular home directory for the user account.
-m, --move-home	Move the user's home directory to a new location. Must be used with the **-d** option.
-s, --shell SHELL	Specify a particular login shell for the user account.
-L, --lock	Lock the user account.

usermod options:	Usage
-U, --unlock	Unlock the user account.

Deleting Users from the Command Line

- The **userdel *username*** command removes the details of **username** from **/etc/passwd**, but leaves the user's home directory intact.
- The **userdel -r *username*** command removes the details of **username** from **/etc/passwd** and also deletes the user's home directory.

Warning

When a user is removed with **userdel** without the **-r** option specified, the system will have files that are owned by an unassigned UID. This can also happen when a file, having a deleted user as its owner, exists outside that user's home directory. This situation can lead to information leakage and other security issues.

In Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux 8, the **useradd** command assigns new users the first free UID greater than or equal to 1000, unless you explicitly specify one using the **-u** option.

This is how information leakage can occur. If the first free UID had been previously assigned to a user account which has since been removed from the system, the old user's UID will get reassigned to the new user, giving the new user ownership of the old user's remaining files.

The following scenario demonstrates this situation.

```
[root@host ~]# useradd user01
[root@host ~]# ls -l /home
drwx------. 3 user01  user01    74 Feb  4 15:22 user01
[root@host ~]# userdel user01
[root@host ~]# ls -l /home
drwx------. 3    1000    1000   74 Feb  4 15:22 user01
[root@host ~]# useradd user02
[root@host ~]# ls -l /home
drwx------. 3 user02     user02       74 Feb  4 15:23 user02
drwx------. 3 user02     user02       74 Feb  4 15:22 user01
```

Notice that **user02** now owns all files that **user01** previously owned.

Depending on the situation, one solution to this problem is to remove all unowned files from the system when the user that created them is deleted. Another solution is to manually assign the unowned files to a different user. The **root** user can use the **find / -nouser -o -nogroup** command to find all unowned files and directories.

Setting Passwords from the Command Line

- The **passwd *username*** command sets the initial password or changes the existing password of **username**.

- The **root** user can set a password to any value. A message is displayed if the password does not meet the minimum recommended criteria, but is followed by a prompt to retype the new password and all tokens are updated successfully.

```
[root@host ~]# passwd user01
Changing password for user user01.
New password: redhat
BAD PASSWORD: The password fails the dictionary check - it is based on a
 dictionary word
Retype new password: redhat
passwd: all authentication tokens updated successfully.
[root@host ~]#
```

- A regular user must choose a password at least eight characters long and is also not based on a dictionary word, the username, or the previous password.

UID Ranges

Specific UID numbers and ranges of numbers are used for specific purposes by Red Hat Enterprise Linux.

- *UID 0* is always assigned to the superuser account, **root**.
- *UID 1-200* is a range of "system users" assigned statically to system processes by Red Hat.
- *UID 201-999* is a range of "system users" used by system processes that do not own files on the file system. They are typically assigned dynamically from the available pool when the software that needs them is installed. Programs run as these "unprivileged" system users in order to limit their access to only the resources they need to function.
- *UID 1000+* is the range available for assignment to regular users.

Note

Prior to RHEL 7, the convention was that UID 1-499 was used for system users and UID 500+ for regular users. Default ranges used by **useradd** and **groupadd** can be changed in the **/etc/login.defs** file.

References

useradd(8), **usermod**(8), **userdel**(8) man pages

Guided Exercise

Creating Users Using Command-line Tools

In this exercise, you will create a number of users on your Linux system, setting and recording an initial password for each user.

Outcomes

- Be able to configure a Linux system with additional user accounts.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

Steps

1. Become the **root** user at the shell prompt.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su -
[root@ip-192-0-2-1 ~]#
```

2. Add the user **juliet**.

```
[root@ip-192-0-2-1 ~]# useradd juliet
```

3. Confirm that **juliet** has been added by examining the **/etc/passwd** file.

```
[root@ip-192-0-2-1 ~]# tail -n 2 /etc/passwd
ec2-user:x:1000:1000:Cloud User:/home/ec2-user:/bin/bash
juliet:x:1001:1001::/home/juliet:/bin/bash
```

4. Use the **passwd** command to initialize **juliet**'s password. For this and any other passwords you set in this section, the values are not important but should be "secure". Be careful, the **root** user can set arbitrarily weak passwords!

   The example below assumes you typed something (the same thing!) for the `New password` and `Retype new password` prompts.

Warning

It is assumed that the AMI used for your Amazon EC2 instance was pre-configured for security reasons to prohibit remote logins over **ssh** which have been authenticated by password, and that key-based authentication is required. Please read the **Warning** box at the end of this exercise for further discussion.

```
[root@ip-192-0-2-1 ~]# passwd juliet
Changing password for user juliet.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[root@ip-192-0-2-1 ~]#
```

5. Continue adding the remaining users in the steps below and set initial passwords. These users will be used in the next exercise.

   5.1. **romeo**

```
[root@ip-192-0-2-1 ~]# useradd romeo
[root@ip-192-0-2-1 ~]# passwd romeo
Changing password for user romeo.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[root@ip-192-0-2-1 ~]#
```

   5.2. **hamlet**

```
[root@ip-192-0-2-1 ~]# useradd hamlet
[root@ip-192-0-2-1 ~]# passwd hamlet
```

   5.3. **sunni**

```
[root@ip-192-0-2-1 ~]# useradd sunni
[root@ip-192-0-2-1 ~]# passwd sunni
```

   5.4. **huong**

```
[root@ip-192-0-2-1 ~]# useradd huong
[root@ip-192-0-2-1 ~]# passwd huong
```

   5.5. **jerlene**

```
[root@ip-192-0-2-1 ~]# useradd jerlene
[root@ip-192-0-2-1 ~]# passwd jerlene
```

6. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Warning

In this exercise, you set passwords for various accounts on a cloud instance that is accessible from the public Internet. This is not a recommended practice if users can connect to the instance using **ssh** and authenticate using their password.

Attackers have been known to scan public cloud providers for instances providing remote **ssh** access and configured with weak passwords. The recommended AMI used to develop this course was configured to prohibit password-based **ssh** authentication, and to provide only **ssh** as a public-facing service, so setting those passwords for this exercise should not matter in practice. However, this is not necessarily true of a default Red Hat Enterprise Linux installation.

Additional keys can be installed by users to allow key-based authentication, but the procedure is beyond the scope of this course.

Managing Local Group Accounts

Objectives

After completing this section, students should be able to create, modify, and delete local group accounts.

Managing Local Groups

A group must exist before a user can be added to that group. Several command-line tools are used to manage local group accounts.

Creating Groups from the Command Line

- The **groupadd** command creates groups. Without options the **groupadd** command uses the next available GID from the range specified in the **/etc/login.defs** file while creating the groups.
- The **-g** option specifies a particular GID for the group to use.

```
[user01@host ~]$ sudo groupadd -g 10000 group01
[user01@host ~]$ tail /etc/group
...output omitted...
group01:x:10000:
```

Note

Given the automatic creation of user private groups (GID 1000+), it is generally recommended to set aside a range of GIDs to be used for supplementary groups. A higher range will avoid a collision with a system group (GID 0-999).

- The **-r** option creates a system group using a GID from the range of valid system GIDs listed in the **/etc/login.defs** file. The **SYS_GID_MIN** and **SYS_GID_MAX** configuration items in **/etc/login.defs** define the range of system GIDs.

```
[user01@host ~]$ sudo groupadd -r group02
[user01@host ~]$ tail /etc/group
...output omitted...
group01:x:10000:
group02:x:988:
```

Modifying Existing Groups from the Command Line

- The **groupmod** command changes the properties of an existing group. The **-n** option specifies a new name for the group.

```
[user01@host ~]$ sudo groupmod -n group0022 group02
[user01@host ~]$ tail /etc/group
...output omitted...
group0022:x:988:
```

Notice that the group name is updated to **group0022** from **group02**.

- The **-g** option specifies a new GID.

```
[user01@host ~]$ sudo groupmod -g 20000 group0022
[user01@host ~]$ tail /etc/group
...output omitted...
group0022:x:20000:
```

Notice that the GID is updated to **20000** from **988**.

Deleting Groups from the Command Line

- The **groupdel** command removes groups.

```
[user01@host ~]$ sudo groupdel group0022
```

Note

You cannot remove a group if it is the primary group of any existing user. As with **userdel**, check all file systems to ensure that no files remain on the system that are owned by the group.

Changing Group Membership from the Command Line

- The membership of a group is controlled with user management. Use the **usermod -g** command to change a user's primary group.

```
[user01@host ~]$ id user02
uid=1006(user02) gid=1008(user02) groups=1008(user02)
[user01@host ~]$ sudo usermod -g group01 user02
[user01@host ~]$ id user02
uid=1006(user02) gid=10000(group01) groups=10000(group01)
```

- Use the **usermod -aG** command to add a user to a supplementary group.

```
[user01@host ~]$ id user03
uid=1007(user03) gid=1009(user03) groups=1009(user03)
[user01@host ~]$ sudo usermod -aG group01 user03
[user01@host ~]$ id user03
uid=1007(user03) gid=1009(user03) groups=1009(user03),10000(group01)
```

Important

The use of the **-a** option makes **usermod** function in *append* mode. Without **-a**, the user will be removed from any of their current supplementary groups that are not included in the **-G** option's list.

References

group(5), **groupadd**(8),**groupdel**(8), and **usermod**(8) man pages

Guided Exercise

Managing Groups Using Command-line Tools

In this exercise, you will add users to newly created supplementary groups.

Outcomes

- The **shakespeare** group consists of users **juliet**, **romeo**, and **hamlet**.
- The **artists** group consists of users **sunni**, **huong**, and **jerlene**.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

Steps

1. Become the **root** user at the shell prompt.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su -
```

2. Create a supplementary group called **shakespeare** with a group ID of **30000**.

```
[root@ip-192-0-2-1 ~]# groupadd -g 30000 shakespeare
```

3. Create a supplementary group called **artists**.

```
[root@ip-192-0-2-1 ~]# groupadd artists
```

4. Confirm that **shakespeare** and **artists** have been added by examining the **/etc/group** file.

```
[root@ip-192-0-2-1 ~]# tail -n 5 /etc/group
sunni:x:1004:
huong:x:1005:
jerlene:x:1006:
shakespeare:x:30000:
artists:x:30001:
```

5. Add the **juliet** user to the **shakespeare** group as a supplementary group.

```
[root@ip-192-0-2-1 ~]# usermod -G shakespeare juliet
```

6. Confirm that **juliet** has been added using the **id** command.

```
[root@ip-192-0-2-1 ~]# id juliet
uid=1001(juliet) gid=1001(juliet) groups=1001(juliet),30000(shakespeare)
```

7. Continue adding the remaining users to groups as follows:

7.1. Add **romeo** and **hamlet** to the **shakespeare** group.

```
[root@ip-192-0-2-1 ~]# usermod -G shakespeare romeo
[root@ip-192-0-2-1 ~]# usermod -G shakespeare hamlet
```

7.2. Add **sunni**, **huong**, and **jerlene** to the **artists** group.

```
[root@ip-192-0-2-1 ~]# usermod -G artists sunni
[root@ip-192-0-2-1 ~]# usermod -G artists huong
[root@ip-192-0-2-1 ~]# usermod -G artists jerlene
```

7.3. Verify the supplemental group memberships by examining the **/etc/group** file.

```
[root@ip-192-0-2-1 ~]# tail -n 5 /etc/group
sunni:x:1004:
huong:x:1005:
jerlene:x:1006:
shakespeare:x:30000:juliet,romeo,hamlet
artists:x:30001:sunni,huong,jerlene
```

8. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Lab

Managing Local Linux Users and Groups

In this lab, you will create three new users and a group, and make those users members of that group as a supplementary group.

Outcomes

- Be able to create three new user acccounts for Dinesh Jonasen (**`djonasen`**), Sumiko Alves (**`salves`**), and Denver Tyson (**`dtyson`**).
- Be able to create a new group called **`consultants`**, including the three new user accounts for Dinesh Jonasen, Sumiko Alves, and Denver Tyson as members.

Before You Begin

Start your Amazon EC2 instance and use **`ssh`** to log in as the user **`ec2-user`**. It is assumed that **`ec2-user`** can use **`sudo`** to run commands as **`root`**.

Steps

1. Create a new group named **`consultants`** with a GID of 40000.
2. Create three new users: **`djonasen`**, **`salves`**, and **`dtyson`**, set a password for each, and add each to the supplementary group **`consultants`**. Each user's primary group should be a user private group with the same name as the user.

Warning

It's assumed that you set a "secure" password for each of these users, as discussed in the **Warning** in a previous exercise in this chapter.

3. Check each user to confirm the account exists and has the correct group memberships.
4. This concludes this lab. Log out and stop your Amazon EC2 instance.

Solution

Managing Local Linux Users and Groups

In this lab, you will create three new users and a group, and make those users members of that group as a supplementary group.

