Chapter 1. Get Started with Red Hat Enterprise Linux

Abstract

Goal	Define open source, Linux, Linux distributions, and Red Hat Enterprise Linux.
Objectives	Explain the purpose of open source, Linux, Linux distributions, and Red Hat Enterprise Linux.
Sections	What Is Linux? (and Quiz)

What Is Linux?

Objectives

Define and explain the purpose of Linux, open source, Linux distributions, and Red Hat Enterprise Linux.

Why Should You Learn about Linux?

Linux is a critical technology for IT professionals to understand.

Linux is in widespread use, worldwide. Internet users interact with Linux applications and web server systems daily, by browsing the World Wide Web and using e-commerce sites to buy and sell products.

Linux is in use for much more than the internet. Linux manages point-of-sale systems and the world's stock markets, powers smart TVs and in-flight entertainment systems, and runs most of the top 500 supercomputers in the world. Linux provides the core technologies that power the cloud revolution and the tools to build the latest generations of container-based microservices applications, software-based storage technologies, and big data solutions.

In the modern data center, Linux and Microsoft Windows are the predominant operating systems. Linux use continues to expand in enterprise, cloud, and device spaces. Due to its widespread adoption, you have many reasons to learn Linux:

- A Windows user needs to interoperate with Linux systems and applications.
- In application development, Linux commonly hosts the application and its runtime.
- In cloud computing, both private and public cloud instances use Linux as the operating system.
- Mobile applications and Internet of Things (IoT) devices commonly run on Linux.
- When looking for new IT career opportunities, Linux skills are in high demand.

What Makes Linux Great?

If someone asks you "What makes Linux great?", then you have many answers to pick from:

- Linux is *open source* software.

  Being open source means that you can see all of how a program or system works. You can also experiment with changes and share them freely for others to use. The open source model means that improvements are easier to make, enabling faster innovation.

- Linux provides a *command-line interface* (CLI) for easy access and powerful scripting.

  Linux is built around a basic design philosophy that users can perform all administration tasks from the CLI. It enables easier automation, deployment, and provisioning, and simplifies both local and remote system administration. Unlike many other operating systems, these capabilities were in the architecture from the start, and result in ease of use and stability.

- Linux is a *modular* operating system that is designed to easily replace or remove components.

  System components can be upgraded and updated when needed. A Linux system can be a general-purpose development workstation or a purposefully minimized 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. Code might be in interpretive form, such as a script, or compiled into a binary executable that the computer runs directly. Source code becomes copyrighted when created, and the copyright holder controls the terms under which the software can be copied, adapted, and distributed. Users can use the software according to its software license.

Some software uses "proprietary" or "closed source" source code that only the originating person, team, or organization can see, or change, or distribute. Proprietary licenses typically restrict the user to running the program, and provide limited or no access to the source.

Open source software is different. When a copyright holder provides software under an open source license, they grant the user the right to run the program and to view, modify, compile, and redistribute the source to others, royalty-free. Open source licensing promotes collaboration, sharing, transparency, and rapid innovation, because it encourages more people to modify and improve the software and to share enhancements more widely.

Open source software can still be provided for commercial use. Open source is a critical part of many organizations' commercial operations. Some open source licenses allow code to be reused in proprietary products. Anyone can sell open source code, but open source licensing generally allows the customer to redistribute the source code. Open source vendors such as Red Hat provide commercial support for deploying, managing, and building solutions that are based on open source products.

Open source has many benefits for the user:

- *Control*: See what the code does and improve it.
- *Training*: Learn from real-world code and develop more applications that are useful.
- *Security*: Inspect sensitive code, and fix it even without the original developers' help.
- *Stability*: Rely on code that can survive the loss of the original developer.

Types of Open Source Licenses

The developers of open source software can license their software in different ways. The software license terms control how the source can be combined with other code or reused. To be open source, licenses must allow users to freely use, view, change, compile, and distribute the code.

Two general classes of open source license are particularly important:

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

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

Permissive licenses maximize the reusability of source code. You can use the source for any purpose if the copyright and license statements are preserved, including reusing code under more restrictive or proprietary licenses. Although permissive licensing makes it easy to reuse code, it risks encouraging proprietary-only enhancements. Examples of permissive licenses include the MIT/X11 license, the Simplified BSD license, and the Apache Software License 2.0.

Who Develops Open Source Software?

Open source development today is overwhelmingly professional. Open source is no longer solely developed by a body of volunteers. Today, most open source developers work for organizations that pay them to participate with open source projects to construct and contribute the enhancements that the organization and their customers need.

Volunteers and the academic community still play a significant role and can make vital contributions, especially in emerging technology. The combination of formal and informal development provides a highly dynamic and productive environment.

What Is a Linux Distribution?

A *Linux distribution* is an installable operating system that is constructed from a Linux kernel and that supports user programs and libraries. A complete *Linux* system is developed by multiple independent development

communities that work cooperatively on individual components. A distribution provides an easy method to install and manage a working Linux system.

In 1991, graduate student Linus Torvalds developed a UNIX-like kernel that he named Linux, and licensed it as open source software under the GPL. The kernel is the core of the operating system and manages hardware, memory, and the scheduling of running programs. The Linux kernel is supplemented with other open source software, including utilities and programs from the GNU Project, a graphical interface from MIT's *X Window System*. The Linux kernel also includes other open source components, such as the Sendmail mail server and the Apache HTTP web server, to become a complete open source UNIX-like operating system.

A major challenge for Linux users is to assemble all these software pieces from many sources. Early Linux developers provided a distribution of prebuilt and tested tools that users could download and install to quickly implement Linux systems.

Many Linux distributions exist, each with differing goals and support criteria. Generally, distributions have some common characteristics:

- Distributions consist of a Linux kernel and support user-space programs.
- Distributions can be small and single-purpose, or can include thousands of open source programs.
- Distributions provide a means to install and update the software and its components.
- The distribution provider supports the software, and ideally, participates in the development community.

Who Is Red Hat?

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

The role of Red Hat is to help customers to connect with the open source community and their partners to effectively use open source software solutions. Red Hat actively participates in and supports the open source community. Many years of experience have convinced the company of the importance of open source to the future of the IT industry.

Red Hat is best known for its participation in the Linux community and the Red Hat Enterprise Linux distribution. Red Hat is also active in other open source communities, including middleware projects that are centered on the JBoss developer community. Red Hat also provides virtualization solutions, cloud technologies such as OpenStack and OpenShift, and the Ceph and Gluster software-based storage projects, plus others.

Red Hat Enterprise Linux Ecosystem

Red Hat Enterprise Linux (RHEL) is Red Hat's commercial production-grade Linux distribution. Red Hat develops and integrates open source software into RHEL through a multistage process.

- Red Hat *participates* in supporting individual open source projects. It contributes code, developer time, resources, and support, and often collaborates with developers from other Linux distributions, to improve the general quality of software for everyone.
- Red Hat sponsors and *integrates* open source projects into the community-driven Fedora distribution. Fedora provides a free working environment to serve as a development lab and proving ground for features to be incorporated into CentOS Stream and RHEL products.
- Red Hat *stabilizes* the CentOS Stream software to be ready for long-term support and standardization, and integrates it into RHEL, the production-ready distribution.

Figure 1.1: The Red Hat Enterprise Linux ecosystem

Fedora

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

Fedora prioritizes innovation and excellence above long-term stability. Major updates occur every six months, and bring significant changes. Fedora supports releases for about a year, which means the latest two updates, making it less suited for supportable production use. Fedora remains the source of innovation for the entire Enterprise Linux ecosystem. In general, packages start out in Fedora and are included into CentOS Stream only when they are considered mature in stability, security, performance, and customer demand.

Extra Packages for Enterprise Linux

A Fedora project Special Interest Group (SIG) builds and maintains a community-supported package repository called Extra Packages for Enterprise Linux (EPEL). EPEL versions align with major RHEL releases, and enable RHEL customers to run workloads with software dependencies that are not supported in RHEL. EPEL packages are not included in Red Hat support, but are equivalent to Fedora's level of quality.

Typically, EPEL packages are built against RHEL releases. EPEL Next is an additional repository for package maintainers to build against CentOS Stream. This repository is useful when CentOS Stream contains an upcoming RHEL library rebase, or if an EPEL package has a minimum version build requirement that is already in CentOS Stream but not yet in RHEL.

CentOS Stream

CentOS Stream is the upstream project for RHEL. Development of the next RHEL version is transparent and open for community contributions that can directly influence the next release. Patches that are submitted to CentOS Stream are integrated faster to RHEL, to allow significant changes during the current RHEL version lifecycle. CentOS Stream is a continuous integration and delivery distribution, with tested and stable nightly builds.

The CentOS project welcomes contributors worldwide, to give RHEL derivatives the opportunity to contribute to CentOS Stream for their own benefit. The CentOS project also aims to promote sustainable open source software that responds faster to security exploits, emerging technologies, and changing customer requirements.

Note

Before 2019, CentOS Linux was a freely available, unsupported distribution, community-built from Red Hat's source code after each major RHEL release. Although the CentOS community enjoyed having a freely available RHEL clone, this model had disadvantages. Commonly, developer contributions to CentOS Linux were not backported to Fedora or RHEL without considerable duplicate effort. Also, significant delays occurred between a RHEL release and its corresponding CentOS distribution build, with a similar delay for critical RHEL security, driver, and tuning fixes. Red Hat switched to the CentOS Stream model to address these issues.

A benefit of CentOS Stream is that, as the source for RHEL development, it is available in all the same architectures as RHEL, including Intel/AMD x86_64, ARM64, IBM Power, and IBM Z.

Many innovative technology organizations have proven that CentOS Stream is a viable replacement for the original downstream CentOS Linux. CentOS Stream can be freely downloaded and installed for many use cases, including development and light production. For community users with use cases that are not suitable for a continuously delivered distribution with asynchronous patch releases, Red Hat provides free individual RHEL developer subscriptions for small-scale use, such as demos, prototyping, quality assurance, and limited production.

Red Hat Enterprise Linux

Red Hat Enterprise Linux (RHEL) is Red Hat's production-ready, commercially supported Linux distribution. In the computing industry, RHEL is acknowledged as the leading platform for open source computing. RHEL is extensively tested and has a worldwide ecosystem of support partners for hardware and software certifications, consulting services, training, and multi-year support and maintenance guarantees.

Red Hat builds RHEL major releases directly from the CentOS Stream continuous development project, which is sourced from Fedora. In contrast to the previous

RHEL development model, the releases were constructed internally with less transparency, and the source was provided only for building as CentOS Linux after the RHEL release. Now the new CentOS Stream development model is open and available to all, for feedback and contribution, and the code is prepared to be the next major RHEL release.

RHEL uses a subscription-based support model, and does not charge license fees for open-source software. Red Hat support subscriptions provide product support, maintenance, updates, security patches, and access to the Customer Portal Knowledgebase, utilities, and downloadable releases of Red Hat products.

The following table lists some key differences between Fedora, CentOS Stream, and RHEL.

	Fedora	**CentOS Stream**	**RHEL**
Expected lifecycle	12-18 months	5 years	10 years
Software vendor certified	No	Usually not	Yes
Documentation provided by	Community	Community	Red Hat
Expert support available	No	No	Yes
Product security team	No	No	Yes
Security certifications	No	No	Yes
No-cost options	Yes	Yes	Yes
Management tools	No	No	Yes

RHEL for Edge

RHEL for Edge is an image-based variant of RHEL, with a different deployment mechanism. RHEL provides the ability to create purpose-built operating system images through a tool called Image Builder. With this mechanism, IT teams can build, deploy, and maintain these RHEL images in less time over the life of the system. Image-based deployments are optimized for various edge architectures, but are customizable for specific edge deployments.

The Edge features in RHEL include secure management and scaling capabilities, including zero-touch provisioning, system health visibility, and quick security remediations from within a single interface.

Red Hat CoreOS

RHEL CoreOS (RHCOS) is not a stand-alone operating system, but it is built from RHEL components, and is then released, upgraded, and managed as part of the Red Hat OpenShift Container Platform (RHOCP) for cloud-native applications. RHCOS is fundamentally an image-based RHEL container host, which uses the Container Runtime Interface (CRI-O)-compliant container engine that is integrated in RHOCP. To learn more about Red Hat CoreOS, begin by becoming familiar with OpenShift and containers.

Red Hat Universal Base Image

A Red Hat Universal Base Image (UBI) is essentially a freely redistributable derivative of RHEL. UBI is designed to be a foundation for cloud-native and web application use cases that are developed in containers. All UBI content is a subset of RHEL, with packages sourced from secure RHEL channels, and UBI is supported similar to RHEL when run on a Red Hat supported platforms such as OpenShift and RHEL hosts.

With UBI, developers can focus their efforts on their application in the container image. UBI is a set of base images, and a set of application images (such as python, ruby, node.js, httpd, or nginx). UBI also consists of a set of RPM repositories, from which you can update any UBI base image to include the package dependencies that your application requires.

Red Hat Enterprise Linux Continuous Development

In the Fedora upstream community, Fedora Rawhide is the continuous development environment for a regular cadence of public Fedora releases. The community tests and prepares new Linux kernel versions, device drivers, utilities, and applications for the next Fedora distribution. Major RHEL release development begins with selection of the latest Fedora release as the base for the current CentOS Stream continuous development distribution.

Before a package is formally introduced to CentOS Stream, it undergoes rigorous testing to meet the standards for packages to be included in RHEL. Updates that are posted to CentOS Stream are the same as to those updates that are posted to the unreleased minor version of RHEL in development.

Figure 1.2: Red Hat Enterprise Linux continuous development

As shown in the figure, Fedora 34 is the original code base for RHEL 9 and for CentOS Stream 9. As packages are updated, they are then pushed into CentOS Stream and the nightly build of RHEL. The solid lines indicate distributions or builds that are available for public use.