Outcomes

- Be able to create three new user acccounts for Dinesh Jonasen (**djonasen**), Sumiko Alves (**salves**), and Denver Tyson (**dtyson**).
- Be able to create a new group called **consultants**, including the three new user accounts for Dinesh Jonasen, Sumiko Alves, and Denver Tyson as members.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

Steps

1. Create a new group named **consultants** with a GID of 40000.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo groupadd -g 40000 consultants
[ec2-user@ip-192-0-2-1 ~]$ tail -n 5 /etc/group
huong:x:1005:
jerlene:x:1006:
shakespeare:x:30000:juliet,romeo,hamlet
artists:x:30001:sunni,huong,jerlene
consultants:x:40000:
```

2. Create three new users: **djonasen**, **salves**, and **dtyson**, set a password for each, and add each to the supplementary group **consultants**. Each user's primary group should be a user private group with the same name as the user.

Warning

It's assumed that you set a "secure" password for each of these users, as discussed in the **Warning** in a previous exercise in this chapter.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo useradd -G consultants djonasen
[ec2-user@ip-192-0-2-1 ~]$ sudo useradd -G consultants salves
[ec2-user@ip-192-0-2-1 ~]$ sudo useradd -G consultants dtyson
[ec2-user@ip-192-0-2-1 ~]$ tail -n 5 /etc/group
artists:x:30001:sunni,huong,jerlene
consultants:x:40000:djonasen,salves,dtyson
djonasen:x:1007:
salves:x:1008:
dtyson:x:1009:
```

```
[ec2-user@ip-192-0-2-1 ~]$ sudo passwd djonasen
Changing password for user djonasen.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[ec2-user@ip-192-0-2-1 ~]$ sudo passwd salves
Changing password for user salves.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[ec2-user@ip-192-0-2-1 ~]$ sudo passwd dtyson
Changing password for user dtyson.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

3. Check each user to confirm the account exists and has the correct group memberships.

   The exact UID numbers may differ for your solution, but the names of the users and groups should be correct, and **`consultants`** should have GID 40000.

```
[ec2-user@ip-192-0-2-1 ~]$ id djonasen
uid=1007(djonasen) gid=1007(djonasen) groups=1007(djonasen),40000(consultants)
[ec2-user@ip-192-0-2-1 ~]$ id salves
uid=1008(salves) gid=1008(salves) groups=1008(salves),40000(consultants)
[ec2-user@ip-192-0-2-1 ~]$ id dtyson
uid=1009(dtyson) gid=1009(dtyson) groups=1009(dtyson),40000(consultants)
```

4. This concludes this lab. Log out and stop your Amazon EC2 instance.

Chapter 6

Controlling Access to Files with Linux File System Permissions

Goal

To set Linux file system permissions on files and interpret the security effects of different permission settings.

Objectives

- Explain how the Linux file permissions model works.
- Change the permissions and ownership of files using command-line tools.
- Configure a directory in which newly created files are automatically writable by members of the group which owns the directory, using special permissions and default umask settings.

Sections

- Linux File System Permissions (and Quiz)
- Managing File System Permissions from the Command Line (and Guided Exercise)
- Managing Default Permissions and File Access (and Guided Exercise)

Lab

Controlling Access to Files with Linux File System Permissions

Linux File System Permissions

Objectives

After completing this section, you should be able to list file-system permissions on files and directories, and interpret the effect of those permissions on access by users and groups.

Linux File-system Permissions

File permissions control access to files. Linux file permissions are simple but flexible, easy to understand and apply, yet still able to handle most normal permission cases easily.

Files have three user categories to which permissions apply. The file is owned by a user, normally the one who created the file. The file is also owned by a single group, usually the primary group of the user who created the file, but this can be changed. Different permissions can be set for the owning user, the owning group, and for all other users on the system that are not the user or a member of the owning group.

The most specific permissions take precedence. *User* permissions override *group* permissions, which override *other* permissions.

In *Figure 6.1*, **`joshua`** is a member of the groups **`joshua`** and **`web`**, and **`allison`** is a member of **`allison`**, **`wheel`**, and **`web`**. When **`joshua`** and **`allison`** need to collaborate, the files should be associated with the group **`web`** and group permissions should allow the desired access.

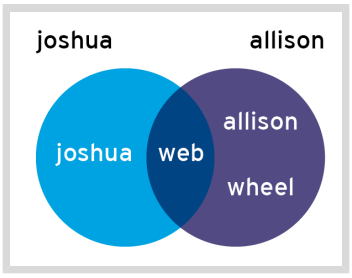


Figure 6.1: Example group membership to facilitate collaboration

Three permission categories apply: read, write, and execute. The following table explains how these permissions affect access to files and directories.

Effects of Permissions on Files and Directories

Permission	Effect on files	Effect on directories
`r` (read)	File contents can be read.	Contents of the directory (the file names) can be listed.
`w` (write)	File contents can be changed.	Any file in the directory can be created or deleted.

Permission	Effect on files	Effect on directories
x (execute)	Files can be executed as commands.	The directory can become the current working directory. (You can **cd** into it, but also require read permission to list files found there.)

Users normally have both read and execute permissions on read-only directories so that they can list the directory and have full read-only access to its contents. If a user only has read access on a directory, the names of the files in it can be listed, but no other information, including permissions or time stamps, are available, nor can they be accessed. If a user only has execute access on a directory, they cannot list file names in the directory. If they know the name of a file that they have permission to read, they can access the contents of that file from outside the directory by explicitly specifying the relative file name.

A file may be removed by anyone who has ownership of, or write permission to, the directory in which the file resides, regardless of the ownership or permissions on the file itself. This can be overridden with a special permission, the *sticky bit*, discussed later in this chapter.

Note

Linux file permissions work differently than the permissions system used by the NTFS file system for Microsoft Windows.

On Linux, permissions apply only to the file or directory on which they are set. That is, permissions on a directory are not inherited automatically by the subdirectories and files within it. However, permissions on a directory can block access to the contents of the directory depending on how restrictive they are.

The **read** permission on a directory in Linux is roughly equivalent to **List folder contents** in Windows.

The **write** permission on a directory in Linux is equivalent to **Modify** in Windows; it implies the ability to delete files and subdirectories. In Linux, if **write** and the **sticky bit** are both set on a directory, then only the file or subdirectory owner may delete it, which is similar to the Windows **Write** permission behavior.

The Linux root user has the equivalent of the Windows **Full Control** permission on all files. However, root may have access restricted by the system's SELinux policy using process and file security contexts. SELinux will be discussed in a later course.

Viewing File and Directory Permissions and Ownership

The **-l** option of the **ls** command shows detailed information about permissions and ownership:

```
[user@host~]$ ls -l test
-rw-rw-r--. 1 student student 0 Feb  8 17:36 test
```

Use the **-d** option to show detailed information about a directory itself, and not its contents.

```
[user@host ~]$ ls -ld /home
drwxr-xr-x. 5 root root 4096 Jan 31 22:00 /home
```

The first character of the long listing is the file type, interpreted like this:

- **-** is a regular file.
- **d** is a directory.
- **l** is a soft link.
- Other characters represent hardware devices (**b** and **c**) or other special-purpose files (**p** and **s**).

The next nine characters are the file permissions. These are in three sets of three characters: permissions that apply to the user that owns the file, the group that owns the file, and all other users. If the set shows **rwx**, that category has all three permissions, read, write, and execute. If a letter has been replaced by **-**, then that category does not have that permission.

After the link count, the first name specifies the user that owns the file, and the second name the group that owns the file.

So in the example above, the permissions for user **student** are specified by the first set of three characters. User **student** has read and write on **test**, but not execute.

Group **student** is specified by the second set of three characters: it also has read and write on **test**, but not execute.

Any other user's permissions are specified by the third set of three characters: they only have read permission on **test**.

The most specific set of permissions apply. So if user **student** has different permissions than group **student**, and user **student** is also a member of that group, then the user permissions will be the ones that apply.

Examples of Permission Effects

The following examples will help illustrate how file permissions interact. For these examples, we have four users with the following group memberships:

User	Group Memberships
operator1	**operator1**, **consultant1**
database1	**database1**, **consultant1**
database2	**database2**, **operator2**
contractor1	**contractor1**, **operator2**

Those users will be working with files in the **dir** directory. This is a long listing of the files in that directory:

```
[database1@host dir]$ ls -la
total 24
drwxrwxr-x.  2 database1 consultant1   4096 Apr  4 10:23 .
drwxr-xr-x. 10 root      root          4096 Apr  1 17:34 ..
-rw-rw-r--.  1 operator1 operator1     1024 Apr  4 11:02 lfile1
-rw-r--rw-.  1 operator1 consultant1   3144 Apr  4 11:02 lfile2
-rw-rw-r--.  1 database1 consultant1  10234 Apr  4 10:14 rfile1
-rw-r-----.  1 database1 consultant1   2048 Apr  4 10:18 rfile2
```

The **-a** option shows the permissions of hidden files, including the special files used to represent the directory and its parent. In this example, **.** reflects the permissions of **dir** itself, and **..** the permissions of its parent directory.

What are the permissions of **rfile1**? The user that owns the file (**database1**) has read and write but not execute. The group that owns the file (**consultant1**) has read and write but not execute. All other users have read but not write or execute.

The following table explores some of the effects of this set of permissions for these users:

Effect	Why is this true?
The user **operator1** can change the contents of **rfile1**.	User **operator1** is a member of the **consultant1** group, and that group has both read and write permissions on **rfile1**.
The user **database1** can view and modify the contents of **rfile2**.	User **database1** owns the file and has both read and write access to **rfile2**.
The user **operator1** can view but not modify the contents of **rfile2** (without deleting it and recreating it).	User **operator1** is a member of the **consultant1** group, and that group only has read access to **rfile2**.
The users **database2** and **contractor1** do not have any access to the contents of **rfile2**.	**other** permissions apply to users **database2** and **contractor1**, and those permissions do not include read or write permission.
operator1 is the only user who can change the contents of **lfile1** (without deleting it and recreating it).	User and group **operator1** have write permission on the file, other users do not. But the only member of group **operator1** is user **operator1**.
The user **database2** can change the contents of **lfile2**.	User **database2** is not the user that owns the file and is not in group **consultant1**, so **other** permissions apply. Those grant write permission.
The user **database1** can view the contents of **lfile2**, but cannot modify the contents of **lfile2** (without deleting it and recreating it).	User **database1** is a member of the group **consultant1**, and that group only has read permissions on **lfile2**. Even though **other** has write permission, the group permissions take precedence.
The user **database1** can delete **lfile1** and **lfile2**.	User **database1** has write permissions on the directory containing both files (shown by **.**), and therefore can delete any file in that directory. This is true even if **database1** does not have write permission on the file itself.

References

ls(1) man page

info coreutils (*GNU Coreutils*)

- Section 13: Changing file attributes

Quiz

Interpreting File and Directory Permissions

Review the following information and use it to answer the quiz questions.

The system has four users assigned to the following groups:

- User **`consultant1`** is in groups **`consultant1`** and **`database1`**
- User **`operator1`** is in groups **`operator1`** and **`database1`**
- User **`contractor1`** is in groups **`contractor1`** and **`contractor3`**
- User **`operator2`** is in groups **`operator2`** and **`contractor3`**

The current directory (`.`) contains four files with the following permissions information:

```
drwxrwxr-x.   operator1     database1     .
-rw-rw-r--.   consultant1   consultant1   lfile1
-rw-r--rw-.   consultant1   database1     lfile2
-rw-rw-r--.   operator1     database1     rfile1
-rw-r-----.   operator1     database1     rfile2
```

1. Which regular file is owned by `operator1` and readable by all users?

a. **`lfile1`**
b. **`lfile2`**
c. **`rfile1`**
d. **`rfile2`**

2. Which file can be modified by the `contractor1` user?

a. **`lfile1`**
b. **`lfile2`**
c. **`rfile1`**
d. **`rfile2`**

3. Which file cannot be read by the `operator2` user?

a. **`lfile1`**
b. **`lfile2`**
c. **`rfile1`**
d. **`rfile2`**

4. Which file has a group ownership of `consultant1`?

a. **`lfile1`**
b. **`lfile2`**
c. **`rfile1`**
d. **`rfile2`**

5. Which files can be deleted by the `operator1` user?

a. **`rfile1`**
b. **`rfile2`**
c. All of the above.
d. None of the above.

6. Which files can be deleted by the `operator2` user?

a. **`lfile1`**
b. **`lfile2`**
c. All of the above.
d. None of the above.

Solution

Interpreting File and Directory Permissions

Review the following information and use it to answer the quiz questions.

The system has four users assigned to the following groups:

- User **`consultant1`** is in groups **`consultant1`** and **`database1`**
- User **`operator1`** is in groups **`operator1`** and **`database1`**
- User **`contractor1`** is in groups **`contractor1`** and **`contractor3`**
- User **`operator2`** is in groups **`operator2`** and **`contractor3`**

The current directory (`.`) contains four files with the following permissions information:

```
drwxrwxr-x.   operator1     database1     .
-rw-rw-r--.   consultant1   consultant1   lfile1
-rw-r--rw-.   consultant1   database1     lfile2
-rw-rw-r--.   operator1     database1     rfile1
-rw-r-----.   operator1     database1     rfile2
```

1. Which regular file is owned by `operator1` and readable by all users?

a. `lfile1`
b. `lfile2`
c. **`rfile1`**
d. `rfile2`

2. Which file can be modified by the `contractor1` user?

a. `lfile1`
b. **`lfile2`**
c. `rfile1`
d. `rfile2`

3. Which file cannot be read by the `operator2` user?

a. `lfile1`
b. `lfile2`
c. `rfile1`
d. **`rfile2`**

4. Which file has a group ownership of `consultant1`?

a. **`lfile1`**
b. `lfile2`
c. `rfile1`
d. `rfile2`

5. Which files can be deleted by the `operator1` user?

a. `rfile1`
b. `rfile2`
c. All of the above.
d. None of the above.

6. Which files can be deleted by the `operator2` user?

a. `lfile1`
b. `lfile2`
c. All of the above.
d. None of the above.

Managing File System Permissions from the Command Line

Objectives

After completing this section, you should be able to change the permissions and ownership of files using command-line tools.

Changing File and Directory Permissions

The command used to change permissions from the command line is **chmod**, which means "change mode" (permissions are also called the *mode* of a file). The **chmod** command takes a permission instruction followed by a list of files or directories to change. The permission instruction can be issued either symbolically (the symbolic method) or numerically (the numeric method).

Changing Permissions with the Symbolic Method

```
chmod WhoWhatWhich file|directory
```

- *Who* is u, g, o, a *(for user, group, other, all)*
- *What* is +, -, = *(for add, remove, set exactly)*
- *Which* is r, w, x *(for read, write, execute)*

The *symbolic* method of changing file permissions uses letters to represent the different groups of permissions: **u** for user, **g** for group, **o** for other, and **a** for all.

With the symbolic method, it is not necessary to set a complete new group of permissions. Instead, you can change one or more of the existing permissions. Use **+** or **-** to add or remove permissions, respectively, or use **=** to replace the entire set for a group of permissions.

The permissions themselves are represented by a single letter: **r** for read, **w** for write, and **x** for execute. When using **chmod** to change permissions with the symbolic method, using a capital **X** as the permission flag will add execute permission only if the file is a directory or already has execute set for user, group, or other.

Note

The **chmod** command supports the **-R** option to recursively set permissions on the files in an entire directory tree. When using the **-R** option, it can be useful to set permissions symbolically using the **X** option. This allows the execute (search) permission to be set on directories so that their contents can be accessed, without changing permissions on most files. Be cautious with the **X** option, however, because if a file has any execute permission set, **X** will set the specified execute permission on that file as well. For example, the following command recursively sets read and write access on **demodir** and all its children for their group owner, but only applies group execute permissions to directories and files that already have execute set for user, group, or other.

```
[root@host opt]# chmod -R g+rwX demodir
```

Examples

- Remove read and write permission for group and other on **file1**:

```
[user@host ~]$ chmod go-rw file1
```

- Add execute permission for everyone on **file2**:

```
[user@host ~]$ chmod a+x file2
```

Changing Permissions with the Numeric Method

In the example below the # character represents a digit.

```
chmod ### file|directory
```

- Each digit represents permissions for an access level: user, group, other.
- The digit is calculated by adding together numbers for each permission you want to add, 4 for read, 2 for write, and 1 for execute.

Using the *numeric* method, permissions are represented by a 3-digit (or 4-digit, when setting advanced permissions) *octal* number. A single octal digit can represent any single value from 0-7.

In the 3-digit octal (numeric) representation of permissions, each digit stands for one access level, from left to right: user, group, and other. To determine each digit:

1. Start with 0.
2. If the read permission should be present for this access level, add 4.
3. If the write permission should be present, add 2.
4. If the execute permission should be present, add 1.

Examine the permissions **-rwxr-x---**. For the user, **rwx** is calculated as 4+2+1=7. For the group, **r-x** is calculated as 4+0+1=5, and for other users, **---** is represented with 0. Putting these three together, the numeric representation of those permissions is 750.

This calculation can also be performed in the opposite direction. Look at the permissions 640. For the user permissions, 6 represents read (4) and write (2), which displays as **rw-**. For the group

part, 4 only includes read (4) and displays as **r--**. The 0 for other provides no permissions (**---**) and the final set of symbolic permissions for this file is **-rw-r-----**.

Experienced administrators often use numeric permissions because they are shorter to type and pronounce, while still giving full control over all permissions.

Examples

- Set read and write permissions for user, read permission for group and other, on **samplefile**:

```
[user@host ~]$ chmod 644 samplefile
```

- Set read, write, and execute permissions for user, read and execute permissions for group, and no permission for other on **sampledir**:

```
[user@host ~]$ chmod 750 sampledir
```

Changing File and Directory User or Group Ownership

A newly created file is owned by the user who creates that file. By default, new files have a group ownership that is the primary group of the user creating the file. In Red Hat Enterprise Linux, a user's primary group is usually a private group with only that user as a member. To grant access to a file based on group membership, the group that owns the file may need to be changed.

Only **root** can change the user that owns a file. Group ownership, however, can be set by **root** or by the file's owner. **root** can grant file ownership to any group, but regular users can make a group the owner of a file only if they are a member of that group.

File ownership can be changed with the **chown** (change owner) command. For example, to grant ownership of the **test_file** file to the **student** user, use the following command:

```
[root@host ~]# chown student test_file
```

chown can be used with the **-R** option to recursively change the ownership of an entire directory tree. The following command grants ownership of **test_dir** and all files and subdirectories within it to **student**:

```
[root@host ~]# chown -R student test_dir
```

The **chown** command can also be used to change group ownership of a file by preceding the group name with a colon (:). For example, the following command changes the group **test_dir** to **admins**:

```
[root@host ~]# chown :admins test_dir
```

The **chown** command can also be used to change both owner and group at the same time by using the *owner*:*group* syntax. For example, to change the ownership of **test_dir** to **visitor** and the group to **guests**, use the following command:

```
[root@host ~]# chown visitor:guests test_dir
```

Instead of using **chown**, some users change the group ownership by using the **chgrp** command. This command works just like **chown**, except that it is only used to change group ownership and the colon (:) before the group name is not required.

Important

You may encounter examples of **chown** commands using an alternative syntax that separates owner and group with a period instead of a colon:

```
[root@host ~]# chown owner.group filename
```

You should not use this syntax. Always use a colon.

A period is a valid character in a user name, but a colon is not. If the user **enoch.root**, the user **enoch**, and the group **root** exist on the system, the result of **chown enoch.root filename** will be to have **filename** owned by the user **enoch.root**. You may have been trying to set the file ownership to the user **enoch** and group **root**. This can be confusing.

If you always use the **chown** colon syntax when setting the user and group at the same time, the results are always easy to predict.

References

ls(1), **chmod**(1), **chown**(1), and **chgrp**(1) man pages

Guided Exercise

Managing File Security from the Command Line

In this exercise, you will create a collaborative directory for pre-existing users.

Outcomes

- Create a directory with permissions that make it accessible by all members of the **ateam** group.
- Create a file owned by user **andy** that can be modified by **alice**.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

Steps

1. Become the **root** user at the shell prompt.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su -
[root@ip-192-0-2-1 ~]#
```

2. Create a group, **ateam**. Create two new users, **andy** and **alice**, who are members of that group.

```
[root@ip-192-0-2-1 ~]# groupadd ateam
[root@ip-192-0-2-1 ~]# useradd -G ateam andy
[root@ip-192-0-2-1 ~]# useradd -G ateam alice
[root@ip-192-0-2-1 ~]# id andy; id alice
uid=1010(andy) gid=1010(andy) groups=1010(andy),40001(ateam)
uid=1011(alice) gid=1011(alice) groups=1011(alice),40001(ateam)
```

3. Create a directory in **/home** called **ateam-text**.

```
[root@ip-192-0-2-1 ~]# mkdir /home/ateam-text
```

4. Change the group ownership of the **ateam-text** directory to **ateam**.

```
[root@ip-192-0-2-1 ~]# chown :ateam /home/ateam-text
```

5. Ensure the permissions of **ateam-text** allows group members to create and delete files.

```
[root@ip-192-0-2-1 ~]# chmod g+w /home/ateam-text
```

6. Ensure the permissions of **ateam-text** forbids others from accessing its files.

```
[root@ip-192-0-2-1 ~]# chmod 770 /home/ateam-text
[root@ip-192-0-2-1 ~]$ ls -ld /home/ateam-text
drwxrwx---.  2 root ateam 6 Jul 24 12:50 /home/ateam-text
```

7. Exit the root shell and switch to the user **andy**.

```
[root@ip-192-0-2-1 ~]# exit
[ec2-user@ip-192-0-2-1 ~]$ sudo su - andy
[andy@ip-192-0-2-1 ~]$
```

8. Navigate to the **/home/ateam-text** folder (remember to open a terminal window first).

```
[andy@ip-192-0-2-1 ~]$ cd /home/ateam-text
```

9. Create an empty file called **andyfile3**.

```
[andy@ip-192-0-2-1 ateam-text]$ touch andyfile3
```

10. Record the default user and group ownership of the new file and its permissions.

```
[andy@ip-192-0-2-1 ateam-text]$ ls -l andyfile3
-rw-rw-r--.  1 andy andy 0 Jul 24 12:59 andyfile3
```

11. Change the group ownership of the new file to **ateam** and record the new ownership and permissions.

```
[andy@ip-192-0-2-1 ateam-text]$ chown :ateam andyfile3
[andy@ip-192-0-2-1 ateam-text]$ ls -l andyfile3
-rw-rw-r--.  1 andy ateam 0 Jul 24 12:59 andyfile3
```

12. Exit the shell and switch to the user **alice**.

```
[andy@ip-192-0-2-1 ateam-text]$ exit
[ec2-user@ip-192-0-2-1 ~]$ sudo su - alice
[alice@ip-192-0-2-1 ~]$
```

13. Navigate to the **/home/ateam-text** folder.

```
[alice@ip-192-0-2-1 ~]$ cd /home/ateam-text
```

14. Determine **alice**'s privileges to access and/or modify **andyfile3**.

```
[alice@ip-192-0-2-1 ateam-text]$ echo "text" >> andyfile3
[alice@ip-192-0-2-1 ateam-text]$ cat andyfile3
text
```

If you didn't set the permissions correctly, the above commands will instead result in something like the following:

```
[alice@ip-192-0-2-1 ateam-text]$ echo "text" >> andyfile3
-bash: /home/ateam-text/andyfile3: Permission denied
```

Important

In the preceding example, the command **echo "text" >> andyfile3** is using a technique called *shell I/O redirection* to append the line **text** to the end of the file **andyfile3**. The output of the **echo** command is appended to the end of the file that the **>>** points at. Be careful to use **>>** and not just one **>**, since the **>** operator will overwrite the entire file and replace its contents with only the line **text**.

If you are interested in more information on shell I/O redirection, an overview is available at http://wiki.bash-hackers.org/syntax/redirection.

Be careful if you choose to test this by having **alice** edit **andyfile3** with **vim**. If the permissions are wrong on the file, **vim** will warn you that you're editing a read-only file. But if the **/home/ateam-test** *directory* is writable by **alice**, then **vim** will still let you use **:wq!** to write the file, even if **alice** doen't have write permission on the file!

How is this possible? It turns out **vim** is "smart" enough to recognize that it can overwrite the file by deleting the file from the directory and creating a new copy. This is allowed because having write on a directory means you can delete any file in that directory, even if you can't write the file directly. The **!** on the **vim** command **:wq!** indicates that you want it to do everything it can to write the file.

Note that one side effect of this is that the file's owner will change to **alice**, and other permissions may change as well.

15. This completes this exercise. Log out and stop your Amazon EC2 instance.

Managing Default Permissions and File Access

Objectives

After completing this section, students should be able to:

- Control the default permissions of new files created by users.
- Explain the effect of special permissions.
- Use special permissions and default permissions to set the group owner of files created in a particular directory.

Special Permissions

Special permissions constitute a fourth permission type in addition to the basic user, group, and other types. As the name implies, these permissions provide additional access-related features over and above what the basic permission types allow. This section details the impact of special permissions, summarized in the table below.

Effects of Special Permissions on Files and Directories

Special permission	Effect on files	Effect on directories
u+s (suid)	File executes as the user that owns the file, not the user that ran the file.	No effect.
g+s (sgid)	File executes as the group that owns the file.	Files newly created in the directory have their group owner set to match the group owner of the directory.
o+t (sticky)	No effect.	Users with write access to the directory can only remove files that they own; they cannot remove or force saves to files owned by other users.

The *setuid* permission on an executable file means that commands run as the user owning the file, not as the user that ran the command. One example is the **passwd** command:

```
[user@host ~]$ ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root root 35504 Jul 16  2010 /usr/bin/passwd
```

In a long listing, you can identify the setuid permissions by a lowercase **s** where you would normally expect the **x** (owner execute permissions) to be. If the owner does not have execute permissions, this is replaced by an uppercase **S**.

The special permission *setgid* on a directory means that files created in the directory inherit their group ownership from the directory, rather than inheriting it from the creating user. This is commonly used on group collaborative directories to automatically change a file from the default

private group to the shared group, or if files in a directory should be always owned by a specific group. An example of this is the **`/run/log/journal`** directory:

```
[user@host ~]$ ls -ld /run/log/journal
drwxr-sr-x. 3 root systemd-journal 60 May 18 09:15 /run/log/journal
```

If setgid is set on an executable file, commands run as the group that owns that file, not as the user that ran the command, in a similar way to setuid works. One example is the **`locate`** command:

```
[user@host ~]$ ls -ld /usr/bin/locate
-rwx--s--x. 1 root slocate 47128 Aug 12 17:17 /usr/bin/locate
```

In a long listing, you can identify the setgid permissions by a lowercase **s** where you would normally expect the **x** (group execute permissions) to be. If the group does not have execute permissions, this is replaced by an uppercase **S**.