Similar to the relationship between Fedora Rawhide and Fedora, CentOS Stream is the continuous development environment for preparing the next minor-version RHEL release. Red Hat performs extensive hardware, integration, dependency, and performance testing before releasing the next public RHEL distribution.

Obtaining Red Hat Enterprise Linux

Red Hat Enterprise Linux is typically obtained with a paid support subscription, and Red Hat provides multiple ways to obtain RHEL and other RHEL ecosystem products, many without cost.

- *Fedora Linux* and derivatives are freely available from the Fedora project at https://getfedora.org/, including an emerging version of Fedora CoreOS.
- *EPEL* and *EPEL Next* packages are freely available from the EPEL project repositories. Learn how to use EPEL at https://docs.fedoraproject.org/en-US/epel/.
- *CentOS Stream* is freely available at https://www.centos.org/centos-stream/.

RHEL Evaluation Download

An evaluation copy of RHEL is available at https://access.redhat.com/products/red-hat-enterprise-linux/evaluation. You must have a (free) Customer Portal account for https://access.redhat.com to access and download evaluation products. Product evaluations entitle you to receive updates and support for a limited period. Support ends when the evaluation period ends, but the evaluation software continues to operate. Additional information for many product evaluations is found on the Customer Portal evaluation pages.

Red Hat Developer Subscription

Red Hat provides a freely available subscription for many products through the Red Hat Developer Program at https://developer.redhat.com. With a Developer subscription, developers can quickly create, prototype, test, and demonstrate their applications on the same Red Hat software as on production systems. Create a

personal account at https://access.redhat.com, and then register for the Developer program. You can use an existing personal account, but do not use an account that is already associated with any organization's support subscription. The Developer subscription is self-supported, but provides ongoing product updates. Red Hat recommends that individuals who want to gain experience with RHEL and developer products should join the Developer Program.

Public Cloud Platforms

The major hyperscale public cloud providers, such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure, offer official images for deploying Red Hat Enterprise Linux instances, with subscription management from the Red Hat Cloud Access service. Fully entitled subscriptions for RHEL and Red Hat products are available through the cloud provider, and are portable in hybrid and multi-vendor clouds.

Containers

You can use Red Hat Universal Base Images and associated content for development and deployment without a Red Hat subscription. For operational support and access to non-UBI tools, containers that are built on UBI must be deployed on a Red Hat-supported platform such as OpenShift or Red Hat Enterprise Linux. Access to non-UBI content requires a Red Hat subscription.

References

Get Started with Red Hat Enterprise Linux

No-cost Red Hat Enterprise Linux Individual Developer Subscription: FAQs

The Open Source Way

Fedora

Red Hat Universal Base Images

Red Hat Cloud Access

Next

Quiz: Get Started with Red Hat Enterprise Linux

Choose the correct answers to the following questions:

1.

2.

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

A	Code can survive the loss of the original developer or distributor.
B	Sensitive portions of code are protected and available only to the original developer.
C	You can learn from real-world code and develop more effective applications.
D	Code remains open provided that it is in a public repository, but the license might change when included with closed source software.

3. CheckResetShow Solution

4.

5.

2. Which two statements are ways in

which Red Hat develops products for the future and interacts with the community? (Choose two.)

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

B. Develop specific integration tools that are available only in Red Hat distributions.

C. Participate in upstream projects.

D. Repackage and relicense community products.

6. CheckResetShow Solution

7.

8.

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

A. Linux is developed entirely by volunteers, which makes it a low-cost operating system.

B. Linux is modular and can be configured for 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, so it is easier to develop custom software.
D	Linux includes a powerful and scriptable command-line interface, which enables easier automation and provisioning.

9. CheckResetShow Solution

Previous Next

Summary

- Open source software has source code that anyone can freely use, study, modify, and share.
- A Linux distribution is an installable operating system that is constructed from a Linux kernel and that supports user programs and libraries.
- Red Hat participates in supporting and contributing code to open source projects; sponsors and integrates project software into community-driven distributions; and stabilizes the software to offer it as supported enterprise-ready products.
- Red Hat Enterprise Linux is the open source, enterprise-ready, commercially supported Linux distribution that Red Hat provides.
- A free Red Hat Developer Subscription is a useful method for obtaining learning resources and information, including developer subscriptions to Red Hat Enterprise Linux and other Red Hat products.

Previous Next

Chapter 2. Access the Command Line

Lab: Access the Command Line
Summary
Abstract

Goal	Log in to a Linux system and run simple commands from the shell.
Objectives	• Log in to a Linux system and run simple commands from the shell. • Log in to the Linux system with the GNOME desktop environment to run commands from a shell prompt in a terminal program. • Save time when running commands from a shell prompt with Bash shortcuts.
Sections	• Access the Command Line (and Quiz) • Access the Command Line with the Desktop (and Guided Exercise) • Execute Commands with the Bash Shell (and Quiz)
Lab	Access the Command Line

Access the Command Line

Objectives

Log in to a Linux system and run simple commands with the shell.

Introduction to the Bash Shell

A *command line* is a text-based interface that is used to input instructions to a computer system. The Linux command line is provided by a program called the *shell*. Many shell program variants have been developed over the years. Every user can use a different shell, but the Red Hat recommends using the default shell for system administration.

The default user shell in Red Hat Enterprise Linux (RHEL) is the `GNU Bourne-Again Shell` (`bash`). The `bash` shell is an improved version of the original `Bourne Shell` (`sh`) on UNIX systems.

The shell displays a string when it is waiting for user input, called the *shell prompt*. When a regular user starts a shell, the prompt includes an ending dollar (`$`) character:

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

A hash (`#`) character replaces the dollar (`$`) character when the shell is running as the superuser, `root`. This character indicates that it is a superuser shell, which helps to avoid mistakes that can affect the whole system.

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

Using `bash` to execute commands can be powerful. The `bash` shell provides a scripting language that can support task automation. The shell has capabilities that can enable or simplify operations that are hard to accomplish at scale with graphical tools.

Note

The `bash` shell is conceptually similar to the Microsoft Windows `cmd.exe` command-line interpreter. However, `bash` has a sophisticated scripting language, and is more similar to Windows PowerShell.

On macOS, the `bash` shell was the default shell before macOS 10.15 Catalina. Starting from macOS 10.15 Catalina, Apple changed the default shell to the `zsh` shell, an alternative shell that is also available in RHEL.

Shell Basics

Commands that are 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 might be followed by one or more options, which adjust the behavior of the command or what it does. Options normally start with one or two dashes (`-a` or `--all`, for example) to distinguish them from arguments. Commands might also be followed by one or more arguments, which often indicate a target that the command should operate on.

For example, in the `usermod -L user01` string, `usermod` is the command, `-L` is the option, and `user01` is the argument. This command locks the password of the `user01` user account.

Log in to a Local System

A *terminal* is a text-based interface to enter commands into and print output from a computer system. To run the shell, you must log in to the computer on a terminal.

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

The computer might provide a graphical login prompt on one of the virtual consoles. You can use the graphical login prompt 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, because users do not log in to servers as a desktop workspace. A server's workload can more effectively use the significant resources that a graphical environment uses.

In Red Hat Enterprise Linux 9, if the graphical environment is available, then the login screen runs on the first virtual console, which is called `tty1`. Five additional text login prompts are available on virtual consoles two (`tty2`) through six (`tty6`).

The graphical environment starts on the first virtual console that a login session is not currently using. Normally, your graphical session replaces the login prompt on the second virtual console (`tty2`). However, if an active text login session (not just a login prompt) is using that console, then 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 switch to another user in the graphical environment without logging out, then another graphical environment is started for that user on the next available virtual console.

When you log out of a graphical environment, it exits the virtual console, and the physical console automatically switches 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 8, the behavior is the same as in Red Hat Enterprise Linux 9.

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

The serial console is normally used to access the server if the server network card becomes misconfigured and logging to the server over the conventional network connection becomes impossible. Most of the time, however, headless servers are accessed by other means over the network, for example by using Virtual Network Computing (VNC) for running a graphical interface on the target machine.

Log in to a Remote System

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 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 `host` machine 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, for tighter security do not allow users to use a password to log in with `ssh`. 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 with 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. If their matching public key is installed in that account on that remote server, then it logs in the user without asking for a password.

In the next example, a user with a shell prompt on the `host` machine logs in to `remotehost` as `remoteuser` with `ssh`, by using the public key authentication method. The `ssh` command `-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 the connection to work, only the user who owns the file can have access to read the private key file. In the preceding example, where the private key is in the `mylab.pem` file, you can use the `chmod 600 mylab.pem` command to ensure that only the owner can read the file. How to set file permissions is discussed in more detail in a later chapter.

Users might also have configured private keys 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 about this topic.

Note

When you first log in to a new machine, you are 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 that you connect to a remote host with `ssh`, the remote host sends its *host key* to authenticate itself and to help to set up encrypted communication. The `ssh` command compares the host key against a list of saved host keys to ensure that it is not changed. If the host key changed, then it might indicate that someone is trying to pretend to be that host to hijack the connection, which is also known as an interceptor attack. In SSH, host keys protect against interceptor attacks; these host keys are unique for each server; and they need to be changed periodically and whenever a compromise is suspected.

You get this warning when your local machine does not have a saved host key for the remote host. If you enter `yes`, then the host key that the remote host sent is accepted and saved for future reference. The login process continues, and you should not see this message again when connecting to this host. If you enter `no`, then the host key is rejected and the connection is closed.

If the local machine does have a saved host key and it does not match the one that the remote host sent, then the connection is closed automatically with a warning.

Log Out from a Remote System

When you are finished with 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 example shows a user who logs out of an SSH session:

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

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

References

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

For more information about OpenSSH and public key authentication, refer to the *Using Secure Communications between Two Systems with OpenSSH* chapter in the *Red Hat Enterprise Linux 9 Securing Networks* guide at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html-single/securing_networks/index

Instructions about how to read `man` pages and other online help documentation are included in the upcoming chapter.

Next

Quiz: Access the Command Line

Choose the correct answer to the following questions:

1.

2.

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

A	Command
B	Console
C	Shell
D	Terminal

3. CheckResetShow Solution

4.

5.

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

A	Argument
B	Command
C	Option
D	Prompt

6. CheckResetShow Solution

7.

8.

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

A Argument

B Command

C Option

D Prompt

9. CheckResetShow Solution

10.

11.

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

12. CheckResetShow Solution

13.

14.

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

15. CheckResetShow Solution

16.

17.

6. Which term describes the hardware display and keyboard to

interact with a system?

A Physical Console

B Virtual Console

C Shell

D Terminal

18. CheckResetShow Solution

19.

20.

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

21. CheckResetShow Solution

22.

23.

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

24. CheckResetShow Solution

Previous Next

Access the Command Line with the Desktop

Objectives

Log in to the Linux system with the GNOME desktop environment to run commands from a shell prompt in a terminal program.

Introduction to the GNOME Desktop Environment

The *desktop environment* is the graphical user interface on a Linux system. `GNOME 40` is the default desktop environment in Red Hat Enterprise Linux 9. It provides an integrated desktop for users and a unified development platform on top of a graphical framework that either `Wayland` (by default) or the legacy `X Window System` provides.

`GNOME Shell` provides the core user interface functions for the GNOME desktop environment. The GNOME Shell application is highly customizable. Red Hat Enterprise Linux 9 defaults the GNOME Shell appearance to the "Standard" theme, which is used in this section. You can default to an alternative "Classic" theme, which is closer to the appearance of earlier versions of GNOME, and which is used on previous RHEL versions. You can select either theme persistently at login by clicking the gear icon next to the **Sign In** button. The gear icon is available after selecting your account but before entering your password.