Lastly, the *sticky bit* for a directory sets a special restriction on deletion of files. Only the owner of the file (and **`root`**) can delete files within the directory. An example is **`/tmp`**:

```
[user@host ~]$ ls -ld /tmp
drwxrwxrwt. 39 root root 4096 Feb  8 20:52 /tmp
```

In a long listing, you can identify the sticky permissions by a lowercase **t** where you would normally expect the **x** (other execute permissions) to be. If other does not have execute permissions, this is replaced by an uppercase **T**.

Setting Special Permissions

- Symbolically: setuid = **`u+s`**; setgid = **`g+s`**; sticky = **`o+t`**
- Numerically (fourth preceding digit): setuid = 4; setgid = 2; sticky = 1

Examples

- Add the setgid bit on **`directory`**:

```
[user@host ~]# chmod g+s directory
```

- Set the setgid bit and add read/write/execute permissions for user and group, with no access for others, on **`directory`**:

```
[user@host ~]# chmod 2770 directory
```

Default File Permissions

When you create a new file or directory, it is assigned initial permissions. There are two things that affect these initial permissions. The first is whether you are creating a regular file or a directory. The second is the current *umask*.

If you create a new directory, the operating system starts by assigning it octal permissions 0777 (**`drwxrwxrwx`**). If you create a new regular file, the operating system assignes it octal permissions 0666 (**`-rw-rw-rw-`**). You always have to explicitly add execute permission to a regular file. This

makes it harder for an attacker to compromise a network service so that it creates a new file and immediately executes it as a program.

However, the shell session will also set a umask to further restrict the permissions that are initially set. This is an octal bitmask used to clear the permissions of new files and directories created by a process. If a bit is set in the umask, then the corresponding permission is cleared on new files. For example, the umask 0002 clears the write bit for other users. The leading zeros indicate the special, user, and group permissions are not cleared. A umask of 0077 clears all the group and other permissions of newly created files.

The **umask** command without arguments will display the current value of the shell's umask:

```
[user@host ~]$ umask
0002
```

Use the **umask** command with a single numeric argument to change the umask of the current shell. The numeric argument should be an octal value corresponding to the new umask value. You can omit any leading zeros in the umask.

The system's default umask values for Bash shell users are defined in the **/etc/profile** and **/etc/bashrc** files. Users can override the system defaults in the **.bash_profile** and **.bashrc** files in their home directories.

umask Example

The following example explains how the umask affects the permissions of files and directories. Look at the default umask permissions for both files and directories in the current shell. The owner and group both have read and write permission on files, and other is set to read. The owner and group both have read, write, and execute permissions on directories. The only permission for other is read.

```
[user@host ~]$ umask
0002
[user@host ~]$ touch default
[user@host ~]$ ls -l default.txt
-rw-rw-r--. 1 user user 0 May  9 01:54 default.txt
[user@host ~]$ mkdir default
[user@host ~]$ ls -ld default
drwxrwxr-x. 2 user user 0 May  9 01:54 default
```

By setting the umask value to 0, the file permissions for other change from read to read and write. The directory permissions for other changes from read and execute to read, write, and execute.

```
[user@host ~]$ umask 0
[user@host ~]$ touch zero.txt
[user@host ~]$ ls -l zero.txt
-rw-rw-rw-. 1 user user 0 May  9 01:54 zero.txt
[user@host ~]$ mkdir zero
[user@host ~]$ ls -ld zero
drwxrwxrwx. 2 user user 0 May  9 01:54 zero
```

To mask all file and directory permissions for other, set the umask value to 007.

```
[user@host ~]$ umask 007
[user@host ~]$ touch seven.txt
[user@host ~]$ ls -l seven.txt
-rw-rw----. 1 user user 0 May  9 01:55 seven.txt
[user@host ~]$ mkdir seven
[user@host ~]$ ls -ld seven
drwxrwx---. 2 user user 0 May  9 01:54 seven
```

A umask of 027 ensures that new files have read and write permissions for user and read permission for group. New directories have read and write access for group and no permissions for other.

```
[user@host ~]$ umask 027
[user@host ~]$ touch two-seven.txt
[user@host ~]$ ls -l two-seven.txt
-rw-r-----. 1 user user 0 May  9 01:55 two-seven.txt
[user@host ~]$ mkdir two-seven
[user@host ~]$ ls -ld two-seven
drwxr-x---. 2 user user 0 May  9 01:54 two-seven
```

The default umask for users is set by the shell startup scripts. By default, if your account's UID is 200 or more and your username and primary group name are the same, you will be assigned a umask of 002. Otherwise, your umask will be 022.

As **root**, you can change this by adding a shell startup script named **/etc/profile.d/local-umask.sh** that looks something like the output in this example:

```
[root@host ~]# cat /etc/profile.d/local-umask.sh
# Overrides default umask configuration
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
    umask 007
else
    umask 022
fi
```

The preceding example will set the umask to 007 for users with a UID greater than 199 and with a username and primary group name that match, and to 022 for everyone else. If you just wanted to set the umask for everyone to 022, you could create that file with just the following content:

```
# Overrides default umask configuration
umask 022
```

To ensure that global umask changes take effect you must log out of the shell and log back in. Until that time the umask configured in the current shell is still in effect.

References

bash(1), **ls**(1), **chmod**(1), and **umask**(1) man pages

Guided Exercise

Controlling New File Permissions and Ownership

In this exercise, you will control default permissions on new files using the **umask** command and **setgid** permission.

Outcomes

- Create a shared directory where new files are automatically owned by the group **ateam**.
- Experiment with various umask settings.
- Adjust default permissions for specific users.
- Confirm your adjustment is correct.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

It is also assumed that you have completed the steps from the preceding exercise in this chapter. The users **alice** and **andy** should exist on your system. The primary group for both users should be a user private group with the same name as the user's username. Both users should also be members of the group **ateam**.

Steps

1. Use **sudo** to switch user to **alice**.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su - alice
Last login: Fri Jul 24 17:01:55 EDT 2020 on pts/0
[alice@ip-192-0-2-1 ~]$
```

2. Use the **umask** command without arguments to display Alice's default umask value.

```
[alice@ip-192-0-2-1 ~]$ umask
0002
```

3. Create a new directory **/tmp/shared** and a new file **/tmp/shared/defaults** to see how the default umask affects permissions.

```
[alice@ip-192-0-2-1 ~]$ mkdir /tmp/shared
[alice@ip-192-0-2-1 ~]$ ls -ld /tmp/shared
drwxrwxr-x. 2 alice alice 6 Jul 24 18:43 /tmp/shared
[alice@ip-192-0-2-1 ~]$ touch /tmp/shared/defaults
[alice@ip-192-0-2-1 ~]$ ls -l /tmp/shared/defaults
-rw-rw-r--. 1 alice alice 0 Jul 24 18:43 /tmp/shared/defaults
```

4. Change the group ownership of **/tmp/shared** to **ateam** and record the new ownership and permissions.

```
[alice@ip-192-0-2-1 ~]$ chown :ateam /tmp/shared
[alice@ip-192-0-2-1 ~]$ ls -ld /tmp/shared
drwxrwxr-x. 2 alice ateam 21 Jul 24 18:43 /tmp/shared
```

5. Create a new file in **/tmp/shared** and record the ownership and permissions.

```
[alice@ip-192-0-2-1 ~]$ touch /tmp/shared/alice3
[alice@ip-192-0-2-1 ~]$ ls -l /tmp/shared/alice3
-rw-rw-r--. 1 alice alice 0 Jul 24 18:46 /tmp/shared/alice3
```

6. Ensure the permissions of **/tmp/shared** cause files created in that directory to inherit the group ownership of **ateam**.

```
[alice@ip-192-0-2-1 ~]$ chmod g+s /tmp/shared
[alice@ip-192-0-2-1 ~]$ ls -ld /tmp/shared
drwxrwsr-x. 2 alice ateam 34 Jul 24 18:46 /tmp/shared
[alice@ip-192-0-2-1 ~]$ touch /tmp/shared/alice4
[alice@ip-192-0-2-1 ~]$ ls -l /tmp/shared
total 0
-rw-rw-r--. 1 alice alice 0 Jul 24 18:46 alice3
-rw-rw-r--. 1 alice ateam 0 Jul 24 18:48 alice4
-rw-rw-r--. 1 alice alice 0 Jul 24 18:43 defaults
```

7. Change the umask for **alice** such that new files are created with read-only access for the group and no access for other users. Create a new file and record the ownership and permissions.

```
[alice@ip-192-0-2-1 ~]$ umask 027
[alice@ip-192-0-2-1 ~]$ touch /tmp/shared/alice5
[alice@ip-192-0-2-1 ~]$ ls -l /tmp/shared
total 0
-rw-rw-r--. 1 alice alice 0 Jul 24 18:46 alice3
-rw-rw-r--. 1 alice ateam 0 Jul 24 18:48 alice4
-rw-r-----. 1 alice ateam 0 Jul 24 18:50 alice5
-rw-rw-r--. 1 alice alice 0 Jul 24 18:43 defaults
```

8. Log out and log back in again as **alice**, starting a new shell, and view her umask.

```
[alice@ip-192-0-2-1 ~]$ umask
0027
[alice@ip-192-0-2-1 ~]$ exit
logout
[ec2-user@ip-192-0-2-1 ~]$ sudo su - alice
Last login: Fri Jul 24 18:31:25 EDT 2020 on pts/0
[alice@ip-192-0-2-1 ~]$ umask
0002
```

Note that **alice**'s umask reverted to her default settings.

9. Change the default umask for **alice** to prohibit all access to "other" on files the user creates, by appending **umask 007** to the end of the **~/.bashrc** file.

```
[alice@ip-192-0-2-1 ~]$ echo "umask 007" >> ~/.bashrc
[alice@ip-192-0-2-1 ~]$ cat ~/.bashrc
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
 . /etc/bashrc
fi

# User specific environment
PATH="$HOME/.local/bin:$HOME/bin:$PATH"
export PATH

# Uncomment the following line if you don't like systemctl's auto-paging feature:
# export SYSTEMD_PAGER=

# User specific aliases and functions
umask 007
```

Important

Rather than using redirection, you could instead simply edit the file with the command **vim ~/.bashrc**, and add **umask 007** as the last line of the file, as shown in the output of **cat ~/.bashrc** from the example.

If you are interested in more in-depth information on shell I/O redirection, an overview is available at http://wiki.bash-hackers.org/syntax/redirection.

10. Log out of **alice**'s **su** session, and then log back into **alice**'s account and confirm that the umask changes you made are persistent.

```
[alice@ip-192-0-2-1 ~]$ exit
logout
[ec2-user@ip-192-0-2-1 ~]$ sudo su - alice
Last login: Fri Jul 24 18:54:02 EDT 2020 on pts/0
[alice@ip-192-0-2-1 ~]$ umask
0007
```

11. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Lab

Controlling Access to Files with Linux File System Permissions

In this lab, you will configure a directory that users in the same group can use to easily share files that are automatically writable by the entire group.

Outcomes

- A directory called **/home/operators** where the members of group **operators** can work collaboratively on files.
- Only the **root** user and **operators** group can access, create, and delete files in this directory.
- Files created in this directory should automatically be assigned a group ownership of **operators**.
- New files created in this directory will not be accessible to normal users who are not in the group or own the files.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

Steps

1. Use **sudo** to get an interactive shell as **root**.
2. Create a new group named **operators**.
3. Create three users, **yasmine**, **louis**, and **liza**. The primary group for each of these users should be a user private group that has the same name as their username. They should also be members of group **operators**.

   You may set passwords for these users so that you can use commands like **su - yasmine** to switch between the users to test your work, or you may leave the accounts locked with invalid passwords and switch between them from the **ec2-user** account with **sudo**.
4. Create the **/home/operators** directory.
5. Change group permissions on the **/home/operators** directory so that it belongs to the **operators** group.
6. Set permissions on the **/home/operators** directory so it is set GID, the owner and group have full read/write/execute permissions, and other users have no permissions on the directory.
7. Check that the permissions were set properly.
8. When you finish, check your work to ensure you have done everything correctly.
9. This concludes this lab. Log out and stop your Amazon EC2 instance.

Solution

Controlling Access to Files with Linux File System Permissions

In this lab, you will configure a directory that users in the same group can use to easily share files that are automatically writable by the entire group.

Outcomes

- A directory called **/home/operators** where the members of group **operators** can work collaboratively on files.
- Only the **root** user and **operators** group can access, create, and delete files in this directory.
- Files created in this directory should automatically be assigned a group ownership of **operators**.
- New files created in this directory will not be accessible to normal users who are not in the group or own the files.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

Steps

1. Use **sudo** to get an interactive shell as **root**.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su -
Last login: Fri Jul 24 20:57:08 EDT 2020 on pts/0
[root@ip-192-0-2-1 ~]#
```

2. Create a new group named **operators**.

```
[root@ip-192-0-2-1 ~]# groupadd operators
```

3. Create three users, **yasmine**, **louis**, and **liza**. The primary group for each of these users should be a user private group that has the same name as their username. They should also be members of group **operators**.