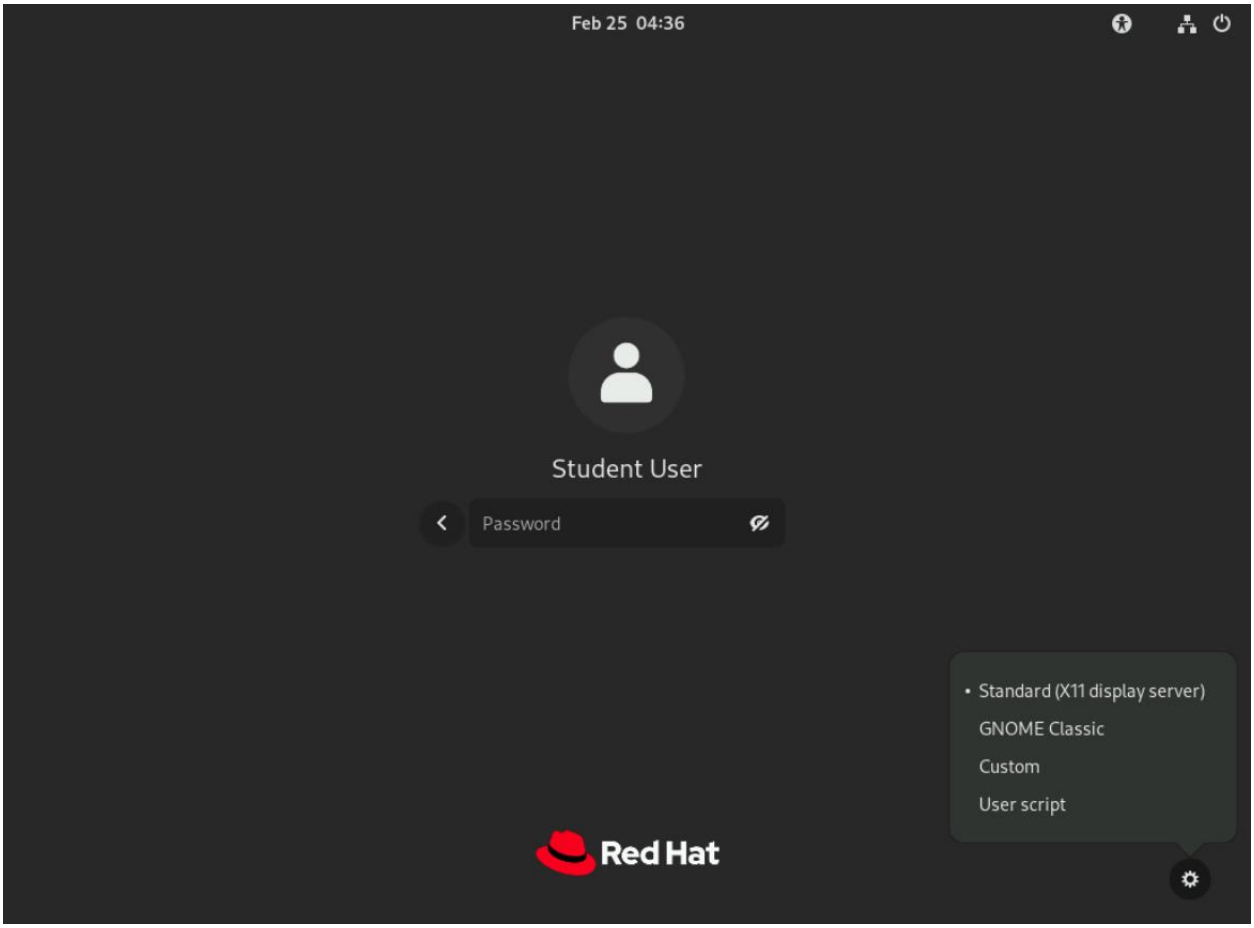


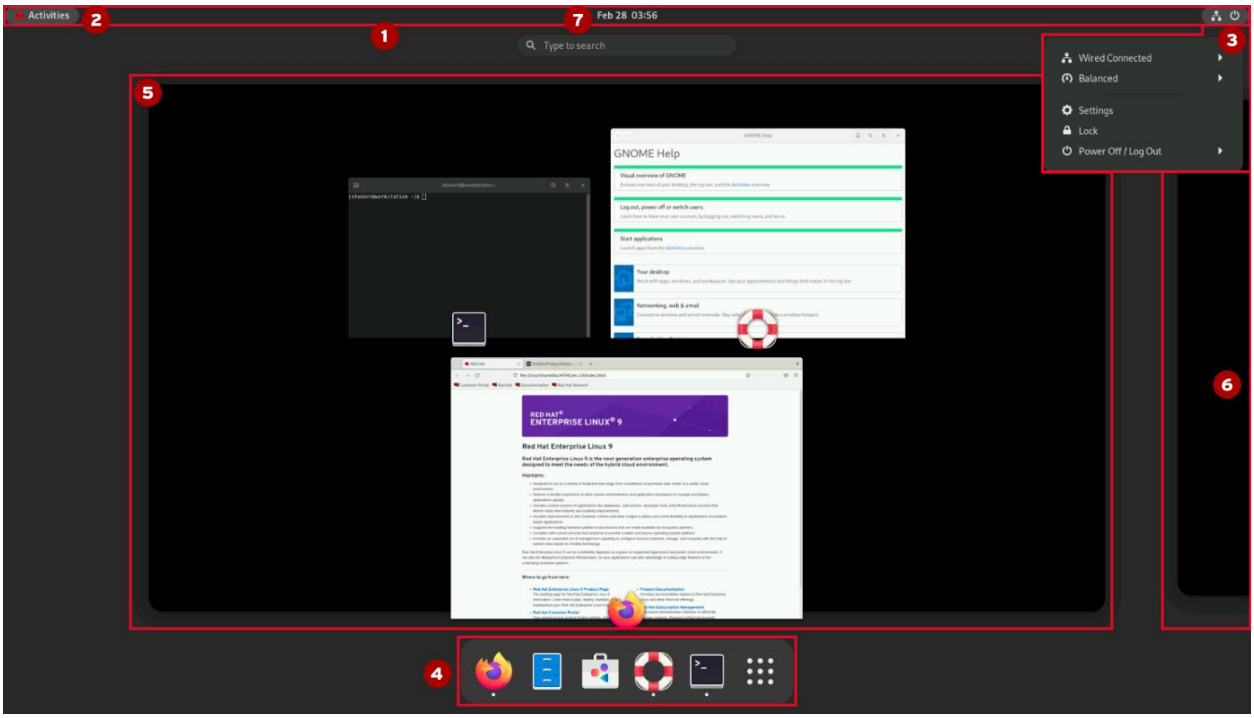
Figure 2.1: The RHEL 9 login screen

The first time that you log in as a new user, you can take an optional initial "Take Tour" program to learn about the new RHEL 9 features. After you either complete it or skip it, the main GNOME screen appears.

To review the documentation in `GNOME Help`, click the **Activities** button on the left side of the top bar. In the dash at the bottom of the screen, click the ring icon to launch it or hover your mouse over the `Help` icon.

Parts of the GNOME Shell

The elements of the GNOME Shell include the following parts, as shown in this screen capture of the GNOME Shell in Activities overview mode:



Top bar: The bar that runs along the top of the screen. It is displayed in the Activities overview and in workspaces. The top bar provides the **Activities** button and controls for volume, networking, calendar access, and switching between keyboard input methods (if more than one method is configured).

Activities overview: This mode helps to organize windows and to start applications. Enter the Activities overview by clicking the **Activities** button at the upper-left corner of the top bar, or by pressing the **Super** key. Find the **Super** key (sometimes called the **Windows** key or **Command** key) near the lower-left corner of most common keyboards. The three main areas are the **dash** at the bottom of the screen, the **windows overview** in the center, and the **workspace selector** on the right side.

System menu: The menu in the upper-right corner on the top bar provides control to adjust the brightness of the screen, and to switch on or off the network connections. Under the

submenu for the user's name are options to adjust account settings and to log out of the system. The system menu also offers buttons to open the **Settings** window, lock the screen, or shut down the system.

Dash: This configurable list of icons shows your favorite applications, running applications, and a **Show Applications** button to select arbitrary applications. Start applications by clicking an icon or by using the **Show Applications** button to find less commonly used applications. The dash is also called the **dock**.

Windows overview: The area in the center of the Activities overview that displays thumbnails of active windows in the current workspace, for bringing windows to the foreground on a cluttered workspace, or moving them to another workspace.

Workspace selector: An area to the right, which displays thumbnails of active workspaces, and allows selecting workspaces and moving windows from one workspace to another.

Message tray: With the message tray, you can review notifications from applications or system components. If a notification occurs, the notification typically first appears briefly as a single line at the top of the screen, and a persistent indicator appears in the top bar next to the clock to inform you of recently received notifications. Open the message tray to review these notifications by clicking the clock on the top bar or by pressing **Super**+**M**. Close the message tray by clicking the clock on the top bar, or by pressing **Esc** or **Super**+**M** again. The message tray also shows the calendar and information about the events in the calendar.

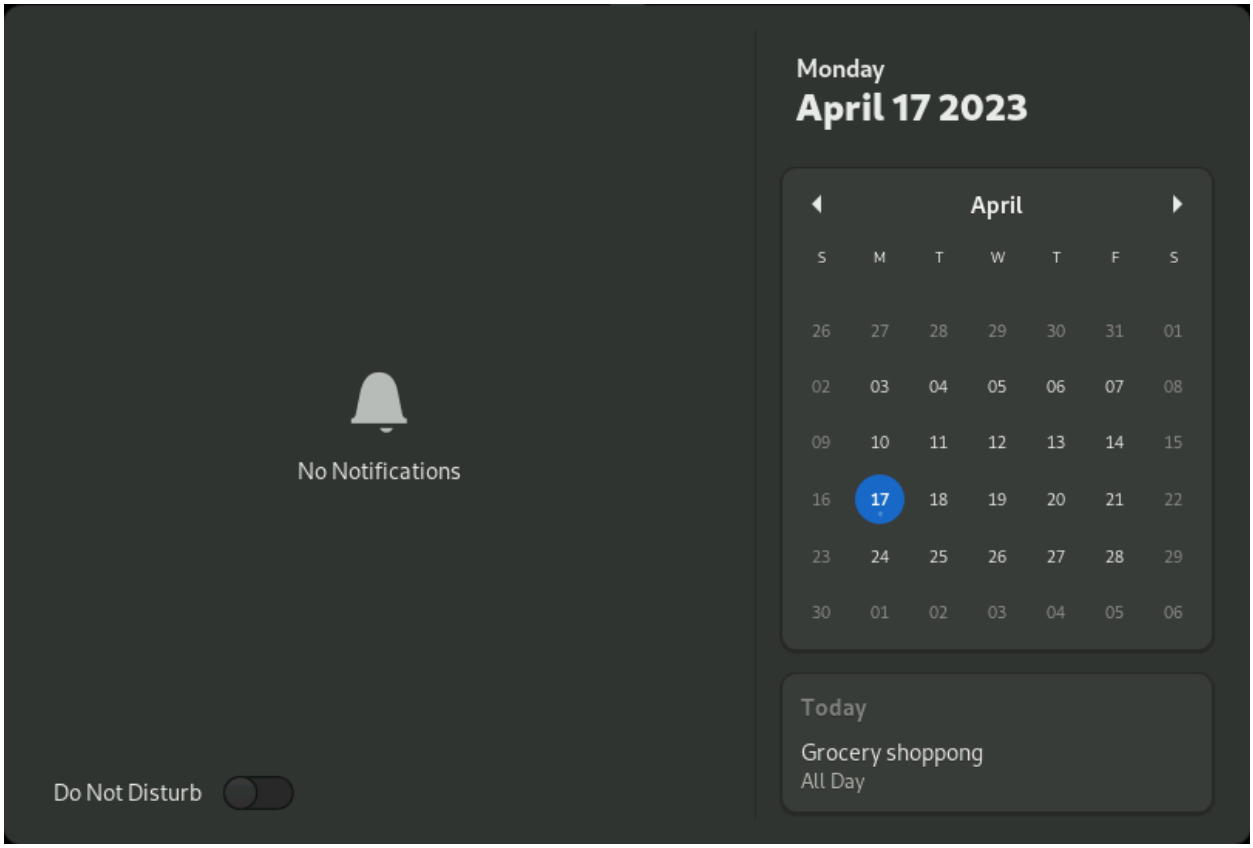


Figure 2.2: Closeup of an open message tray

View and edit the GNOME keyboard shortcuts that your account uses. Open the system menu on the right side of the top bar. Click the **Settings** button on the bottom of the menu on the left. In the application window that opens, select **Keyboard** from the left pane. The right pane displays your current shortcut settings under the **Keyboard Shortcuts** → **Customize Shortcuts** section.

Note

Some keyboard shortcuts, such as function keys or the **Super** key, might be difficult to send to a virtual machine. Special keystrokes that those shortcuts use might be captured by your local operating system, or by the application that you are using to access the graphical desktop of your virtual machine.

Important

In the current virtual training and self-paced training environments provided by Red Hat, use of the **Super** key can be tricky, because your web browser might not pass it to the virtual machine in the classroom environment.

At the top of your browser window that displays the interface for your virtual machine, click the keyboard icon on the right side. An on-screen keyboard opens. Click the icon again to close the on-screen keyboard.

The on-screen keyboard treats **Super** as a modifier key that is often held down while pressing another key. If you click it once, then it turns yellow, to indicate that the key is being held down. So for example, to enter **Super**+**M** in the on-screen keyboard, you can click **Super** and then click **M**.

To press and release **Super** in the on-screen keyboard, then click it twice. The first click holds down the **Super** key, and the second click releases it.

The other keys that the on-screen keyboard treats as modifier keys (like **Super**) are **Shift**, **Ctrl**, **Alt**, and **Caps**. The **Esc** and **Menu** keys are treated like normal keys and *not* modifier keys.

Access Workspaces

Workspaces are separate desktop screens that have different application windows. You can use workspaces to organize the working environment by grouping open application windows by task. For example, you can group windows for a particular

system maintenance activity (such as setting up a new remote server) in one workspace, and you can group email and other communication applications in another workspace.

Choose between two methods to switch between workspaces. The first method is to press **Ctrl**+**Alt**+**LeftArrow** or **Ctrl**+**Alt**+**RightArrow** to switch between workspaces sequentially. The second is to switch to the **Activities** overview and click the chosen workspace.

An advantage of using the **Activities** overview is that you can click and drag windows between workspaces by using the **workspace selector** on the right side of the screen and the **windows overview** in the center of the screen.

Important

Like **Super**, in the current virtual training and self-paced training environments provided by Red Hat, your web browser does not usually pass **Ctrl**+**Alt** key combinations to the virtual machine in the classroom environment.

You can enter these key combinations to switch workspaces by using the on-screen keyboard. At least two workspaces must be in use. Open the on-screen keyboard and click **Ctrl**, **Alt**, and then either **LeftArrow** or **RightArrow**.

However, in those training environments, it is generally simpler to avoid the keyboard shortcuts and the on-screen keyboard. Switch workspaces by clicking the **Activities** button and then, in the workspace selector to the right of the Activities overview, clicking the workspace to switch to.

Start a Terminal

To get a shell prompt in GNOME, start a graphical terminal application such as `GNOME Terminal`. Use one of the following methods to start a terminal:

- From the **Activities** overview, select `Terminal` from the **dash**, either in Favorites or with the **Show Applications** button.
- Search for `terminal` in the search field at the top of the **windows overview**.
- Press the **Alt**+**F2** key combination to open the **Enter a Command** and enter `gnome-terminal`.

When you open a terminal window, a shell prompt is displayed for the user who started the graphical terminal program. The shell prompt and the terminal window's title bar indicate the current username, hostname, and working directory.