   You may set passwords for these users so that you can use commands like **su - yasmine** to switch between the users to test your work, or you may leave the accounts locked with invalid passwords and switch between them from the **ec2-user** account with **sudo**.

```
[root@ip-192-0-2-1 ~]# useradd -G operators yasmine
[root@ip-192-0-2-1 ~]# useradd -G operators louis
[root@ip-192-0-2-1 ~]# useradd -G operators liza
```

4. Create the **/home/operators** directory.

```
[root@ip-192-0-2-1 ~]# mkdir /home/operators
```

5. Change group permissions on the **`/home/operators`** directory so that it belongs to the **`operators`** group.

```
[root@ip-192-0-2-1 ~]# chown :operators /home/operators
```

6. Set permissions on the **`/home/operators`** directory so it is set GID, the owner and group have full read/write/execute permissions, and other users have no permissions on the directory.

```
[root@ip-192-0-2-1 ~]# chmod 2770 /home/operators
```

7. Check that the permissions were set properly.

```
[root@ip-192-0-2-1 ~]# ls -ld /home/operators
drwxrws---. 2 root operators 6 Jul 24 11:38 /home/operators
```

8. When you finish, check your work to ensure you have done everything correctly.

```
[root@ip-192-0-2-1 ~]# exit
logout
[ec2-user@ip-192-0-2-1 ~]$ touch /home/operators/ec2-user-test
touch: cannot touch ‘/home/operators/ec2-user-test’: Permission denied
[ec2-user@ip-192-0-2-1 ~]$ sudo su - yasmine
[yasmine@ip-192-0-2-1 ~]$ touch /home/operators/yasmine-test
[yasmine@ip-192-0-2-1 ~]$ ls -l /home/operators
total 0
-rw-rw-r--. 1 yasmine operators 0 Jul 24 11:47 yasmine-test
[yasmine@ip-192-0-2-1 ~]$ exit
logout
[ec2-user@ip-192-0-2-1 ~]$ sudo su - louis
[louis@ip-192-0-2-1 ~]$ touch /home/operators/louis-test
[louis@ip-192-0-2-1 ~]$ ls -l /home/operators
total 0
-rw-rw-r--. 1 louis   operators 0 Jul 24 11:48 louis-test
-rw-rw-r--. 1 yasmine operators 0 Jul 24 11:47 yasmine-test
[louis@ip-192-0-2-1 ~]$ exit
logout
[ec2-user@ip-192-0-2-1 ~]$
```

9. This concludes this lab. Log out and stop your Amazon EC2 instance.

Chapter 7

Monitoring and Managing Linux Processes

Goal

To evaluate and control processes running on a Red Hat Enterprise Linux system.

Objectives

- List and interpret basic information about processes running on the system.
- Control processes in the shell's session using bash job control.
- Terminate and control processes using signals.
- Monitor resource usage and system load due to process activity.

Sections

- Processes (and Quiz)
- Controlling Jobs (and Guided Exercise)
- Killing Processes (and Guided Exercise)
- Monitoring Processes (and Guided Exercise)

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).

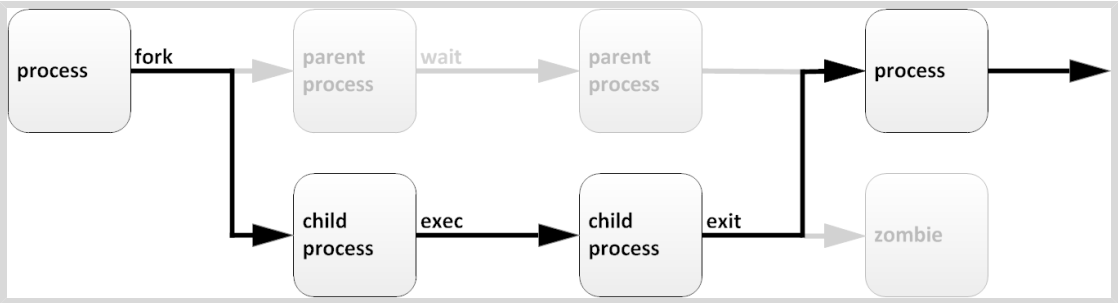


Figure 7.1: Process life cycle

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.

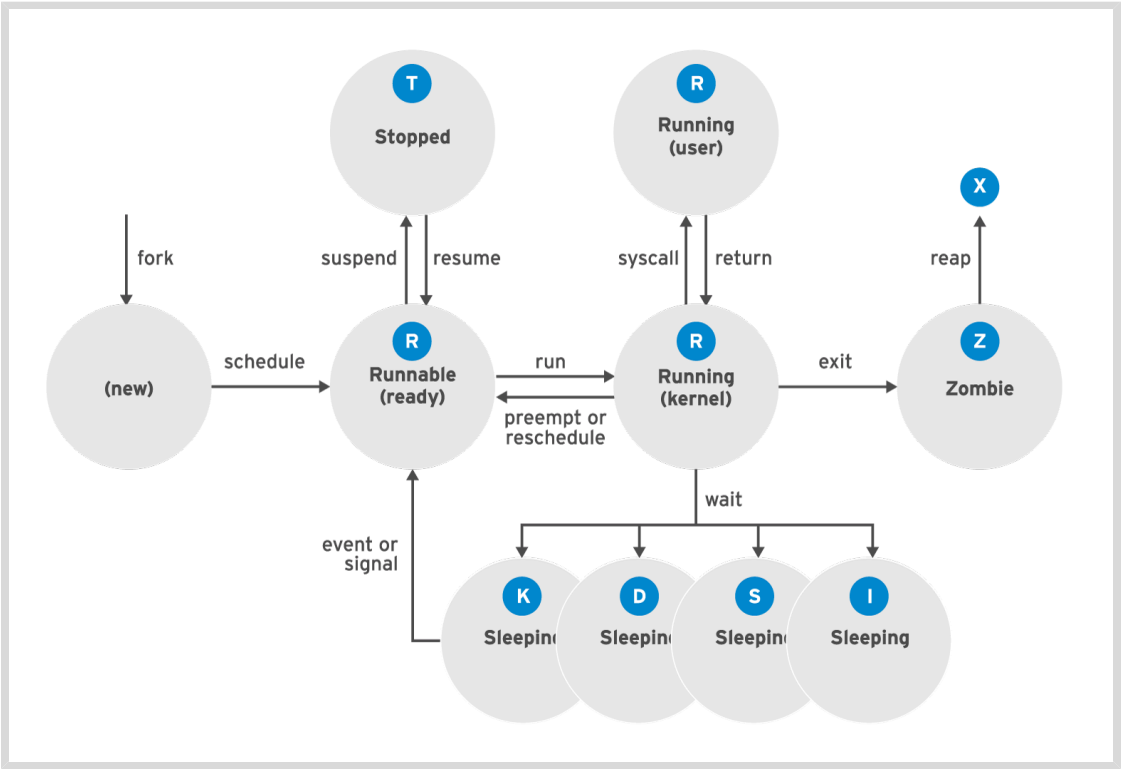


Figure 7.2: Linux process states

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

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.

Name	Flag	Kernel-defined state name and description
	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

Processes

Choose the correct answers to the following questions:

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

a. D
b. R
c. S
d. T
e. Z

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

a. D
b. R
c. S
d. T
e. Z

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

a. exec
b. fork
c. zombie
d. syscall
e. reap

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

a. D
b. R
c. S
d. T
e. Z

Solution

Processes

Choose the correct answers to the following questions:

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

a. D
b. R
c. S
d. T
e. Z

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

a. D
b. R
c. S
d. T
e. Z

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

a. exec
b. fork
c. zombie
d. syscall
e. reap

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

a. D
b. R
c. S
d. T
e. Z

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

Background and Foreground Processes

In this exercise, students will start, suspend, and reconnect to multiple processes using job control.

Outcomes

- Practice suspending and restarting user processes.

Before You Begin

Start your Amazon EC2 instance.

Important

The exercises in this chapter require you to have *two* simultaneous **ssh** login sessions as the user **ec2-user** open to your Red Hat Enterprise Linux instance. The instructions assume that the program you are using to run **ssh** allows you to do this in two separate windows.

For example, macOS users can use Terminal to open two windows each with a shell prompt, which can each use the same **ssh** command and private key to open independent shells on the Red Hat Enterprise Linux instance on Amazon EC2. Other programs and operating systems may require you to access different tabs in the same window or run two separate instances of the same program to get two **ssh** sessions running.

The following instructions refer to your two login sessions as being in a *left window* and a *right window* respectively. If you can arrange the terminals or applications running **ssh** in this way on your personal workstation, this will make the exercises easier to understand. It does not actually matter whether they are really two separate windows or two tabs in the same window as long as you can keep straight which window or tab is the *left window* and which the *right window* from the perspective of the exercise's instructions.

Steps

1. We need a short program to demonstrate job control. In either window, make a new directory named **/home/ec2-user/bin**. In that new directory, use **vim** to create a short shell script named **forever** as shown:

```
#!/bin/bash

while true; do
  echo -n "$@ " >> ~/outfile
  sleep 1
done
```

The **forever** script will run until terminated and will append any command line arguments passed to it to the file **~/outfile**, once per second. Set the executable permission on the file so **ec2-user** can run this script like any other program.

```
[ec2-user@ip-192-0-2-1 ~]$ mkdir bin
[ec2-user@ip-192-0-2-1 ~]$ vim bin/forever
  ...output omitted...
[ec2-user@ip-192-0-2-1 ~]$ cat bin/forever
#!/bin/bash

while true; do
  echo -n "$@ " >> ~/outfile
  sleep 1
done
[ec2-user@ip-192-0-2-1 ~]$ chmod +x bin/forever
```

2. In the left window, use your **forever** script to start a process that continuously appends the word "rock" and a space to the file **~/outfile** at one-second intervals.

```
[ec2-user@ip-192-0-2-1 ~]$ forever rock
```

3. In the right window, use **tail** to confirm that the new process is writing to the file.

```
[ec2-user@ip-192-0-2-1 ~]$ tail -f ~/outfile
```

4. In the left window, suspend the running process by pressing **Ctrl**+**z**. The shell returns the job ID in square brackets. In the right window, confirm that the process output has stopped.

```
^Z
[1]+  Stopped                 forever rock
[ec2-user@ip-192-0-2-1 ~]$
```

The **tail** command running in the right window should show that **rock** is no longer being appended to the **~/outfile** every second.

5. In the left window, view the **jobs** list. The **+** denotes the *current job*. Restart the job in the background. In the right window, confirm that the process output is again active.

```
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Stopped                 forever rock
[ec2-user@ip-192-0-2-1 ~]$ bg
[1]+ forever rock &
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Running                 forever rock &
```

6. In the left window, use **forever** to start two more processes which append strings to **~/outfile**. Use the ampersand (**&**) to start the processes in the background, and use the arguments **paper** and **scissors** in place of **rock** so you can tell all three processes apart.

```
[ec2-user@ip-192-0-2-1 ~]$ forever paper &
[2] 11986
[ec2-user@ip-192-0-2-1 ~]$ forever scissors &
[3] 12010
```

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, which will probably be different when you run these commands.

7. In the left window, view **jobs** to see all three processes "Running". In the right window, confirm that all three processes are appending to the file.

```
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]   Running                 forever rock &
[2]-  Running                 forever paper &
[3]+  Running                 forever scissors &
```

8. Suspend the **forever rock** process. In the left window, foreground the job, using the job ID determined from the **jobs** listing, then suspend it using **Ctrl**+**z**. Confirm that the **forever rock** process is "**Stopped**. In the right window, confirm that "rock" is no longer being appended to **~/outfile** every second.

```
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]   Running                 forever rock &
[2]-  Running                 forever paper &
[3]+  Running                 forever scissors &
[ec2-user@ip-192-0-2-1 ~]$ fg %1
forever rock
^Z
[1]+  Stopped                 forever rock
[ec2-user@ip-192-0-2-1 ~]$
```

9. Terminate the **forever paper** process. In the left window, foreground the job, then terminate it using **Ctrl**+**c**. Confirm that the **forever paper** process has disappeared using **jobs**. In the right window, confirm that **forever paper** output is no longer active.

```
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Stopped                 forever rock
[2]   Running                 forever paper &
[3]-  Running                 forever scissors &
[ec2-user@ip-192-0-2-1 ~]$ fg %2
forever paper
^C
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Stopped                 forever rock
[3]-  Running                 forever scissors &
[ec2-user@ip-192-0-2-1 ~]$
```

10. In the left window, view the remaining jobs associated with the left window session by using the **ps jT** command. The suspended job has state **T**. The other background job is sleeping (**S**), since **ps** is "on cpu" (**R**) while displaying.

```
[ec2-user@ip-192-0-2-1 ~]$ ps jT
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2773 10960 10960 10960 pts/0    16325 Ss    1000   0:00 -bash
10960 11029 11029 10960 pts/0    16325 T     1000   0:00 /bin/bash /home/ec2-use
10960 12010 12010 10960 pts/0    16325 S     1000   0:00 /bin/bash /home/ec2-use
11029 14167 11029 10960 pts/0    16325 T     1000   0:00 sleep 1
12010 16323 12010 10960 pts/0    16325 S     1000   0:00 sleep 1
10960 16325 16325 10960 pts/0    16325 R+    1000   0:00 ps jT
```

11. Terminate the remaining two jobs. In the left window, foreground either job. Terminate it using **Ctrl+c**. Repeat with the remaining job. The **Stopped** job will temporarily restart when foregrounded. Confirm that no jobs remain and that output has stopped.

```
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Stopped                 forever rock
[3]-  Running                 forever scissors &
[ec2-user@ip-192-0-2-1 ~]$ fg %1
forever rock
^C
[ec2-user@ip-192-0-2-1 ~]$ fg %3
forever scissors
^C
[ec2-user@ip-192-0-2-1 ~]$ jobs
[ec2-user@ip-192-0-2-1 ~]$
```

12. In the right window, use **Ctrl+c** to terminate the **tail** command. Remove **~/outfile**.

```
[ec2-user@ip-192-0-2-1 ~]$ rm ~/outfile
```

13. This concludes this exercise. Log out of both sessions and stop your Amazon EC2 instance.

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.

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.
15 *default*	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, students will use keyboard sequences and signals to manage and stop processes.

Outcomes

- Experience with observing the results of starting and stopping multiple shell processes.

Before You Begin

Start your Amazon EC2 instance.

This exercise assumes that you have completed the preceding exercise. You will need two terminals running independent **`ssh`** sessions as **`ec2-user`** on your Red Hat Enterprise Linux instance again, which will be referred to as your *left window* and your *right window*.

This exercise also assumes that you still have the program to demonstrate job control from the preceding exercise. This is the file **`/home/ec2-user/bin/forever`**, which should have the executable permission set and should contain the content

```
#!/bin/bash

while true; do
  echo -n "$@ " >> ~/outfile
  sleep 1
done
```

Steps

1. In the left window, start three **`forever`** processes that append text to **`~/outfile`** at one-second intervals.

```
[ec2-user@ip-192-0-2-1 ~]$ forever game &
[1] 9613
[ec2-user@ip-192-0-2-1 ~]$ forever set &
[2] 9618
[ec2-user@ip-192-0-2-1 ~]$ forever match &
[3] 9627
[ec2-user@ip-192-0-2-1 ~]$
```

   Your processes will probably have different PID numbers than the example above.

2. In the right window, use **`tail`** to confirm that all three processes are appending to the file.

```
[ec2-user@ip-192-0-2-1 ~]$ tail -f ~/outfile
```

3. In the left window, view **`jobs`** to see all three processes in state **`Running`**.

```
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]   Running                 forever game &
[2]-  Running                 forever set &
[3]+  Running                 forever match &
[ec2-user@ip-192-0-2-1 ~]$
```

4. Suspend the **`forever game`** process using signals. Confirm that the "game" process is **`Stopped`**. In the right window, confirm that **`game`** output is no longer being appended to **`~/outfile`**.

```
[ec2-user@ip-192-0-2-1 ~]$ kill -SIGSTOP %1
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Stopped                 forever game
[2]   Running                 forever set &
[3]-  Running                 forever match &
```

5. Terminate the **`forever set`** process using signals. Confirm that the **`forever set`** job has disappeared. In the right window, confirm that **`set`** output is no longer being appended to **`~/outfile`**.

```
[ec2-user@ip-192-0-2-1 ~]$ kill -SIGTERM %2
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Stopped                 forever game
[2]   Terminated              forever set
[3]-  Running                 forever match &
```

6. Continue the **`forever game`** process using signals. Confirm that the **`forever game`** process is **`Running`**. In the right window, confirm that **`game`** output is again being appended to **`~/outfile`**.

```
[ec2-user@ip-192-0-2-1 ~]$ kill -SIGCONT %1
[ec2-user@ip-192-0-2-1 ~]$ jobs
[1]+  Running                 forever game &
[3]-  Running                 forever match &
```

7. Terminate the remaining two jobs. Confirm that no jobs remain and that output has stopped. From the left window, use **`pkill`** to terminate all **`tail`** commands running as **`ec2-user`** by name. Remove **`~/outfile`**.

```
[ec2-user@ip-192-0-2-1 ~]$ kill -SIGTERM %1
[ec2-user@ip-192-0-2-1 ~]$ kill -SIGTERM %3
[1]+  Terminated              forever game
[ec2-user@ip-192-0-2-1 ~]$ jobs
[3]+  Terminated              forever match
[ec2-user@ip-192-0-2-1 ~]$ pkill -SIGTERM tail
[ec2-user@ip-192-0-2-1 ~]$ rm ~/outfile
```

Note

Note that the shell reported the termination of **`forever game`** automatically when the next command was run. In fact, if we'd simply pressed **`Enter`** to get a new shell prompt after the second **`kill`** command, the shell would have automatically reported the termination of job 3.

8. This concludes this exercise. Log out of both sessions and stop your Amazon EC2 instance.

Monitoring Processes

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**).

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.

Key	Purpose
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, students will use the **top** command to dynamically view, sort, and stop processes.

Outcomes

- Practice with managing processes in real time.

Before You Begin

Start your Amazon EC2 instance.

This exercise assumes that you have completed the preceding exercises in this chapter. You will need two terminals running independent **ssh** sessions as **ec2-user** on your Red Hat Enterprise Linux instance again, which will be referred to as your *left window* and your *right window*.

Steps

1. We need a short program which will generate artificial CPU load. In either window, use **vim** to create a file named **/home/ec2-user/bin/process101** which contains a short shell script as shown:

```
#!/bin/bash

while true; do
  var=1
  while [[ var -lt 50000 ]]; do
    var=$(($var+1))
  done
  sleep 1
done
```

   The **process101** script will endlessly perform fifty thousand addition problems, then sleep for one second, then reset the variable and repeat.

   Set the executable permission on the file so that **ec2-user** can run this script like any other program.

```
[ec2-user@ip-192-0-2-1 ~]$ vim bin/process101
  ...output omitted...
[ec2-user@ip-192-0-2-1 ~]$ cat bin/process101
#!/bin/bash

while true; do
  var=1
  while [[ var -lt 50000 ]]; do
    var=$(($var+1))
  done
```

```
  sleep 1
done
[ec2-user@ip-192-0-2-1 ~]$ chmod +x bin/process101
```

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

```
[ec2-user@ip-192-0-2-1 ~]$ top
```

3. In the left window, determine the number of logical CPUs on this virtual machine.

```
[ec2-user@ip-192-0-2-1 ~]$ grep "model name" /proc/cpuinfo | wc -l
1
```

4. In the left window, run a single instance of the **process101** executable.

```
[ec2-user@ip-192-0-2-1 ~]$ process101 &
[1] 32105
```

5. In the right window, 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.

6. Note the process ID (PID) for **process101**. View the CPU percentage for the process, which is expected to hover around 15% or 30%.

   View the load averages. On a single-CPU virtual machine, for example, the one-minute load average is currently less than a value of 1. The value observed may be affected by resource contention from other processes you may be running.

7. In the left window, run a second instance of **process101**.

```
[ec2-user@ip-192-0-2-1 ~]$ process101 &
[2] 32252
```

8. In **top**, note the process ID (PID) for the second **process101**. View the CPU percentage for the process, also expected to hover around 15% or 30%.

   View the one-minute load average again, which may still be less than 1. Wait up to one minute to allow the calculation to adjust to the new workload.

9. In the left window, run a third instance of **process101**.

```
[ec2-user@ip-192-0-2-1 ~]$ process101 &
[3] 32408
```

10. Note the process ID (PID) for the third **process101**. View the CPU percentage for each of the **process101** processes. It may begin to drop as they each contend for the single CPU on the system.

    View the one-minute load average again, which may now be above 1. Wait up to one minute to allow the calculation to again adjust to the new workload.

11. When finished observing the load average values, terminate each of the **process101** processes from within **top**.

    11.1. Press **k**. Observe the prompt below the headers and above the columns. (The default pid will probably be different on your system.)

```
PID to signal/kill [default pid = 32105]
```

    11.2. Type the PID for one of the **process101** instances. Press **Enter**.

    11.3. Press **Enter** again to use the default **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.

12. Repeat the previous step for each remaining **process101** instance. Confirm that no **process101** instances remain in **top**.

13. In the right window, press **q** to exit **top**.

14. This concludes this exercise. Log out of both sessions and stop your Amazon EC2 instance.

Chapter 8

Installing and Updating Software Packages

Goal

To download, install, update, and manage software packages from Red Hat and YUM package repositories.

Objectives

- Explain what an RPM package is and how RPM packages are used to manage software on a Red Hat Enterprise Linux system.
- Find, install, and update software packages using the **`yum`** command.

Sections

- RPM Software Packages and Yum (and Quiz)
- Managing Software Updates with Yum (and Guided Exercise)

RPM Software Packages and Yum

Objectives

After completing this section, you should be able to explain how software is provided as RPM packages, and investigate the packages installed on the system with Yum and RPM.

Software packages and RPM

The RPM Package Manager, originally developed by Red Hat, provides a standard way to package software for distribution. Managing software in the form of *RPM packages* is much simpler than working with software that has simply been extracted into a file system from an archive. It lets administrators track which files were installed by the software package and which ones need to be removed if it is uninstalled, and check to ensure that supporting packages are present when it is installed. Information about installed packages is stored in a local RPM database on each system. All software provided by Red Hat for Red Hat Enterprise Linux is provided as an RPM package.

RPM package files names consist of four elements (plus the **`.rpm`** suffix): **`name-version-release.architecture`**:

Figure 8.1: RPM file name elements

- NAME is one or more words describing the contents (coreutils).
- VERSION is the version number of the original software (8.30).
- RELEASE is the release number of the package based on that version, and is set by the packager, who might not be the original software developer (4.el8).
- ARCH is the processor architecture the package was compiled to run on. **`noarch`** indicates that this package's contents are not architecture-specific (as opposed to **`x86_64`** for 64-bit, **`aarch64`** for 64-bit ARM, and so on).

Only the package name is required for installing packages from repositories. If multiple versions exist, the package with the higher version number is installed. If multiple releases of a single version exist, the package with the higher release number is installed.

Each RPM package is a special archive made up of three components:

- The files installed by the package.
- Information about the package (metadata), such as the name, version, release, and arch; a summary and description of the package; whether it requires other packages to be installed; licensing; a package change log; and other details.

- Scripts that may run when this package is installed, updated, or removed, or are triggered when other packages are installed, updated, or removed.

Typically, software providers digitally sign RPM packages using GPG keys (Red Hat digitally signs all packages it releases). The RPM system verifies package integrity by confirming that the package was signed by the appropriate GPG key. The RPM system refuses to install a package if the GPG signature does not match.

Updating Software with RPM Packages

Red Hat generates a complete RPM package to update software. An administrator installing that package gets only the most recent version of the package. Red Hat does not require that older packages be installed and then patched. To update software, RPM removes the older version of the package and installs the new version. Updates usually retain configuration files, but the packager of the new version defines the exact behavior.

In most cases, only one version or release of a package may be installed at a time. However, if a package is built so that there are no conflicting file names, then multiple versions may be installed. The most important example of this is the **`kernel`** package. Since a new kernel can only be tested by booting to that kernel, the package is specifically designed so that multiple versions may be installed at once. If the new kernel fails to boot, the old kernel is still available and bootable.

References

`rpm`(8) man page

Quiz

RPM Software Packages

Choose the correct answer to the following questions:

1. **Which portion of an RPM references the version of the upstream course code?**
   a. Repository
   b. Version
   c. Release
   d. Changelog

2. **Which portion of an RPM lists the reasons for each package build?**
   a. Errata
   b. Version
   c. Release
   d. Changelog

3. **Which portion of an RPM references the version of the package build?**
   a. Errata
   b. Version
   c. Release
   d. Changelog

4. **Which portion of an RPM references the processor type required for a specific package?**
   a. Architecture
   b. Release
   c. Version
   d. Errata

5. **Which term is used to descibe a collection of RPM packages and package groups?**
   a. Software Collection
   b. Distribution
   c. Repository
   d. Changelog

6. **Which term is used to verify the source and integrity of a package?**
   a. Release
   b. GPG signature
   c. Repository Certificate
   d. Changelog

Solution

RPM Software Packages

Choose the correct answer to the following questions:

1. Which portion of an RPM references the version of the upstream course code?

a. Repository
b. Version
c. Release
d. Changelog

2. Which portion of an RPM lists the reasons for each package build?

a. Errata
b. Version
c. Release
d. Changelog

3. Which portion of an RPM references the version of the package build?

a. Errata
b. Version
c. Release
d. Changelog

4. Which portion of an RPM references the processor type required for a specific package?

a. Architecture
b. Release
c. Version
d. Errata

5. Which term is used to descibe a collection of RPM packages and package groups?

a. Software Collection
b. Distribution
c. Repository
d. Changelog

6. Which term is used to verify the source and integrity of a package?

a. Release
b. GPG signature
c. Repository Certificate
d. Changelog

Managing Software Updates with Yum

Objectives

After completing this section, you should be able to find, install, and update software packages, using the **yum** command.

Managing Software Packages with Yum

The low-level **rpm** command can be used to install packages, but it is not designed to work with package repositories or resolve dependencies from multiple sources automatically.

Yum is designed to be a better system for managing RPM-based software installation and updates. The **yum** command allows you to install, update, remove, and get information about software packages and their dependencies. You can get a history of transactions performed and work with multiple Red Hat and third-party software repositories.