Lock the Screen and Log Out

Lock the screen, or log out entirely, from the system menu on the far right of the top bar.

To lock the screen, from the system menu in the upper-right corner, click the lock button at the bottom of the menu or press **Super**+**L** (which might be easier to remember as **Windows**+**L**). The screen also locks if the graphical session is idle for a few minutes.

A **lock screen curtain** appears that shows the system time and the name of the logged-in user. To unlock the screen, you can press **Enter**, **Space**, or click the left mouse button. Then, enter that user's password on the **lock screen**.

To log out and end the current graphical login session, select the system menu in the upper-right corner on the top bar and select **Power Off/Log Out** → **Log Out**. A window is displayed that offers the option to **Cancel** or confirm the **Log Out** action.

Power Off or Reboot the System

To shut down the system, from the system menu in the upper-right corner, select **Power Off/Log out** → **Power Off** or press **Ctrl**+**Alt**+**Del**. A window is displayed that offers the option to **Cancel** or confirm the **Power Off** action. If you do not make a choice, then the system automatically shuts down after 60 seconds.

To reboot the system, from the system menu in the upper-right corner, select **Power Off/Log out** → **Restart**. A window is displayed that offers the option to **Cancel** or confirm the **Restart** action. If you do not make a choice, then the system automatically restarts after 60 seconds.

References

GNOME Help

- `yelp`

GNOME Help: *Visual Overview of GNOME*

- `yelp help:gnome-help/shell-introduction`

GNOME 40 Web Page

Previous Next

Guided Exercise: Access the Command Line with the Desktop

In this exercise, you log in through the graphical display manager as a regular user to become familiar with the GNOME Standard desktop environment that GNOME 40 provides.

Outcomes

- Log in to a Linux system by using the GNOME 40 desktop environment.
- Run commands from a shell prompt in a terminal program.

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start cli-desktop
```

Instructions

1. Log in to `workstation` as `student` with `student` as the password.
    1. On `workstation`, at the GNOME login screen, click the `student` user account. Enter `student` when prompted for the password.
    2. Press **Enter**.
2. Change the password for `student` from `student` to `55TurnK3y`.

> Important

The `finish` script resets the password for the `student` user to `student`. You must execute the script at the end of the exercise.

1. Open a `Terminal` window and use the `passwd` command at the shell prompt.

   In the virtual learning environment with a visual keyboard, press the **Super** key twice to enter the **Activities** overview. Type `terminal` and then press **Enter** to start `Terminal`.

2. In the `Terminal` window that opens, type `passwd` at the shell prompt. Change the `student` password from `student` to `55TurnK3y`.

```
[student@workstation ~]$ passwd
Changing password for user student.
Current password: student
New password: 55TurnK3y
Retype new password: 55TurnK3y
```

```
passwd: all authentication tokens updated successfully.
```

3. Log out and log back in as `student` with `55TurnK3y` as the password to verify the changed password.
   1. Click the system menu in the upper-right corner.
   2. Select **Power Off/Log Out** → **Log Out**.
   3. Click **Log Out** in the confirmation dialog box that is displayed.
   4. At the GNOME login screen, click the `student` user account. Enter `55TurnK3y` when prompted for the password.
   5. Press **Enter**.
4. Lock the screen.
   1. From the system menu in the upper-right corner, press the **Lock** button.
5. Unlock the screen.
   1. Press **Enter** to unlock the screen.
   2. In the **Password** field, enter `55TurnK3y` as the password.
   3. Press **Enter**.
6. Determine how to shut down `workstation` from the graphical interface, but **Cancel** the operation without shutting down the system.
   1. From the system menu in the upper-right corner, select **Power Off/Log Out** → **Power Off**. A dialog box is displayed with the options to either **Cancel** or **Power Off** the machine.

2. Click **Cancel** in the dialog box that is displayed.

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish cli-desktop
```

This concludes the section.

Previous Next

Execute Commands with the Bash Shell

Objectives

Save time when running commands from a shell prompt with Bash shortcuts.

Basic Command Syntax

The `GNU Bourne-Again Shell` (`bash`) is a program that interprets commands that the user types. Each string that is typed into the shell can have up to three parts: the command, options (which usually begin with a hyphen `-` or double hyphen `--` characters), and arguments. Each word that is typed into the shell is separated from other words with spaces. Commands are the names of programs that are installed on the system. Each command has its 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 following shell prompt appears.

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

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 a special interpretation for `bash`. In this case, the output of both commands is displayed before the following shell prompt appears.

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

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

Write Simple Commands

The `date` command displays the current date and time. The superuser or a privileged user can also use the `date` command to set the system clock. Use the plus sign (**+**) as an argument to specify a format string for the date command.

```
[user@host ~]$ date
Sun Feb 27 08:32:42 PM EST 2022
[user@host ~]$ date +%R
20:33
[user@host ~]$ date +%x
02/27/2022
```

The `passwd` command with no options changes the current user's password. To change the password, first specify the original password for the account. By default, the `passwd` command is configured to require a strong password, to consist of lowercase letters, uppercase letters, numbers, and symbols, and not to be based on a dictionary word. A superuser or privileged user can use the `passwd` command to change another user's password.

```
[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 compiled header of a file for a 2-digit `magic number` and displays its type. Text files are recognized because they are not compiled.

```
[user@host ~]$ file /etc/passwd
/etc/passwd: ASCII text
[user@host ~]$ file /bin/passwd
/bin/passwd: setuid ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamica
lly linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=a467cb9c8fa7306d41
b96a820b0178f3a9c66055, for GNU/Linux 3.2.0, stripped
[user@host ~]$ file /home
/home: directory
```

View the Contents of Files

The `cat` command is often used in Linux. Use this command to create single or multiple files, view the contents of files, concatenate the contents from various files, and redirect contents of the file to a terminal or to files.

The following 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...
```

To display the contents of multiple files, add the file names to the `cat` command as arguments:

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

Some files are long and might need more space to be displayed than the terminal provides. 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 you can scroll at your leisure.

Use the `less` command to page forward and backward through longer files 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 the end of a file, respectively. By default, these commands display 10 lines of the file, but they both have a `-n` option to specify a different number of lines.

```
[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:980:978::/run/gnome-initial-setup/:/sbin/nologin
dnsmasq:x:979:977:Dnsmasq DHCP and DNS server:/var/lib/dnsmasq:/sbin/nologin
```

The `wc` command counts lines, words, and characters in a file. Use the `-l`, `-w`, or `-c` options to display only the given number of lines, words, or characters, respectively.

```
[user@host ~]$ wc /etc/passwd
41   98 2338 /etc/passwd
[user@host ~]$ wc -l /etc/passwd ; wc -l /etc/group
```

```
41 /etc/passwd
63 /etc/group
[user@host ~]$ wc -c /etc/group /etc/hosts
883 /etc/group
114 /etc/hosts
997 total
```

Understand Tab Completion

With tab completion, users can quickly complete commands or file names after typing enough at the prompt to make it unique. If the typed characters are not unique, then pressing the **Tab** key twice displays all commands that begin with the typed characters.

```
[user@host ~]$ pasTab+Tab
passwd       paste        pasuspender

[user@host ~]$ passTab
[user@host ~]$ passwd
Changing password for user user.
Current password:
```

Press **Tab** twice.

Press **Tab** once.

Tab completion helps to complete file names when typing them as arguments to commands. Press **Tab** to complete as much of the file name as possible.
Pressing **Tab** a second time causes the shell to list all files that the current pattern matches. Type additional characters until the name is unique, and then use tab completion to complete the command.

```
[user@host ~]$ ls /etc/pasTab

[user@host ~]$ ls /etc/passwdTab
passwd   passwd-
```

Press **Tab** once.

Press **Tab** once.

Use the `useradd` command to create users on the system. The `useradd` command has many options that might be hard to remember. By using tab completion, you can complete the option name with minimal typing.

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

Press **Tab** twice.

Write a Long Command on Multiple Lines

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

To write one command in more than one line, use a backslash character (\), which is referred to as the *escape character*. The backslash character ignores the meaning of the following character.

Previously, you learned that to complete a command entry, you press the **Enter** key, the newline character. By escaping the newline character, the shell moves to a new command line without executing the command. This way, the shell acknowledges the request by displaying a continuation prompt on an empty new line, which is known as the secondary prompt, and uses the greater-than character (>) by default. Commands can continue over many lines.

One issue with the secondary prompt's use of the greater-than character (>) is that new learners might mistakenly insert it as part of the typed command. Then, the

shell interprets a typed greater-than character as *output redirection*, which the user did not intend. Output redirection is discussed in an upcoming chapter. This course book does not show secondary prompts in screen outputs, to avoid confusion. A user still sees the secondary prompt in their shell window, but the course material intentionally displays only the characters to be typed, as demonstrated in the following 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
```

Display the Command History

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

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

```
[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 student student 6 Feb 27 19:24 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 help to navigate through previous commands in the shell's history. The **UpArrow** edits the previous command in the history list. The **DownArrow** edits the next command in the history list. The **LeftArrow** and **RightArrow** move the cursor left and right in the current command from the history list so that you can edit the command before running it.

Use either the **Esc**+**.** or **Alt**+**.** key combination simultaneously to insert the last word of the previous command at the cursor's current location. The repeated use of the key combination replaces that text with the last word of earlier commands in history. The **Alt**+**.** key combination is particularly convenient, because you can hold down **Alt** and press **.** repeatedly to quickly cycle earlier commands.

Edit the Command Line

When used interactively, `bash` has a command-line editing feature. Use the text editor commands to move around and modify the currently typed command. Using the arrow keys to move within the current command and to step through the command history was introduced earlier in this section. The following table shows further powerful editing commands.

Table 2.1. 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.

These command-line editing commands are the most helpful for new users. For other commands, refer to the `bash`(1) man page.

References

`bash`(1), `date`(1), `file`(1), `magic`(5), `cat`(1), `more`(1), `less`(1), `head`(1), `passwd`(1), `tail`(1), and `wc`(1) man pages

Previous Next

Quiz: Execute Commands with the Bash Shell

Choose the correct answers to the following questions:

1.

2.

1. Which Bash command displays the last five lines of the `/var/log/messages` file?

A `head -n 10 /var/log/messages`

B `tail 10 /var/log/messages`

C `tail -n 5 /var/log/messages`

D	`tail -l 10 /var/log/messages`
E	`less /var/log/messages`

3. CheckResetShow Solution

4.

5.

2. Which Bash shortcut or command separates commands on the same line?

A	Pressing **Tab**
B	`history`
C	**;**
D	`!`*`string`*
E	Pressing **Esc**+**.**

6. CheckResetShow Solution

7.

8.

3. Which Bash command is used to change a user's password?

A `password`

B `pass`

C `passwd`

D `usermod`

E `userpassword`

9. CheckResetShow Solution

10.

11.

4. Which Bash command is used to display the file type?

A `file`

B `less`

C `cat`

D `history`

E `view`

12. CheckResetShow Solution

13.

14.

5. Which Bash

shortcut or command is used for completing commands, file names, and options?

A ;

B !*number*

C `history`

D Pressing **Tab**

E Pressing **Esc**+**.**

15. CheckResetShow Solution

16.

17.

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**+**.**

18. CheckResetShow Solution

19.

20.

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**

21. CheckResetShow Solution

22.

23.

8. Which Bash shortcut or command displays the

list of previously executed commands?

A Pressing **Tab**

B `!string`

C `!number`

D `history`

E Pressing **Esc**+**.**

24. CheckResetShow Solution

25.

26.

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**+**.**

27. CheckResetShow Solution

Previous Next

Summary

- The Bash shell is a command interpreter that prompts interactive users to specify Linux commands.
- Many commands have a `--help` option that displays a usage message or screen.
- You can use workspaces to organize multiple application windows.
- The **Activities** button at the upper-left corner of the top bar provides an overview mode that helps to organize windows and to start applications.
- The `file` command scans the beginning of a file and displays what type it is.
- The `head` and `tail` commands display the beginning and end of a file, respectively.
- You can use tab completion to complete file names when typing them as arguments to commands.
- You can use the graphical interface for many administrative tasks. You can disable the interface to preserve resources for running applications.
- You can write many commands in the same line by using the semicolon `;` character, and can run a single command in multiple lines by using the backslash `\` character.

Previous Next

Chapter 3. Manage Files from the Command Line

Lab: Manage Files from the Command Line
Summary
Abstract

Goal	Copy, move, create, delete, and organize files from the Bash shell.
Objectives	• Describe how Linux organizes files, and the purposes of various directories in the file-system hierarchy. • Specify the absolute location and relative location of files to the current working directory, determine and change the working directory, and list the contents of directories. • Create, copy, move, and remove files and directories. • Create multiple file name references to the same file with hard links and symbolic (or "soft") links. • Efficiently run commands that affect many files by using pattern matching features of the Bash shell.
Sections	• Describe Linux File System Hierarchy Concepts (and Quiz) • Specify Files by Name (and Quiz) • Manage Files with Command-line Tools (and Guided Exercise) • Make Links Between Files (and Guided Exercise) • Match File Names with Shell Expansions (and Quiz)
Lab	Manage Files from the Command Line

Describe Linux File System Hierarchy Concepts

Objectives

Describe how Linux organizes files, and the purposes of various directories in the file-system hierarchy.

The File-system Hierarchy

The Linux system stores all files on file systems, which are organized into a single inverted tree known as a file-system hierarchy. This hierarchy is an inverted tree because the tree root is at the top, and the branches of directories and subdirectories stretch below the root.

Figure 3.1: Significant file-system directories in Red Hat Enterprise Linux 9

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, then refer to that directory as `/etc`. Likewise, if the `/etc` directory contains a file that is named `issue`, then refer to that file as `/etc/issue`.