Finding Software with Yum

- **yum help** displays usage information.
- **yum list** displays installed and available packages.

```
[user@host ~]$ yum list 'http*'
Available Packages
http-parser.i686              2.8.0-2.el8                      rhel8-appstream
http-parser.x86_64            2.8.0-2.el8                      rhel8-appstream
httpcomponents-client.noarch  4.5.5-4.module+el8+2452+b359bfcd  rhel8-appstream
httpcomponents-core.noarch    4.4.10-3.module+el8+2452+b359bfcd  rhel8-appstream
httpd.x86_64                  2.4.37-7.module+el8+2443+605475b7  rhel8-appstream
httpd-devel.x86_64            2.4.37-7.module+el8+2443+605475b7  rhel8-appstream
httpd-filesystem.noarch       2.4.37-7.module+el8+2443+605475b7  rhel8-appstream
httpd-manual.noarch           2.4.37-7.module+el8+2443+605475b7  rhel8-appstream
httpd-tools.x86_64            2.4.37-7.module+el8+2443+605475b7  rhel8-appstream
```

- **yum search *KEYWORD*** lists packages by keywords found in the name and summary fields only.

  To search for packages that have "web server" in their name, summary, and description fields, use **search all**:

```
[user@host ~]$ yum search all 'web server'
================= Summary & Description Matched: web server ====================
pcp-pmda-weblog.x86_64 : Performance Co-Pilot (PCP) metrics from web server logs
nginx.x86_64 : A high performance web server and reverse proxy server
======================== Summary Matched: web server ===========================
libcurl.x86_64 : A library for getting files from web servers
libcurl.i686 : A library for getting files from web servers
libcurl.x86_64 : A library for getting files from web servers
====================== Description Matched: web server =========================
httpd.x86_64 : Apache HTTP Server
```

```
git-instaweb.x86_64 : Repository browser in gitweb
...output omitted...
```

- **`yum info`** ***`PACKAGENAME`*** returns detailed information about a package, including the disk space needed for installation.

  To get information on the Apache HTTP Server:

```
[user@host ~]$ yum info httpd
Available Packages
Name         : httpd
Version      : 2.4.37
Release      : 7.module+el8+2443+605475b7
Arch         : x86_64
Size         : 1.4 M
Source       : httpd-2.4.37-7.module+el8+2443+605475b7.src.rpm
Repo         : rhel8-appstream
Summary      : Apache HTTP Server
URL          : https://httpd.apache.org/
License      : ASL 2.0
Description  : The Apache HTTP Server is a powerful, efficient, and extensible
             : web server.
```

- **`yum provides`** ***`PATHNAME`*** displays packages that match the path name specified (which often include wildcard characters).

  To find packages that provide the **`/var/www/html`** directory, use:

```
[user@host ~]$ yum provides /var/www/html
httpd-filesystem-2.4.37-7.module+el8+2443+605475b7.noarch : The basic directory
 layout for the Apache HTTP server
Repo        : rhel8-appstream
Matched from:
Filename    : /var/www/html
```

Installing and removing software with yum

- **`yum install`** ***`PACKAGENAME`*** obtains and installs a software package, including any dependencies.

```
[user@host ~]$ yum install httpd
Dependencies resolved.
================================================================================
 Package                 Arch       Version             Repository        Size
================================================================================
Installing:
 httpd                   x86_64     2.4.37-7.module...  rhel8-appstream   1.4 M
Installing dependencies:
 apr                     x86_64     1.6.3-8.el8         rhel8-appstream   125 k
 apr-util                x86_64     1.6.1-6.el8         rhel8-appstream   105 k
...output omitted...
Transaction Summary
================================================================================
Install  9 Packages
```

```
Total download size: 2.0 M
Installed size: 5.4 M
Is this ok [y/N]: y
Downloading Packages:
(1/9): apr-util-bdb-1.6.1-6.el8.x86_64.rpm          464 kB/s |  25 kB     00:00
(2/9): apr-1.6.3-8.el8.x86_64.rpm                   1.9 MB/s | 125 kB     00:00
(3/9): apr-util-1.6.1-6.el8.x86_64.rpm              1.3 MB/s | 105 kB     00:00
...output omitted...
Total                                               8.6 MB/s | 2.0 MB     00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                        1/1
  Installing       : apr-1.6.3-8.el8.x86_64                                 1/9
  Running scriptlet: apr-1.6.3-8.el8.x86_64                                 1/9
  Installing       : apr-util-bdb-1.6.1-6.el8.x86_64                        2/9
...output omitted...
Installed:
  httpd-2.4.37-7.module+el8+2443+605475b7.x86_64 apr-util-bdb-1.6.1-6.el8.x86_64
  apr-util-openssl-1.6.1-6.el8.x86_64            apr-1.6.3-8.el8.x86_64
...output omitted...
Complete!
```

- **`yum update`** ***`PACKAGENAME`*** obtains and installs a newer version of the specified package, including any dependencies. Generally the process tries to preserve configuration files in place, but in some cases, they may be renamed if the packager thinks the old one will not work after the update. With no PACKAGENAME specified, it installs all relevant updates.

```
[user@host ~]$ sudo yum update
```

Since a new kernel can only be tested by booting to that kernel, the package is specifically designed so that multiple versions may be installed at once. If the new kernel fails to boot, the old kernel is still available. Using **`yum update kernel`** will actually *install* the new kernel. The configuration files hold a list of packages to *always install* even if the administrator requests an update.

Note

Use **yum list kernel** to list all installed and available kernels. To view the currently running kernel, use the **uname** command. The **-r** option only shows the kernel version and release, and the **-a** option shows the kernel release and additional information.

```
[user@host ~]$ yum list kernel
Installed Packages
kernel.x86_64          4.18.0-60.el8          @anaconda
kernel.x86_64          4.18.0-67.el8          @rhel-8-for-x86_64-baseos-htb-rpms
[user@host ~]$ uname -r
4.18.0-60.el8.x86_64
[user@host ~]$ uname -a
Linux host.lab.example.com 4.18.0-60.el8.x86_64 #1 SMP Fri Jan 11 19:08:11 UTC
 2019 x86_64 x86_64 x86_64 GNU/Linux
```

- **yum remove *PACKAGENAME*** removes an installed software package, including any supported packages.

```
[user@host ~]$ sudo yum remove httpd
```

Warning

The **yum remove** command removes the packages listed *and any package that requires the packages being removed* (and packages which require those packages, and so on). This can lead to unexpected removal of packages, so carefully review the list of packages to be removed.

Installing and removing groups of software with yum

- **yum** also has the concept of *groups*, which are collections of related software installed together for a particular purpose. In Red Hat Enterprise Linux 8, there are two kinds of groups. Regular groups are collections of packages. *Environment groups* are collections of regular groups. The packages or groups provided by a group may be **mandatory** (they must be installed if the group is installed), **default** (normally installed if the group is installed), or **optional** (not installed when the group is installed, unless specifically requested).

  Like **yum list**, the **yum group list** command shows the names of installed and available groups.

```
[user@host ~]$ yum group list
Available Environment Groups:
   Server with GUI
   Minimal Install
   Server
...output omitted...
Available Groups:
   Container Management
   .NET Core Development
```

```
   RPM Development Tools
...output omitted...
```

Some groups are normally installed through environment groups and are hidden by default. List these hidden groups with the **`yum group list hidden`** command.

- **`yum group info`** displays information about a group. It includes a list of mandatory, default, and optional package names.

```
[user@host ~]$ yum group info "RPM Development Tools"
Group: RPM Development Tools
 Description: These tools include core development tools such rpmbuild.
  Mandatory Packages:
    redhat-rpm-config
    rpm-build
  Default Packages:
    rpmdevtools
  Optional Packages:
    rpmlint
```

- **`yum group install`** installs a group that installs its mandatory and default packages and the packages they depend on.

```
[user@host ~]$ sudo yum group install "RPM Development Tools"
...output omitted...
Installing Groups:
 RPM Development Tools

 Transaction Summary
 ================================================================================
 Install  64 Packages

 Total download size: 21 M
 Installed size: 62 M
 Is this ok [y/N]: y
...output omitted...
```

Important

The behavior of Yum groups changed starting in Red Hat Enterprise Linux 7. In RHEL 7 and later, groups are treated as *objects*, and are tracked by the system. If an installed group is updated, and new mandatory or default packages have been added to the group by the Yum repository, those new packages are installed upon update.

RHEL 6 and earlier consider a group to be installed if all its mandatory packages have been installed, or if it had no mandatory packages, or if any default or optional packages in the group are installed. Starting in RHEL 7, a group is considered to be installed *only* if **`yum group install`** was used to install it. The command **`yum group mark install GROUPNAME`** can be used to mark a group as installed, and any missing packages and their dependencies are installed upon the next update.

Finally, RHEL 6 and earlier did not have the two-word form of the **`yum group`** commands. In other words, in RHEL 6 the command **`yum grouplist`** existed, but the equivalent RHEL 7 and RHEL 8 command **`yum group list`** did not.

Viewing transaction history

- All install and remove transactions are logged in **`/var/log/dnf.rpm.log`**.

```
[user@host ~]$ tail -n 5 /var/log/dnf.rpm.log
2019-02-26T18:27:00Z SUBDEBUG Installed: rpm-build-4.14.2-9.el8.x86_64
2019-02-26T18:27:01Z SUBDEBUG Installed: rpm-build-4.14.2-9.el8.x86_64
2019-02-26T18:27:01Z SUBDEBUG Installed: rpmdevtools-8.10-7.el8.noarch
2019-02-26T18:27:01Z SUBDEBUG Installed: rpmdevtools-8.10-7.el8.noarch
2019-02-26T18:38:40Z INFO --- logging initialized ---
```

- **`yum history`** displays a summary of install and remove transactions.

```
[user@host ~]$ sudo yum history
ID     | Command line             | Date and time    | Action(s)      | Altered
-------------------------------------------------------------------------------
     7 | group install RPM Develo | 2019-02-26 13:26 | Install        |   65
     6 | update kernel            | 2019-02-26 11:41 | Install        |    4
     5 | install httpd            | 2019-02-25 14:31 | Install        |    9
     4 | -y install @base firewal | 2019-02-04 11:27 | Install        |  127 EE
     3 | -C -y remove firewalld - | 2019-01-16 13:12 | Removed        |   11 EE
     2 | -C -y remove linux-firmw | 2019-01-16 13:12 | Removed        |    1
     1 |                          | 2019-01-16 13:05 | Install        |  447 EE
```

- The **`history undo`** option reverses a transaction.

```
[user@host ~]$ sudo yum history undo 5
Undoing transaction 7, from Tue 26 Feb 2019 10:40:32 AM EST
    Install apr-1.6.3-8.el8.x86_64                              @rhel8-appstream
    Install apr-util-1.6.1-6.el8.x86_64                         @rhel8-appstream
    Install apr-util-bdb-1.6.1-6.el8.x86_64                     @rhel8-appstream
    Install apr-util-openssl-1.6.1-6.el8.x86_64                 @rhel8-appstream
    Install httpd-2.4.37-7.module+el8+2443+605475b7.x86_64      @rhel8-appstream
...output omitted...
```

Summary of Yum Commands

Packages can be located, installed, updated, and removed by name or by package groups.

Task:	Command:
List installed and available packages by name	`yum list [NAME-PATTERN]`
List installed and available groups	`yum group list`
Search for a package by keyword	`yum search KEYWORD`
Show details of a package	`yum info PACKAGENAME`
Install a package	`yum install PACKAGENAME`
Install a package group	`yum group install GROUPNAME`
Update all packages	`yum update`
Remove a package	`yum remove PACKAGENAME`
Display transaction history	`yum history`

References

yum(1) and **yum.conf**(5) man pages

For more information, refer to the *Installing software packages* chapter in the *Red Hat Enterprise Linux 8 Configuring basic system settings* guide at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/configuring_basic_system_settings/index#installing-software-packages_managing-software-packages

Guided Exercise

Installing and Updating Software with Yum

In this exercise, you will install and remove software packages and package groups.

Outcomes

- Install and remove packages which have dependencies.

Before You Begin

Start your Amazon EC2 instance and use **ssh** to log in as the user **ec2-user**. It is assumed that **ec2-user** can use **sudo** to run commands as **root**.

It is also assumed that the AMI that you are using for your Amazon EC2 instance has been pre-configured with a Red Hat Enterprise Linux subscription and can get software packages and updates from the Red Hat Update Infrastructure (RHUI) in AWS. This should be the case for all official AMIs as discussed at https://aws.amazon.com/partners/redhat/faqs/.