Subdirectories of `/` are used for standardized purposes to organize files by type and purpose to make it easier to find files. For example, in the root directory, the `/boot` subdirectory is used for storing files 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 might be modified or appended by active processes.
- *Persistent* content remains after a reboot, such as configuration settings.
- *Runtime* content from a process or from the system is deleted on reboot.

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

Table 3.1. Significant Red Hat Enterprise Linux Directories

Location	Purpose
`/boot`	Files to start the boot process.
`/dev`	Special device files that the system uses to access hardware.
`/etc`	System-specific configuration files.
`/home`	Home directory, where regular users store their data and configuration files.
`/root`	Home directory for the administrative superuser, `root`.
`/run`	Runtime data for processes that started since the last boot. This data includes process ID files and lock files. The contents of this directory are re-created on reboot. This directory consolidates the `/var/run` and `/var/lock` directories from earlier versions of Red Hat Enterprise Linux.
`/tmp`	A world-writable space for temporary files. Files that are not accessed, changed, or modified for 10 days are deleted from this directory automatically. The `/var/tmp` directory is also a temporary directory, in which files that are not accessed, changed, or modified in more than 30 days are deleted automatically.

Location	Purpose
`/usr`	Installed software, shared libraries, including files, and read-only program data. Significant subdirectories in the `/usr` directory include the following commands: • `/usr/bin`: User commands • `/usr/sbin`: System administration commands • `/usr/local`: Locally customized software
`/var`	System-specific variable data should persist between boots. Files that dynamically change, such as databases, cache directories, log files, printer-spooled documents, and website content, might be found under `/var`.

Important

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

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

Earlier versions of Red Hat Enterprise Linux had distinct directories with 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

Next

Quiz: Describe Linux File System Hierarchy Concepts

Choose the correct answers to the following questions:

1.

2.

1. Which directory contains persistent, system-specific configuration data?

A `/etc`

B `/root`

C `/run`

D `/usr`

3. CheckResetShow Solution

4.

5.

2. Which directory is the top of the system's file-system hierarchy?

A `/etc`

B `/`

C `/home/root`

D `/root`

6. CheckResetShow Solution

7.

8.

3. Which directory contains user home directories?

A `/`

B `/home`

C `/root`

D `/user`

9. CheckResetShow Solution

10.

11.

4. Which directory contains files to boot the system?

A `/boot`

B `/home/root`

C `/bootable`

D `/etc`

12. CheckResetShow Solution

13.

14.

5. Which directory contains system files to access hardware?

A `/etc`

B `/run`

C `/dev`

D `/usr`

15. CheckResetShow Solution

16.

17.

6. Which directory is the administrative superuser's home directory?

A `/etc`

B `/`

C `/home/root`

D `/root`

18. CheckResetShow Solution

19.

20.

7. Which directory contains regular commands and utilities?

A `/commands`

B `/run`

C `/usr/bin`

D `/usr/sbin`

21. CheckResetShow Solution

22.

23.

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

A `/tmp`

B `/etc`

C `/run`

D `/var`

24. CheckResetShow Solution

25.

26.

9. Which directory contains installed software programs and libraries?

A	`/etc`
B	`/lib`
C	`/usr`
D	`/var`

27. CheckResetShow Solution

Previous Next

Specify Files by Name

Objectives

Specify the absolute location and relative location of files to the current working directory, determine and change the working directory, and list the contents of directories.

Absolute Paths and Relative Paths

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

Important

A space character is acceptable in a Linux file name. The shell also uses spaces to distinguish options and arguments on the command line. If a command includes a file with a space in its name, then the shell can misinterpret the command and assume that the file name is multiple arguments. To avoid this mistake, surround such file names in quotation marks so that the shell interprets the name as a single argument. Red Hat recommends avoiding spaces at all in file names.

Absolute Paths

An *absolute path* is a *fully qualified name* that specifies the exact location of the file in the file-system hierarchy. The absolute path begins at the root (`/`) directory and includes each subdirectory that must be traversed to reach the specific file. Every file in a file system has a unique absolute path name, which is 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 can also be located relative to the current working directory of your shell prompt.

The Current Working Directory and Relative Paths

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

Similar to an absolute path, a *relative* path identifies a unique location, and specifies only the necessary path to reach the location from the working directory. Relative path names follow this rule: a path name with *anything other than* a forward slash as the first character is a relative path name. For example, relative to the `/var` directory, the message log file is `log/messages`.

Linux file systems, including `ext4`, `XFS`, `GFS2`, and `GlusterFS`, are case-sensitive. Creating the `FileCase.txt` and `filecase.txt` files in the same directory results in two unique files.

Non-Linux file systems might work differently. For example, VFAT, Microsoft NTFS, and Apple HFS+ have *case-preserving behavior*. Although these file systems are *not* case-sensitive, they do display file names with the file's original capitalization. By creating the files in the preceding example on a VFAT file system, both names would point to the same file instead of to two different files.

Navigate Paths in the File System

The `pwd` command displays the full path name of the current working directory for that shell. This command helps you to determine the syntax to reach files by 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, then it changes 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 ~]$
```

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 the `/home/user/Videos` directory, only the `Videos` directory is displayed. The prompt displays the tilde character (~) when your current working directory is your home directory.

The `touch` command updates the time stamp of a file to the current date and time without otherwise modifying it. This command is useful for creating empty files, and can be used for practice, because when you use the `touch` command with a file name that does not exist, the file is 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 options 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 0
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Desktop
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Documents
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Downloads
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Music
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Pictures
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Public
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Templates
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Videos
[user@host ~]$ ls -la
total 40
drwx------. 17 user user 4096 Mar  2 03:07 .
```

```
drwxr-xr-x.  4 root root   35 Feb 10 10:48 ..
drwxr-xr-x.  4 user user   27 Mar  2 03:01 .ansible
-rw-------.  1 user user  444 Mar  2 04:32 .bash_history
-rw-r--r--.  1 user user   18 Aug  9  2021 .bash_logout
-rw-r--r--.  1 user user  141 Aug  9  2021 .bash_profile
-rw-r--r--.  1 user user  492 Aug  9  2021 .bashrc
drwxr-xr-x.  9 user user 4096 Mar  2 02:45 .cache
drwxr-xr-x.  9 user user 4096 Mar  2 04:32 .config
drwxr-xr-x.  2 user user    6 Mar  2 02:45 Desktop
drwxr-xr-x.  2 user user    6 Mar  2 02:45 Documents
...output omitted...
```

At the top of the listing are two special directories. One dot (`.`) refers to the current directory, and two dots (`..`) refer to the parent directory. These special directories exist in every directory on the system, and they are useful when using file management commands.

Important

File names that begin with a dot (`.`) indicate *hidden* files; you cannot see them in the normal view with `ls` and other commands. This behavior 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, and prevent 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*.

You can also use the tilde (~) special character in combination with other commands for better interaction with the home directory.

```
[user@host ~]$ cd /var/log/
[user@host log]$ ls -l ~
total 0
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Desktop
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Documents
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Downloads
```

```
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Music
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Pictures
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Public
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Templates
drwxr-xr-x. 2 user user 6 Mar  2 02:45 Videos
[user@host ~]$
```

The `cd` command has many options. Some options are useful to practice early and to use often. The `cd -` command changes to the previous directory, where the user was *previously* to the current directory. The following example illustrates this behavior, and alternates 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 where the current location is either the source or destination argument, and avoids the need to type 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

https://www.gnu.org/software/libc/manual/html_node/File-Name-Resolution.html

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

UTF-8 and Unicode

Previous Next

Quiz: Specify Files by Name

Choose the correct answers to the following questions:

1.

2.

1. Which command returns to your current home directory, assuming that the current working directory is `/tmp` and your home directory is `/home/user`?

A `cd`

B `cd ..`

C `cd .`

D `cd *`

E `cd /home`

3. CheckResetShow Solution

4.

5.

2. Which command displays the absolute

path name of the current location?

A `cd`

B `pwd`

C `ls ~`

D `ls -d`

6. CheckResetShow Solution

7.

8.

3. Which command returns you to the working directory before the current working directory?

A `cd -`

B `cd -p`

C `cd ~`

D `cd ..`

9. CheckResetShow Solution

10.

11.

4. Which command changes the working directory up two levels from the current location?

A `cd ~/..`

B `cd ../ ..`

C `cd ../..`

D `cd ~/`

12. CheckResetShow Solution

13.

14.

5. Which command lists files in the current location, with a long format, and including hidden files?

A `llong ~`

B `ls -a`

C `ls -l`

D `ls -al`

15. CheckResetShow Solution

16.

17.

6. Which command creates an empty file called `helloworld.py` in the `user` home directory, assuming that your current directory is `/home`?

A `touch ./helloworld.py`

B `touch ~/helloworld.py`

C `touch helloworld.py`

D `touch ../helloworld.py`

18. CheckResetShow Solution

19.

20.

7. Which command changes the working directory to the parent of the

current
location?

A `cd ~`

B `cd ..`

C `cd ../..`

D `cd -u1`

21. CheckResetShow Solution

22.

23.

8. Which command changes the working directory to `/tmp` if the current working directory is `/home/student`?

A `cd tmp`

B `cd ..`

C `cd ../../tmp`

D `cd ~tmp`

24. CheckResetShow Solution

Previous Next

Manage Files with Command-line Tools

Objectives

Create, copy, move, and remove files and directories.

Command-line File Management

Creating, copying, moving, and, removing files and directories are common operations for a system administrator. Without options, some commands are used to interact with files, or they can manipulate directories with the appropriate set of options.

Be aware of the options that are used when running a command. The meaning of some options might differ between commands.

Create Directories

The `mkdir` command creates one or more directories or subdirectories. It takes as an argument a list of paths to the directories that you want to create.

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 ProjectZ
[user@host Documents]$ ls
ProjectX  ProjectY ProjectZ
```

If the directory exists, or a parent directory of the directory that you are trying to create does not exist, then the `mkdir` command fails and it displays an error.

The `mkdir` command `-p` (*parent*) option creates any missing parent directories for the requested destination. In the following example, the `mkdir` command creates three `ChapterN` subdirectories with one command. The `-p` option creates the missing `Thesis` parent directory.

```
[user@host Documents]$ mkdir -p Thesis/Chapter1 Thesis/Chapter2 Thesis/Chapter3
```

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

Thesis/Chapter1:

Thesis/Chapter2:

Thesis/Chapter3:
```

Use the `mkdir` command `-p` option with caution, because spelling mistakes can create unintended directories without generating error messages. In the following example, imagine that you are trying to create a `Watched` subdirectory in the `Videos` directory, but you accidentally omitted 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 fails because the `Video` directory does not exist. If you had used the `mkdir` command with the `-p` option, then the `Video` directory would be created unintentionally. The `Watched` subdirectory would be created in that incorrect directory.

Copy Files and Directories

The `cp` command copies a file, and creates a file either in the current directory or in a different specified directory.

```
[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
```

You can also use the `cp` command to copy multiple files to a directory. In this scenario, the last argument must be a directory. The copied files retain their original names in the new directory. If a file with the same name exists in the target directory, then the existing file is overwritten.

Note

By default, the `cp` command does not copy directories; it ignores them.

In the following example, two directories are listed as arguments, the `Thesis` and `ProjectX` directories. The last argument, the `ProjectX` directory, is the target and is valid as a destination. The `Thesis` argument is ignored by the `cp` command, because it is intended to be copied and it is a directory.

```
[user@host Documents]$ cp thesis_chapter1.txt thesis_chapter2.txt Thesis ProjectX
cp: -r not specified; omitting directory 'Thesis'
[user@host Documents]$ ls Thesis ProjectX
ProjectX:
thesis_chapter1.txt  thesis_chapter2.txt


Thesis:
Chapter1  Chapter2  Chapter3
```

The `Thesis` directory failed to copy, but the copying of the `thesis_chapter1.txt` and `thesis_chapter2.txt` files succeeded.

You can copy directories and their contents by using the `cp` command `-r` option. Keep in mind that you can use the `.` and `..` special directories in command combinations. In the following example, the `Thesis` directory and its contents are copied to the `ProjectY` directory.

```
[user@host Documents]$ cd ProjectY
[user@host ProjectY]$ cp -r ../Thesis/ .
[user@host ProjectY]$ ls -lR
.:
total 0
drwxr-xr-x. 5 user user 54 Mar  7 15:08 Thesis
```

```
./Thesis:
total 0
drwxr-xr-x. 2 user user 6 Mar  7 15:08 Chapter1
drwxr-xr-x. 2 user user 6 Mar  7 15:08 Chapter2
drwxr-xr-x. 2 user user 6 Mar  7 15:08 Chapter3

./Thesis/Chapter1:
total 0

./Thesis/Chapter2:
total 0

./Thesis/Chapter3:
total 0
```

Move Files and Directories

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

Use the `mv` command to *rename* a file. In the following example, the `mv thesis_chapter2.txt` command renames the `thesis_chapter2.txt` file to `thesis_chapter2_reviewed.txt` in the same directory.

```
[user@host Documents]$ ls -l
-rw-r--r--. 1 user user  7100 Mar  7 14:37 thesis_chapter1.txt
-rw-r--r--. 1 user user 11431 Mar  7 14:39 thesis_chapter2.txt
...output omitted...
[user@host Documents]$ mv thesis_chapter2.txt thesis_chapter2_reviewed.txt
[user@host Documents]$ ls -l
-rw-r--r--. 1 user user  7100 Mar  7 14:37 thesis_chapter1.txt
-rw-r--r--. 1 user user 11431 Mar  7 14:39 thesis_chapter2_reviewed.txt
...output omitted...
```

Use the `mv` command to *move* a file to a different directory. In the next example, the `thesis_chapter1.txt` file is moved from the `~/Documents` directory to the `~/Documents/Thesis/Chapter1` directory. You can use the `mv` command `-v` option to display a detailed output of the command operations.

```
[user@host Documents]$ ls Thesis/Chapter1
[user@host Documents]$
[user@host Documents]$ mv -v thesis_chapter1.txt Thesis/Chapter1
renamed 'thesis_chapter1.txt' -> 'Thesis/Chapter1/thesis_chapter1.txt'
[user@host Documents]$ ls Thesis/Chapter1
thesis_chapter1.txt
[user@host Documents]$ ls -l
-rw-r--r--. 1 user user 11431 Mar  7 14:39 thesis_chapter2_reviewed.txt
...output omitted...
```

Remove Files and Directories

The `rm` command removes files. By default, `rm` does not remove directories. You can use the `rm` command `-r` or the `--recursive` option to enable the `rm` command to remove directories and their contents. The `rm -r` command traverses each subdirectory first, and individually removes their files before removing each directory.