Steps

1. Attempt to run the **gnuplot** command.

   1.1. Attempt to run the **gnuplot** command. You should find that it is not installed.

```
[ec2-user@ip-192-0-2-1 ~]$ gnuplot
-bash: gnuplot: command not found
```

2. Search for the package that provides **gnuplot**.

   2.1. Get an interactive shell as the **root** user.

```
[ec2-user@ip-192-0-2-1 ~]$ sudo su -
Last login: Fri Jul 24 15:42:55 EDT 2020 on pts/0
[root@ip-192-0-2-1 ~]#
```

   2.2. Search for plotting packages.

```
[root@ip-192-0-2-1 ~]# yum search plot
Last metadata expiration check: 0:51:39 ago on Tue 28 Jul 2020 07:00:56 AM UTC.
======================== Name & Summary Matched: plot ========================
gnuplot-common.x86_64 : The common gnuplot parts
texlive-pst-plot.noarch : Plot data using PSTricks
gnuplot.x86_64 : A program for plotting mathematical expressions and data
[root@ip-192-0-2-1 ~]#
```

   2.3. Find out more information about the *gnuplot* package.

```
[root@ip-192-0-2-1 ~]# yum info gnuplot
Last metadata expiration check: 0:51:53 ago on Tue 28 Jul 2020 07:00:56 AM UTC.
Available Packages
Name         : gnuplot
Version      : 5.2.4
Release      : 1.el8
Architecture : x86_64
Size         : 892 k
Source       : gnuplot-5.2.4-1.el8.src.rpm
Repository   : rhel-8-for-x86_64-appstream-rpms
Summary      : A program for plotting mathematical expressions and data
URL          : http://www.gnuplot.info/
License      : gnuplot and MIT
Description  : Gnuplot is a command-line driven, interactive function plotting
             : program especially suited for scientific data representation.
             : Gnuplot can be used to plot functions and data points in both
             : two and three dimensions and in many different formats.
             :
             : Install gnuplot if you need a graphics package for scientific
             : data representation.
             :
             : This package provides a Qt based terminal version of gnuplot.

[root@ip-192-0-2-1 ~]#
```

3. Install the *gnuplot* package.

```
[root@ip-192-0-2-1 ~]# yum install gnuplot
Last metadata expiration check: 0:53:39 ago on Tue 28 Jul 2020 07:00:56 AM UTC.
Dependencies resolved.
================================================================================
 Package          Arch   Version        Repository                       Size
================================================================================
Installing:
 gnuplot          x86_64 5.2.4-1.el8    rhel-8-for-x86_64-appstream-rpms 892 k
Installing dependencies:
 avahi-libs       x86_64 0.7-19.el8     rhel-8-for-x86_64-baseos-rpms     63 k
 cairo            x86_64 1.15.12-3.el8  rhel-8-for-x86_64-appstream-rpms 721 k
...output omitted...
Transaction Summary
================================================================================
Install  59 Packages

Total download size: 28 M
Installed size: 88 M
Is this ok [y/N]: y
Downloading Packages:
(1/59): fontpackages-filesystem-1.44-22.el8.noa  39 kB/s |  16 kB     00:00
(2/59): dejavu-fonts-common-2.35-6.el8.noarch.r 157 kB/s |  74 kB     00:00
...output omitted...
  Installing       : libjpeg-turbo-1.5.3-10.el8.x86_64                    1/59
  Installing       : libX11-xcb-1.6.8-3.el8.x86_64                        2/59
  Installing       : mesa-libglapi-19.3.4-2.el8.x86_64                    3/59
```

```
...output omitted...
  Verifying        : fontpackages-filesystem-1.44-22.el8.noarch           1/59
  Verifying        : dejavu-fonts-common-2.35-6.el8.noarch                2/59
  Verifying        : dejavu-sans-fonts-2.35-6.el8.noarch                  3/59
...output omitted...
Installed:
  avahi-libs-0.7-19.el8.x86_64
  cairo-1.15.12-3.el8.x86_64
  cups-libs-1:2.2.6-33.el8.x86_64
...output omitted...
Complete!
[root@ip-192-0-2-1 ~]#
```

4. Test the **`gnuplot`** command.

```
[root@ip-192-0-2-1 ~]# gnuplot

 G N U P L O T
 Version 5.2 patchlevel 4    last modified 2018-06-01

 Copyright (C) 1986-1993, 1998, 2004, 2007-2018
 Thomas Williams, Colin Kelley and many others

 gnuplot home:     http://www.gnuplot.info
 faq, bugs, etc:   type "help FAQ"
 immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> quit
[root@ip-192-0-2-1 ~]#
```

5. Test the **`yum remove`** command.

5.1. Use the **`yum remove`** command to remove the *gnuplot* package, but respond with **no** when prompted. How many packages would be removed?

```
[root@ip-192-0-2-1 ~]# yum remove gnuplot
Dependencies resolved.
================================================================================
 Package          Arch   Version       Repository                          Size
================================================================================
Removing:
 gnuplot          x86_64 5.2.4-1.el8   @rhel-8-for-x86_64-appstream-rpms 2.1 M
Removing unused dependencies:
 avahi-libs       x86_64 0.7-19.el8    @rhel-8-for-x86_64-baseos-rpms    159 k
 cairo            x86_64 1.15.12-3.el8 @rhel-8-for-x86_64-appstream-rpms 1.8 M
...output omitted...
Transaction Summary
================================================================================
Remove  59 Packages

Freed space: 88 M
```

```
Is this ok [y/N]: n
Operation aborted.
[root@ip-192-0-2-1 ~]#
```

5.2. Use the **`yum remove`** command to remove the *gnuplot-common* package, but respond with **no** when prompted. How many packages would be removed?

```
[root@ip-192-0-2-1 ~]# yum remove gnuplot-common
Dependencies resolved.
================================================================================
 Package          Arch   Version       Repository                        Size
================================================================================
Removing:
 gnuplot-common   x86_64 5.2.4-1.el8   @rhel-8-for-x86_64-appstream-rpms 1.7 M
Removing dependent packages:
 gnuplot          x86_64 5.2.4-1.el8   @rhel-8-for-x86_64-appstream-rpms 2.1 M
Removing unused dependencies:
 avahi-libs       x86_64 0.7-19.el8    @rhel-8-for-x86_64-baseos-rpms    159 k
 cairo            x86_64 1.15.12-3.el8 @rhel-8-for-x86_64-appstream-rpms 1.8 M
...output omitted...
Transaction Summary
================================================================================
Remove  59 Packages

Freed space: 88 M
Is this ok [y/N]: n
Operation aborted.
[root@ip-192-0-2-1 ~]#
```

6. Gather information about the "RPM Development Tools" component group and install it.

6.1. Use the **`yum group list`** command to list all available component groups.

```
[root@ip-192-0-2-1 ~]# yum group list
Last metadata expiration check: 0:55:45 ago on Tue 28 Jul 2020 07:00:56 AM UTC.
Available Environment Groups:
   Server with GUI
   Server
   Minimal Install
   Workstation
   Virtualization Host
   Custom Operating System
Available Groups:
   RPM Development Tools
   Container Management
   .NET Core Development
   Graphical Administration Tools
   Network Servers
   System Tools
   Scientific Support
   Smart Card Support
   Headless Management
   Development Tools
```

```
   Legacy UNIX Compatibility
   Security Tools
[root@ip-192-0-2-1 ~]#
```

6.2. Use the **`yum group info`** command to find out more information about the *RPM Development Tools* component group, including a list of included packages.

```
[root@ip-192-0-2-1 ~]# yum group info "RPM Development Tools"
Last metadata expiration check: 0:56:48 ago on Tue 28 Jul 2020 07:00:56 AM UTC.

Group: RPM Development Tools
 Description: Tools used for building RPMs, such as rpmbuild.
 Mandatory Packages:
   redhat-rpm-config
   rpm-build
 Default Packages:
   rpmdevtools
 Optional Packages:
   rpmlint
[root@ip-192-0-2-1 ~]#
```

6.3. Use the **`yum group install`** command to install the *RPM Development Tools* component group.

```
[root@ip-192-0-2-1 ~]# yum group install "RPM Development Tools"
Last metadata expiration check: 0:57:17 ago on Tue 28 Jul 2020 07:00:56 AM UTC.
Dependencies resolved.
================================================================================
 Package           Arch   Version        Repository                       Size
================================================================================
Installing group/module packages:
 redhat-rpm-config noarch 122-1.el8      rhel-8-for-x86_64-appstream-rpms  83 k
 rpm-build         x86_64 4.14.2-37.el8  rhel-8-for-x86_64-appstream-rpms 171 k
 rpmdevtools       noarch 8.10-7.el8     rhel-8-for-x86_64-appstream-rpms  87 k
Installing dependencies:
 binutils          x86_64 2.30-73.el8    rhel-8-for-x86_64-baseos-rpms    5.7 M
 bzip2             x86_64 1.0.6-26.el8   rhel-8-for-x86_64-baseos-rpms     60 k
...output omitted...
Transaction Summary
================================================================================
Install  74 Packages

Total download size: 28 M
Installed size: 90 M
Is this ok [y/N]: y
Downloading Packages:
(1/75): perl-Scalar-List-Utils-1.49-2.el8.x86_6 200 kB/s |  68 kB     00:00
(2/75): perl-PathTools-3.74-1.el8.x86_64.rpm    249 kB/s |  90 kB     00:00
...output omitted...
  Installing       : perl-Carp-1.42-396.el8.noarch                       1/75
  Installing       : perl-Exporter-5.72-396.el8.noarch                   2/75
  Installing       : perl-libs-4:5.26.3-416.el8.x86_64                   3/75
...output omitted...
  Verifying        : perl-Scalar-List-Utils-3:1.49-2.el8.x86_64          1/75
```

```
  Verifying        : perl-PathTools-3.74-1.el8.x86_64                    2/75
  Verifying        : perl-Data-Dumper-2.167-399.el8.x86_64               3/75
...output omitted...
Installed:
  binutils-2.30-73.el8.x86_64
  bzip2-1.0.6-26.el8.x86_64
  dwz-0.12-9.el8.x86_64
...output omitted...
Complete!
[root@ip-192-0-2-1 ~]#
```

7. Explore the **`history`** options of **`yum`**.

7.1. Use the **`yum history`** command to display recent **`yum`** history.

```
[root@ip-192-0-2-1 ~]# yum history
ID     | Command line             | Date and time    | Action(s)      | Altered
-------------------------------------------------------------------------------
     6 | group install RPM Develo | 2020-07-28 07:59 | Install        |   75
     5 | install gnuplot          | 2020-07-28 07:55 | Install        |   59
     4 | -y install vim-enhanced  | 2020-07-28 07:06 | Install        |    4
...output omitted...
[root@ip-192-0-2-1 ~]#
```

7.2. Use the **`yum history info`** command to confirm that the last transaction is the group installation.

```
[root@ip-192-0-2-1 ~]# yum history info 6
Transaction ID : 6
Begin time     : Tue 28 Jul 2020 07:59:12 AM UTC
Begin rpmdb    : 467:81427725b5ac0fd4c0adc6f0b8f799f6fe1315b8
End time       : Tue 28 Jul 2020 07:59:20 AM UTC (8 seconds)
End rpmdb      : 541:da6e11d22bd61700281e7a237e0d18e412b9d9f9
User           : Cloud User <ec2-user>
Return-Code    : Success
Releasever     : 8
Command Line   : group install RPM Development Tools
Packages Altered:
    Install rust-srpm-macros-5-2.el8.noarch         @rhel-8-appstream-rhui-rpms
    Install perl-URI-1.73-3.el8.noarch              @rhel-8-appstream-rhui-rpms
    Install perl-Net-SSLeay-1.88-1.el8.x86_64       @rhel-8-appstream-rhui-rpms
...output omitted...
[root@ip-192-0-2-1 ~]#
```

Important

Note that the transaction ID number (**6** above) may be different for you on your system. Use the most recent ID number that appears in the output of the **`yum history`** command.

7.3. Use the **`yum history undo`** command to remove the set of packages that were installed when the **`gnuplot`** package was installed.

```
[root@ip-192-0-2-1 ~]# yum history undo 5
Last metadata expiration check: 1:01:57 ago on Tue 28 Jul 2020 07:00:56 AM UTC.
Undoing transaction 5, from Tue 28 Jul 2020 07:55:18 AM UTC
    Install libXxf86vm-1.1.4-9.el8.x86_64           @rhel-8-appstream-rhui-rpms
    Install libSM-1.2.3-1.el8.x86_64                @rhel-8-appstream-rhui-rpms
...output omitted...

Transaction Summary
================================================================================
Remove  59 Packages

Freed space: 88 M
Is this ok [y/N]: y
...output omitted...
Complete!
[root@ip-192-0-2-1 ~]#
```

8. Use the **yum update** command to make sure all packages on the system are up to date.

```
[root@ip-192-0-2-1 ~]# yum update
Last metadata expiration check: 1:03:51 ago on Tue 28 Jul 2020 07:00:56 AM UTC.
Dependencies resolved.
================================================================================
 Package              Arch   Version           Repository                  Size
================================================================================
Installing:
 kernel               x86_64 4.18.0-193.13.2.el8_2
                                                rhel-8-baseos-rhui-rpms    2.8 M
 kernel-core          x86_64 4.18.0-193.13.2.el8_2
                                                rhel-8-baseos-rhui-rpms     28 M
 kernel-modules       x86_64 4.18.0-193.13.2.el8_2
                                                rhel-8-baseos-rhui-rpms     24 M
Upgrading:
 NetworkManager       x86_64 1:1.22.8-5.el8_2  rhel-8-baseos-rhui-rpms    2.3 M
 NetworkManager-libnm x86_64 1:1.22.8-5.el8_2  rhel-8-baseos-rhui-rpms    1.7 M
...output omitted...

Transaction Summary
================================================================================
Install   5 Packages
Upgrade  53 Packages

Total download size: 183 M
```

```
Is this ok [y/N]: y
...output omitted...
Complete!
[root@ip-192-0-2-1 ~]#
```

Important

There may be a delay of a few minutes during the cleanup and verification phase of the package update depending on how many packages are installed and what they are. Be patient if this occurs.

Note that the list of packages may be larger or smaller and different than portrayed in the preceding example, depending on the version of your instance's AMI and any additional software updates that become available after this course is published.

9. This concludes this exercise. Log out and stop your Amazon EC2 instance.

Important

This also concludes the final exercise of this course. Remember to terminate your Amazon EC2 instance to remove it entirely from the Amazon EC2 cloud and cloud storage once you are done using it.

Thank you for your participation in this course.