In the following example, the `rm` command removes the `thesis_chapter1.txt` file without options, but to remove the `Thesis/Chapter1` directory, you must add the `-r` option.

```
[user@host Documents]$ ls -l Thesis/Chapter1
-rw-r--r--. 1 user user 7100 Mar  7 14:37 thesis_chapter1.txt
[user@host Documents]$ rm Thesis/Chapter1/thesis_chapter1.txt
[user@host Documents]$ rm Thesis/Chapter1
rm: cannot remove 'Thesis/Chapter1': Is a directory
[user@host Documents]$ rm -r Thesis/Chapter1
[user@host Documents]$ ls -l Thesis
drwxr-xr-x. 2 user user 6 Mar  7 12:37 Chapter2
drwxr-xr-x. 2 user user 6 Mar  7 12:37 Chapter3
```

Important

Red Hat Enterprise Linux does not provide a command-line undelete feature, nor a "trash bin" from which you can restore files held for deletion. A trash bin is a component of a desktop environment such as GNOME, but is not used by commands that are run from a shell.

It is a good idea to verify your current working directory before you remove a file or directory by using relative paths.

```
[user@host Documents]$ pwd
/home/user/Documents
[user@host Documents]$ ls -l thesis*
-rw-r--r--. 1 user user 11431 Mar  7 14:39 thesis_chapter2_reviewed.txt
[user@host Documents]$ rm thesis_chapter2_reviewed.txt
[user@host Documents]$ ls -l thesis*
ls: cannot access 'thesis*': No such file or directory
```

You can use the `rm` command `-i` option to interactively prompt for confirmation before deleting. This option is essentially the opposite of using the `rm` command `-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.txt'? y
rm: remove directory 'Thesis/Chapter2'? y
rm: remove directory 'Thesis/Chapter3'? y
rm: remove directory 'Thesis'? y
```

Warning

If you specify both the `-i` and `-f` options, then the `-f` option takes priority and you are not prompted for confirmation before `rm` removes files.

You can also use the `rmdir` command to remove empty directories. Use the `rm` command `-r` option to remove non-empty directories.

```
[user@host Documents]$ pwd
/home/user/Documents
[user@host Documents]$ rmdir ProjectZ
[user@host Documents]$ rmdir ProjectX
rmdir: failed to remove 'ProjectX': Directory not empty
[user@host Documents]$ rm -r ProjectX
[user@host Documents]$
```

References

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

Previous Next

Guided Exercise: Manage Files with Command-line Tools

In this exercise, you create, organize, copy, and remove files and directories.

Outcomes

- Create, organize, copy, and remove files and directories.

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start files-manage
```

Instructions

1. Log in to the `servera` machine as the `student` user. In the `student` user's home directory, create three subdirectories: `Music`, `Pictures`, and `Videos`.

1. Use the `ssh` command to log in to the `servera` machine as the `student` user. The systems are configured to use SSH keys for authentication; therefore, a password is not required.

```
[student@workstation ~]$ ssh student@servera
...output omitted...
```

```
[student@servera ~]$
```

4. In the `student` user's home directory, use the `mkdir` command to create three subdirectories: `Music`, `Pictures`, and `Videos`.

```
[student@servera ~]$ mkdir Music Pictures Videos
```

2. Use the `touch` command to create sets of empty practice files to use during this lab. In each set, replace x with the numbers 1 through 6.
    1. Create six files with names of the form `song`*x*`.mp3`.
    2. Create six files with names of the form `snap`*x*`.jpg`.
    3. Create six files with names of the form `film`*x*`.avi`.

```
[student@servera ~]$ touch song1.mp3 song2.mp3 song3.mp3 \
song4.mp3 song5.mp3 song6.mp3
[student@servera ~]$ touch snap1.jpg snap2.jpg snap3.jpg \
snap4.jpg snap5.jpg snap6.jpg
[student@servera ~]$ touch film1.avi film2.avi film3.avi \
film4.avi film5.avi film6.avi
[student@servera ~]$ ls -l
total 0
-rw-r--r--. 1 student student 0 Mar  7 20:58 film1.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film2.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film3.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film4.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film5.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film6.avi
drwxr-xr-x. 2 student student 6 Mar  7 20:58 Music
drwxr-xr-x. 2 student student 6 Mar  7 20:58 Pictures
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap1.jpg
```

```
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap2.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap3.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap4.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap5.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap6.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 song1.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song2.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song3.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song4.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song5.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song6.mp3
```

```
drwxr-xr-x. 2 student student 6 Mar  7 20:58 Videos
```

31. Move the song files (`.mp3` extension) to the `Music` directory, the snapshot files (`.jpg` extension) to the `Pictures` directory, and the movie files (`.avi` extension) to the `Videos` directory.

```
[student@servera ~]$ mv song1.mp3 song2.mp3 song3.mp3 \
song4.mp3 song5.mp3 song6.mp3 Music
[student@servera ~]$ mv snap1.jpg snap2.jpg snap3.jpg \
snap4.jpg snap5.jpg snap6.jpg Pictures
[student@servera ~]$ mv film1.avi film2.avi film3.avi \
film4.avi film5.avi film6.avi Videos
[student@servera ~]$ ls -l Music Pictures Videos
Music:
total 0
-rw-r--r--. 1 student student 0 Mar  7 20:58 song1.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song2.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song3.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song4.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song5.mp3
-rw-r--r--. 1 student student 0 Mar  7 20:58 song6.mp3

Pictures:
```

```
total 0
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap1.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap2.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap3.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap4.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap5.jpg
-rw-r--r--. 1 student student 0 Mar  7 20:58 snap6.jpg

Videos:
total 0
-rw-r--r--. 1 student student 0 Mar  7 20:58 film1.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film2.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film3.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film4.avi
-rw-r--r--. 1 student student 0 Mar  7 20:58 film5.avi
```

```
-rw-r--r--. 1 student student 0 Mar  7 20:58 film6.avi
```

64. Create three subdirectories for organizing your files, and name the subdirectories `friends`, `family`, and `work`. Use a single command to create all three subdirectories at the same time.

```
[student@servera ~]$ mkdir friends family work
[student@servera ~]$ ls -l
total 0
drwxr-xr-x. 2 student student   6 Mar  7 21:01 family
drwxr-xr-x. 2 student student   6 Mar  7 21:01 friends
drwxr-xr-x. 2 student student 108 Mar  7 21:00 Music
drwxr-xr-x. 2 student student 108 Mar  7 21:00 Pictures
drwxr-xr-x. 2 student student 108 Mar  7 21:00 Videos
```

```
drwxr-xr-x. 2 student student   6 Mar  7 21:01 work
```

73. Copy files that contain numbers `1` and `2` to the `friends` directory, and files that contain numbers `3` and `4` to the `family` directory. Keep in mind that you are

making copies; therefore, the original files must remain in their original locations after you complete the step.

When you copy files from multiple locations into a single location, Red Hat recommends that you change to the destination directory before you copy the files. Use the simplest path syntax, whether absolute or relative, to reach the source for each file management task.

1. Copy files that contain numbers `1` and `2` to the `friends` directory.

```
[student@servera ~]$ cd friends
[student@servera friends]$ cp ~/Music/song1.mp3 ~/Music/song2.mp3 \
~/Pictures/snap1.jpg ~/Pictures/snap2.jpg ~/Videos/film1.avi \
~/Videos/film2.avi .
[student@servera friends]$ ls -l
total 0
-rw-r--r--. 1 student student 0 Mar  7 21:02 film1.avi
-rw-r--r--. 1 student student 0 Mar  7 21:02 film2.avi
-rw-r--r--. 1 student student 0 Mar  7 21:02 snap1.jpg
-rw-r--r--. 1 student student 0 Mar  7 21:02 snap2.jpg
-rw-r--r--. 1 student student 0 Mar  7 21:02 song1.mp3
```

```
-rw-r--r--. 1 student student 0 Mar  7 21:02 song2.mp3
```

13. Copy files that contain numbers `3` and `4` to the `family` directory.

```
[student@servera friends]$ cd ../family
[student@servera family]$ cp ~/Music/song3.mp3 ~/Music/song4.mp3 \
~/Pictures/snap3.jpg ~/Pictures/snap4.jpg ~/Videos/film3.avi \
~/Videos/film4.avi .
[student@servera family]$ ls -l
total 0
total 0
-rw-r--r--. 1 student student 0 Mar  7 21:04 film3.avi
-rw-r--r--. 1 student student 0 Mar  7 21:04 film4.avi
-rw-r--r--. 1 student student 0 Mar  7 21:04 snap3.jpg
-rw-r--r--. 1 student student 0 Mar  7 21:04 snap4.jpg
```

```
-rw-r--r--. 1 student student 0 Mar  7 21:04 song3.mp3
```

```
-rw-r--r--. 1 student student 0 Mar  7 21:04 song4.mp3
```

74. Copy the `family` and `friends` directories and their contents to the `work` directory.

```
[student@servera family]$ cd ../work
[student@servera work]$ cp -r ~/family ~/friends .
[student@servera work]$ ls -l
total 0
drwxr-xr-x. 2 student student 108 Mar  7 21:05 family
```

```
drwxr-xr-x. 2 student student 108 Mar  7 21:05 friends
```

80. Your project tasks are now complete, and it is time to clean up the directories. Use the `rm -r` command to recursively delete the `family`, `friends`, and `work` directories and their contents.

```
[student@servera work]$ cd ..
[student@servera ~]$ rm -r family friends work
[student@servera ~]$ ls -l
total 0
drwxr-xr-x. 2 student student 108 Mar  7 21:00 Music
drwxr-xr-x. 2 student student 108 Mar  7 21:00 Pictures
```

```
drwxr-xr-x. 2 student student 108 Mar  7 21:00 Videos
```

87. Return to the `workstation` system as the `student` user.

```
[student@servera ~]$ exit
logout
Connection to servera closed.
```

```
[student@workstation ~]$
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish files-manage
```

This concludes the section.

Previous Next

Make Links Between Files

Objectives

Make multiple file names reference the same file with hard links and symbolic (or "soft") links.

Manage Links Between Files

You can create multiple file names that point to the same file. These file names are called *links*.

You can create two types of links: a *hard link*, or a *symbolic link* (sometimes called a *soft link*). Each way has its advantages and disadvantages.

Create Hard Links

Every file starts with a single hard link, from its initial name to the data on the file system. When you create a hard link to a file, you create another name that points to that same data. The new hard link acts exactly like the original file name. After the link is created, you cannot tell the difference between the new hard link and the original name of the file.

You can determine whether a file has multiple hard links by using the `ls -l` command. One item that it reports is each file's *link count*, the number of hard links that the file has. In the next example, the link count of the `newfile.txt` file is 1. It has exactly one absolute path, which is the `/home/user/newfile.txt` location.

```
[user@host ~]$ pwd
/home/user
[user@host ~]$ ls -l newfile.txt
-rw-r--r--. 1 user user 0 Mar 11 19:19 newfile.txt
```

You can use the `ln` command to create a hard link (another file name) that points to an existing file. The command needs at least two arguments: a path to the existing file, and the path to the hard link that you want to create.

The following example creates a hard link called `newfile-hlink2.txt` for the existing `newfile.txt` file in the `/tmp` directory.

```
[user@host ~]$ ln newfile.txt /tmp/newfile-hlink2.txt
[user@host ~]$ ls -l newfile.txt /tmp/newfile-hlink2.txt
-rw-rw-r--. 2 user user 12 Mar 11 19:19 newfile.txt
-rw-rw-r--. 2 user user 12 Mar 11 19:19 /tmp/newfile-hlink2.txt
```

To determine whether two files are hard linked, use the `ls` command `-i` option to list each file's *inode number*. If the files are on the same file system and their inode numbers are the same, then the files are hard links that point to the same data file content.

```
[user@host ~]$ ls -il newfile.txt /tmp/newfile-hlink2.txt
8924107 -rw-rw-r--. 2 user user 12 Mar 11 19:19 newfile.txt
8924107 -rw-rw-r--. 2 user user 12 Mar 11 19:19 /tmp/newfile-hlink2.txt
```

Important

Hard links that reference the same file share the inode structure with the link count, access permissions, user and group ownership, time stamps, and file content. When that information is changed for one hard link, then the other hard links for the same file also show the new information. This behavior is because each hard link points to the same data on the storage device.

Even if the original file is deleted, you can still access the contents of the file provided that at least one other hard link exists. Data is deleted from storage only when the last hard link is deleted, which makes the file contents unreferenced by any hard link.

```
[user@host ~]$ rm -f newfile.txt
[user@host ~]$ ls -l /tmp/newfile-hlink2.txt
-rw-rw-r--. 1 user user 12 Mar 11 19:19 /tmp/newfile-hlink2.txt
[user@host ~]$ cat /tmp/newfile-hlink2.txt
```

```
Hello World
```

Limitations of Hard Links

Hard links have some limitations. First, you can use hard links only with regular files. You cannot use the `ln` command to create a hard link to a directory or special file.

Second, you can use hard links only if both files are on the same *file system*. The file-system hierarchy can be composed of multiple storage devices. Depending on the configuration of your system, when you change into a new directory, that directory and its contents might be stored on a different file system.

You can use the `df` command to list the directories that are on different file systems. For example, you might see the following output:

```
[user@host ~]$ df
Filesystem                   1K-blocks    Used Available Use% Mounted on
devtmpfs                        886788       0    886788   0% /dev
tmpfs                           902108       0    902108   0% /dev/shm
tmpfs                           902108    8696    893412   1% /run
tmpfs                           902108       0    902108   0% /sys/fs/cgroup
/dev/mapper/rhel_rhel9--root  10258432 1630460   8627972  16% /
/dev/sda1                      1038336  167128    871208  17% /boot
tmpfs                           180420       0    180420   0% /run/user/1000
```

Files in two different "Mounted on" directories and their subdirectories are on different file systems. So, in the system in this example, you can create a hard link between the `/var/tmp/link1` and `/home/user/file` files, because they are both subdirectories of the `/` directory but not of any other directory on the list. However, you cannot create a hard link between the `/boot/test/badlink` and `/home/user/file` files. The first file is in a subdirectory of the `/boot` directory (on the "Mounted on" list) and it is in the `/dev/sda1` file system. The second file is in the `/dev/mapper/rhel_rhel9–root` file system.

Create Symbolic Links

The `ln` command `-s` option creates a symbolic link, which is also called a "soft link". A symbolic link is not a regular file, but a special type of file that points to an existing file or directory.

Symbolic links have some advantages over hard links:

- Symbolic links can link two files on different file systems.
- Symbolic links can point to a directory or special file, not just to a regular file.

In the following example, the `ln -s` command creates a symbolic link for the `/home/user/newfile-link2.txt` file. The name for the symbolic link is `/tmp/newfile-symlink.txt`.

```
[user@host ~]$ ln -s /home/user/newfile-link2.txt /tmp/newfile-symlink.txt
[user@host ~]$ ls -l newfile-link2.txt /tmp/newfile-symlink.txt
-rw-rw-r--. 1 user user 12 Mar 11 19:19 newfile-link2.txt
lrwxrwxrwx. 1 user user 11 Mar 11 20:59 /tmp/newfile-symlink.txt -> /home/user/newfil
e-link2.txt
[user@host ~]$ cat /tmp/newfile-symlink.txt
Symbolic Hello World
```

In the preceding example, the first character of the long listing for the `/tmp/newfile-symlink.txt` file is `l` (letter l) instead of `-`. This character indicates that the file is a symbolic link and not a regular file.

When the original regular file is deleted, the symbolic link still points to the file but the target is gone. A symbolic link that points to a missing file is called a "dangling symbolic link".

```
[user@host ~]$ rm -f newfile-link2.txt
[user@host ~]$ ls -l /tmp/newfile-symlink.txt
lrwxrwxrwx. 1 user user 11 Mar 11 20:59 /tmp/newfile-symlink.txt -> /home/user/newfil
e-link2.txt
[user@host ~]$ cat /tmp/newfile-symlink.txt
cat: /tmp/newfile-symlink.txt: No such file or directory
```

Important

One side-effect of the dangling symbolic link in the preceding example is that if you create a file with the same name as the deleted file (`/home/user/newfile-link2.txt`), then the symbolic link is no longer "dangling" and points to the new file. Hard links do not work in this way. If you delete a hard link and then use normal tools (rather than `ln`) to create a file with the same name, then the new file is not linked to the old file. Consider the following way to compare hard links and symbolic links, to understand how they work:

- A hard link points a name to data on a storage device.
- A symbolic link points a name to another name, which points to data on a storage device.

A symbolic link can point to a directory. The symbolic link then acts like the directory. If you use `cd` to change to the symbolic link, then the current working directory becomes the linked directory. Some tools might track that you followed a symbolic link to get there. For example, by default, `cd` updates your current working directory by using the name of the symbolic link rather than the name of the actual directory. If you want to update the current working directory by using the name of the actual directory, then you can use the `-P` option.

The following example creates a symbolic link called `/home/user/configfiles` that points to the `/etc` directory.

```
[user@host ~]$ ln -s /etc /home/user/configfiles
[user@host ~]$ cd /home/user/configfiles
[user@host configfiles]$ pwd
/home/user/configfiles
[user@host configfiles]$ cd -P /home/user/configfiles
[user@host etc]$ pwd
/etc
```

References

`ln`(1) man page

`info ln` (*Make links between files*)

Guided Exercise: Make Links Between Files

In this exercise, you create hard links and symbolic links and compare the results.

Outcomes

- Create hard links and symbolic links between files.

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start files-make
```

Instructions

1. Use the `ssh` command to log in to the `servera` machine as the `student` user. The system's configuration supports the use SSH keys for authentication; therefore, you do not require a password.

```
2. [student@workstation ~]$ ssh student@servera
3. ...output omitted...
```

```
[student@servera ~]$
```

4. Create a hard link called `/home/student/links/file.hardlink` for the file `/home/student/files/target.file`. Verify the link count for the original file and the new linked file.
    1. View the link count for the file `/home/student/files/target.file`.

```
2. [student@servera ~]$ ls -l files/target.file
3. total 4
```

```
-rw-r--r--. 1 student student 11 Mar  3 06:51 files/target.file
```

    4. Create a hard link called `/home/student/links/file.hardlink`. Link it to the file `/home/student/files/target.file`.

```
[student@servera ~]$ ln /home/student/files/target.file \
/home/student/links/file.hardlink
```

6. Verify the link count for the original file `/home/student/files/target.file` and the new linked file, `/home/student/files/file.hardlink`. The link count should be `2` for both files.

```
[student@servera ~]$ ls -l files/target.file links/file.hardlink
-rw-r--r--. 2 student student 11 Mar  3 06:51 files/target.file
-rw-r--r--. 2 student student 11 Mar  3 06:51 links/file.hardlink
```

5. Create a symbolic link called `/home/student/tempdir` that points to the `/tmp` directory on the `servera` machine. Verify the newly created symbolic link.
    1. Create a symbolic link called `/home/student/tempdir` and link it to the `/tmp` directory.

```
[student@servera ~]$ ln -s /tmp /home/student/tempdir
```

    2. Use the `ls -l` command to verify the newly created symbolic link.

```
[student@servera ~]$ ls -l /home/student/tempdir
lrwxrwxrwx. 1 student student 4 Mar  3 06:55 /home/student/tempdir -> /tmp
```

6. Return to the `workstation` system as the `student` user.

```
[student@servera ~]$ exit
logout
Connection to servera closed.
[student@workstation ~]$
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish files-make
```

This concludes the section.

Previous Next

Match File Names with Shell Expansions

Objectives

Efficiently run commands that affect many files by using pattern matching features of the Bash shell.

Command-line Expansions

When you type a command at the Bash shell prompt, the shell processes that command line through multiple *expansions* before running it. You can use these shell expansions to perform complex tasks that would otherwise be difficult or impossible.

Following are the main expansions that Bash shell performs:

- *Brace expansion*, which can generate multiple strings of characters
- *Tilde expansion*, which expand to a path to a user home directory
- *Variable expansion*, which replaces text with the value that is stored in a shell variable
- *Command substitution*, which replaces text with the output of a command
- *Pathname expansion*, which helps to select one or more files by pattern matching

Pathname expansion, historically called *globbing*, is one of the most useful features of Bash. With this feature, it is easier to manage many files. By using *metacharacters* that "expand" to match the file and path names that you are looking for, commands can act on a focused set of files at once.

Pathname Expansion and Pattern Matching

Pathname expansion expands a pattern of special characters that represent wild cards or classes of characters into a list of file names that match the pattern. Before

the shell runs your command, it replaces the pattern with the list of file names that matched. If the pattern does not match anything, then the shell tries to use the pattern as a literal argument for the command that it runs. The following table lists common metacharacters and pattern classes that are used for pattern matching.

Table 3.2. 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 that is not a space or alphanumeric
[[:digit:]]	Any single digit from 0 to 9
[[:space:]]	Any single white space character, which might include tabs, newlines, carriage returns, form feeds, or spaces

For the next example, imagine that you ran 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]$
```

In the next example, the first two commands use simple pattern matches with the asterisk (*) to match all the file names that start with "a" and all the file names that contain an "a", respectively. The third command uses the asterisk and square brackets to match all the file names that start with "a" or "c".

```
[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
```

The next example also uses question mark (?) characters to match some of those file names. The two commands match only file names with four and five characters in length, respectively.

```
[user@host glob]$ ls ????
able  cast  easy  echo
[user@host glob]$ ls ?????
baker  bravo  delta
```

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 that precedes or follows the brace definition. Brace expansions might be nested, one inside another. You can also use double-dot syntax (..), which expands to a sequence. For example, the `{m..p}` double-dot syntax inside braces expands 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
```

A practical use of brace expansion is to create multiple files or directories.

```
[user@host glob]$ mkdir ../RHEL{7,8,9}
```

```
[user@host glob]$ ls ../RHEL*
RHEL7 RHEL8 RHEL9
```

Tilde Expansion

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

In the following example, the `echo` command is used to display the value of the tilde character. You can also use the `echo` command 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]$ echo ~nonexistinguser
~nonexistinguser
```

Variable Expansion

A variable acts like a named container that stores a value in memory. Variables simplify accessing and modifying the stored data either from the command line or within a shell script.

You can assign data as a value to a variable with 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 tries to use the rest of that string as a variable name and to replace it with the variable value.

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

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

To prevent 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
```

Variable names can contain only letters (uppercase and lowercase), numbers, and underscores. Variable names are case-sensitive and cannot start with a number.

Command Substitution

Command substitution enables the output of a command to replace the command itself on the command line. Command substitution occurs when you enclose a command in parentheses and precede it 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.
```

Note

An earlier form of command substitution uses backticks: `` `command` ``. Although the Bash shell still accepts this format, try to avoid it because it is easy to visually confuse backticks with single quotation marks, and backticks cannot be nested.

Protecting Arguments from Expansion

Many characters have a special meaning in the Bash shell. To prevent 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 protects the following character 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.
```

In the preceding example, with the dollar sign protected from expansion, Bash treats it as a regular character, without variable expansion on `$HOME`.

To protect longer character strings, you can use single quotation marks (`'`) or double quotation marks (`"`) to enclose strings. They have slightly different effects. Single quotation marks stop all shell expansion. Double quotation marks stop *most* shell expansion.

Double quotation marks suppress special characters other than the dollar sign (`$`), backslash (`\`), backtick (`` ` ``), and exclamation point (`!`) from operating inside the quoted text. Double quotation marks block pathname expansion, but still allow command substitution and variable expansion to occur.

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

Use single quotation marks to interpret *all* text between the quotes literally.

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

Important

It is easy to confuse the single quotation mark (') and the command substitution backtick (`), on both the screen and the keyboard. The use of one when you mean to use the other leads to unexpected shell behavior.

References

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

Previous Next

Quiz: Match File Names with Shell Expansions

Choose the correct answers to the following questions:

1.

2.

1. Which pattern matches only file names that end with "b"?

A `b*`

B `*b`

C `*b*`

D `[!b]*`

3. CheckResetShow Solution

4.

5.

2. Which pattern matches only

file names that begin with "b"?

A `b*`

B `*b`

C `*b*`

D `[!b]*`

6. CheckResetShow Solution

7.

8.

3. Which pattern matches only file names where the first character is not "b"?

A `b*`

B `*b`

C `*b*`

D `[!b]*`

9. CheckResetShow Solution

10.

11.

4. Which pattern matches all file names that contain a "b"?

A `b*`

B `*b`

C `*b*`

D `[!b]*`

12. CheckResetShow Solution

13.

14.

5. Which pattern matches only file names that contain a number?

A `*#*`

B `*[[:digit:]]*`

C `*[digit]*`

D `[0-9]`

15. CheckResetShow Solution

16.

17.

6. Which pattern matches only file names that begin with an uppercase letter?

A `^?*`

B `^*`

C `[upper]*`

D `[[:upper:]]*`

E `[[CAP]]*`

18. CheckResetShow Solution

19.

20.

7. Which pattern matches only file names with at least three characters?

A `???*`

B `???`

C	\3*
D	+++*
E	...*

21. CheckResetShow Solution

Previous Next

Summary

- Files on a Linux system are organized into a single inverted tree of directories, a file-system hierarchy.
- Absolute paths start with a forward slash character (`/`) and specify the location of a file in the file-system hierarchy.
- Relative paths do not start with a forward slash character.
- Relative paths specify a file location in relation to the current working directory.
- You can use commands in combination with the dot (`.`), double dot (`..`), and tilde (~) special characters to refer to a file location in the file system.
- The `mkdir`, `rmdir`, `cp`, `mv`, and `rm` commands are key commands to manage files in Linux.
- Hard links and soft links are different ways for multiple file names to point to the same data.
- The Bash shell provides pattern matching, expansion, and substitution features to help you to run commands efficiently.

Previous Next

Chapter 4. Get Help in Red Hat Enterprise Linux

Read Manual Pages

Abstract

Goal	Resolve problems by using local help systems.
Objectives	Find information in local Linux system manual pages.
Sections	Read Manual Pages (and Guided Exercise)
Lab	Get Help in Red Hat Enterprise Linux

Read Manual Pages

Objectives

Find information in local Linux system manual pages.

Introduction to the Linux Manual Pages

One source of documentation that is generally available on the local system is system manual pages or *man pages*. Software packages ship these pages to provide documentation, and you can access them from the command line by using the `man` command. The pages are stored in subdirectories of the `/usr/share/man` directory.

Man pages originated from the historical Linux Programmer's Manual, which because of its size is split into multiple sections. Each section contains information about a particular topic.

Table 4.1. Common Sections of the Linux Manual

Section	Content type	Description
1	User commands	Both executable and shell programs
2	System calls	Kernel routines that are invoked from user space
3	Library functions	Provided by program libraries
4	Special files	Such as device files
5	File formats	For many configuration files and structures
6	Games and screensavers	Historical section for amusing programs
7	Conventions, standards, and miscellaneous	Protocols and file systems

Section	Content type	Description
8	System administration and privileged commands	Maintenance tasks
9	Linux kernel API	Internal kernel calls

To distinguish identical topic names in different sections, man page references include the section number in parentheses after the topic. For example, `passwd`(1) describes the command to change passwords, whereas `passwd`(5) explains the `/etc/passwd` file format for storing local user accounts.

To read specific man pages, use the `man` *`topic`* command. The man pages display contents one screen at a time. The `man` command searches manual sections in alphanumeric order. For example, `man passwd` displays `passwd`(1) by default. To display the man page topic from a specific section, you can use the `man` *`section topic`* command. For example, `man 5 passwd` displays `passwd`(5).

Popular system administration topics are in sections 1 (user commands), 5 (file formats), and 8 (administrative commands). Administrators who use certain troubleshooting tools also use section 2 (system calls). The remaining sections are generally for programmer reference or advanced administration.

Navigate and Search man Pages

It is a critical administration skill to search efficiently for topics and to navigate man pages. Although you can use GUI tools to configure common system resources, using the command-line interface is more efficient. To navigate the command line effectively, you must be able to find the information that you need in the man pages.

The following table lists some navigation commands when viewing man pages:

Table 4.2. Navigate man Pages

Command	Result
Spacebar	Scroll forward (down) one screen.
PageDown	Scroll forward one screen.
PageUp	Scroll backward (up) one screen.
DownArrow	Scroll forward one line.
UpArrow	Scroll backward one line.

Command	Result
D	Scroll forward one half-screen.
U	Scroll backward one half-screen.
/*string*	Search forward for *string* in the man page.
N	Repeat previous search forward in the man page.
Shift+**N**	Repeat previous search backward in the man page.
G	Go to the start of the man page.
Shift+**G**	Go to the end of the man page.
Q	Exit man and return to the command shell prompt.

Important

You can use regular expressions to search in `man` pages. Although simple text search (such as `passwd`) works as expected, regular expressions use metacharacters (such as `$`, `*`, `.`, and `^`) for more sophisticated pattern matching. Therefore, searching with strings that include program expression metacharacters, such as `make $$$`, might yield unexpected results.

You can find more information about regular expressions and syntax in the `regex`(7) man topic.

Read man Pages

Man pages separate each topic into several parts. Most topics use the same headings and follow the same order. Typically, a topic does not feature all headings, because not all headings apply to all topics.

Common headings are as follows:

Table 4.3. Headings

Heading	Description
NAME	Subject name, usually a command or file name, a brief description
SYNOPSIS	Summary of the command syntax
DESCRIPTION	Description to provide a basic understanding of the topic
OPTIONS	Explanation of the command execution options
EXAMPLES	Examples of how to use the command, function, or file
FILES	A list of files and directories that are related to the man page
SEE ALSO	Related information, normally other man page topics
BUGS	Known bugs in the software

Heading	Description
AUTHOR	Information about who contributed to the development of the topic

Search for man Pages by Keyword

Use the `man` command `-k` option (equivalent to the `apropos` command) to search for a keyword in `man` page titles and descriptions. As a result, the keyword search displays a list of keyword-matching man page topics with section numbers. For example, the following command searches for man pages with the word `passwd`:

```
[user@host ~]$ man -k passwd
chgpasswd (8)        - update group passwords in batch mode
chpasswd (8)         - update passwords in batch mode
fgetpwent_r (3)      - get passwd file entry reentrantly
getpwent_r (3)       - get passwd file entry reentrantly
...
passwd (1)           - update user's authentication tokens
passwd (1ossl)       - OpenSSL application commands
passwd (5)           - password file
passwd2des (3)       - RFS password encryption
...
```

The `man` command `-K` (uppercase) option searches for the keyword in the full-text page, not only in the titles and descriptions. A full-text search uses greater system resources and takes more time.

With the full-text page search, the `man` command displays the first page with a match. Press **Q** to exit this first page, and the `man` command displays the next page.

In this example, `man` displays each match, and you can view or skip each one.

```
[user@host ~]# man -K passwd
--Man-- next: cut(1p) [ view (return) | skip (Ctrl-D) | quit (Ctrl-C) ]
Ctrl-D
--Man-- next: logname(1p) [ view (return) | skip (Ctrl-D) | quit (Ctrl-C) ]
Ctrl-D
--Man-- next: sort(1p) [ view (return) | skip (Ctrl-D) | quit (Ctrl-C) ]
```

```
Ctrl-D
--Man-- next: xargs(1) [ view (return) | skip (Ctrl-D) | quit (Ctrl-C) ]
Ctrl-D
--Man-- next: chage(1) [ view (return) | skip (Ctrl-D) | quit (Ctrl-C) ]
Ctrl-C
```

Note

Keyword searches rely on an index that is generated by the `mandb`(8) command, which must be run as `root`.

The `man-db-cache-update` service automatically runs the `mandb` command when installing any package with man pages.

References

`man`(1), `mandb`(8), `man-pages`(7), `less`(1), `intro`(1), `intro`(2), `intro`(5), `intro`(7), and `intro`(8) man pages

Next

Guided Exercise: Read Manual Pages

In this exercise, you practice finding relevant information by using `man` options and arguments.

Outcomes

- Use the `man` Linux manual system and find useful information by searching and browsing.

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start help-manual
```

Instructions

1. On `workstation`, view the `gedit` man page. View the options for editing a specific file by using `gedit` from the command line.

   Use one of the options from the `gedit` man page to open the `/home/student/manual` file by using `gedit` with the cursor at the end of the file.

   1. View the `gedit` man page.

```
[student@workstation ~]$ man gedit
GEDIT(1)    General Commands Manual      GEDIT(1)
NAME
       gedit - text editor for the GNOME Desktop

SYNOPSIS
       gedit [OPTION...] [FILE...] [+LINE[:COLUMN]]
       gedit [OPTION...] -
```

```
...output omitted...
```

   10. In the `gedit` man page, learn the options for editing a specific file from the command line.

```
OPTIONS
...output omitted...
      FILE  Specifies the file to open when gedit starts.
...output omitted...
      +LINE  For the first file, go to the line specified by LINE (do not
   insert a space between the "+" sign and the number). If LINE is missing, g
   o to the last line.
```

```
...output omitted...
```

   Press **q** to quit the man page.

16. Use the `gedit +` command to open the `manual` file. The missing line number next to the `+` option opens a file that is passed as an argument with the cursor at the end of the last line.

```
[student@workstation ~]$ gedit + manual
this is the first line
```

```
the quick brown fox just came over to greet the lazy poodle!
```

Confirm that the file is opened with the cursor at the end of the last line in the file. Press **Ctrl**+**q** to close the application.

2. Read the `su`(1) man page.

If you omit the *user* argument, then the `su` command assumes that the user is `root`. If the `su` command is followed by a single dash (`-`), then it starts a child login shell. Without the dash, the `su` command creates a non-login child shell that matches the user's current environment.

```
[student@workstation ~]$ man 1 su
SU(1)              User Commands                       SU(1)
NAME
    su - run a command with substitute user and group ID


SYNOPSIS
    su [options] [-] [user [argument...]]


DESCRIPTION
    su allows to run commands with a substitute user and group ID.


    When called with no user specified, su defaults to running an interactive sh
ell as root.
...output omitted...
OPTIONS
...output omitted...
       -, -l, --login
           Start the shell as a login shell with an environment similar to a rea
l login.
```

```
...output omitted...
```

> **Note**
>
> Comma-separated options on a single line, such as `-`, `-l`, and `--login`, all result in the same behavior.

Press **q** to quit the man page.

3. The `man` command also has its own manual pages. Open the `man`(1) command manual page.

```
[student@workstation ~]$ man man
MAN(1)             Manual pager utils                                MAN(1)

NAME
   man - an interface to the on-line reference manuals
...output omitted...
DESCRIPTION
       man is the system's manual pager. Each page argument given to man is
       normally the name of a program, utility or function. The manual page
       associated with each of these arguments is then found and displayed.
       A section, if provided, will direct man to look only in that section
       of the manual.
```

```
...output omitted...
```

Press **q** to quit the man page.

16. The `/usr/share/man` directory contains all man pages. Locate the binary, source, and manual pages for the `passwd` utility by using the `whereis` command. Verify that the `/usr/share/man` directory contains the man pages for the `passwd` utility.

```
[student@workstation ~]$ whereis passwd
```

```
passwd: /usr/bin/passwd /etc/passwd /usr/share/man/man1/passwd.1ossl.gz /usr/sha
re/man/man1/passwd.1.gz /usr/share/man/man5/passwd.5.gz
```

18. Use the `man -k zip` command to list the man page with detailed information about a ZIP archive.

```
[student@workstation ~]$ man -k zip
...output omitted...
zipinfo (1) - list detailed information about a ZIP archive
zipnote (1)          - write the comments in zipfile to stdout, edit comments an
d rename files in zipfile
```

```
zipsplit (1)         - split a zipfile into smaller zipfiles
```

23. Use the `man -k boot` command to list the man page with a list of parameters that can be passed to the kernel at boot time.

```
[student@workstation ~]$ man -k boot
binfmt.d (5)         - Configure additional binary formats for executables at bo
ot
bootparam (7)        - introduction to boot time parameters of the Linux kernel
bootup (7)           - System bootup process
```

```
...output omitted...
```

28. Use the `man -k ext4` command to find the command to tune ext4 file-system parameters.

```
[student@workstation ~]$ man -k ext4
...output omitted...
resize2fs (8)        - ext2/ext3/ext4 file system resizer
```

```
tune2fs (8)          - adjust tunable filesystem parameters on ext2/ext3/ext4 fi
lesystems
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish help-manual
```

This concludes the section.

Previous Next

Summary

- Use the `man` command to view man pages and to display information about components of a Linux system, such as files, commands, and functions.
- By convention, to refer to a man page, the name of a page is followed by its section number in parentheses.
- You can use regular expressions to search content in man pages.

Previous Next

Chapter 5. Create, View, and Edit Text Files

Abstract

Goal	Create, view, and edit text files from command output or in a text editor.
Objectives	• Save output or errors to a file with shell redirection, and process command output through multiple command-line programs with pipes. • Create and edit text files from the command line with the `vim` editor. • Set shell variables to run commands, and edit Bash startup scripts to set shell and environment variables to modify the behavior of the shell and programs that are run from the shell.
Sections	• Redirect Output to a File or Program (and Quiz) • Edit Text Files from the Shell Prompt (and Guided Exercise) • Change the Shell Environment (and Guided Exercise)
Lab	Create, View, and Edit Text Files

Redirect Output to a File or Program

Objectives

Save output or errors to a file with shell redirection, and process command output through multiple command-line programs with pipes.

Standard Input, Standard Output, and Standard Error

A running program, or *process*, reads input and writes output. When you run a command from the shell prompt, it normally reads its input from the keyboard and sends its output to the terminal window.

A process uses numbered channels called *file descriptors* to get input and send output. All processes start with at least three file descriptors. *Standard input* (channel 0) reads input from the keyboard. *Standard output* (channel 1) sends normal output to the terminal. *Standard error* (channel 2) sends error messages to the terminal.

If a program opens separate connections to other files, then it might use higher-numbered file descriptors.

Figure 5.1: Process I/O channels (file descriptors)

The next table summarizes the information about the file descriptors:

Table 5.1. Channels (File Descriptors)

Number	Channel name	Description	Default connection	Usage
0	`stdin`	Standard input	Keyboard	Read only
1	`stdout`	Standard output	Terminal	Write only
2	`stderr`	Standard error	Terminal	Write only
3+	*`filename`*	Other files	None	Read, write, or both

Redirect Output to a File

The Input/Output (I/O) redirection changes how the process gets its input or output. Instead of getting input from the keyboard, or sending output and errors to the terminal, the process can read from or write to files. With redirection, you can save the messages to a file instead of displaying the output on the terminal. Alternatively, you can use redirection to discard output or errors, so they are not displayed on the terminal or saved.

You can redirect a process `stdout` to suppress the process output from appearing on the terminal. If you redirect `stdout` to a file and the file does not exist, then the file is created. If the file does exist and the redirection does not append to the file, then the redirection overwrites the file's contents. To discard the output of a process, you can redirect to the empty `/dev/null` special file that discards channel output that is redirected to it.

As viewed in the following table, redirecting *only* `stdout` does not suppress displaying `stderr` error messages on the terminal.

Table 5.2. Output Redirection Operators

Usage	Explanation	Visual aid

> *file*	Redirect `stdout` to overwrite a file.	

Usage	Explanation	Visual aid

>> file	Redirect `stdout` to append to a file.	

Usage	Explanation	Visual aid

2> *file*	Redirect `stderr` to overwrite a file.	

Usage	Explanation	Visual aid

2> /dev/null	Discard `stderr` error messages by redirecting them to `/dev/null`.	

Usage	Explanation	Visual aid
> *file* 2>&1		

&> file	Redirect `stdout` and `stderr` to overwrite the same file.	

Usage	Explanation	Visual aid
>> *file* 2>&1	Redirect `stdout` and `stderr`	

&>> *file*	to append to the same file.	

Usage	Explanation	Visual aid

Important

The order of redirection operations is important. The following sequence redirects standard output to the `output.log` file and then redirects standard error messages to the same place as standard output (`output.log`).

```
> output.log 2>&1
```

The next sequence redirects in the opposite order. This sequence redirects standard error messages to the default place for standard output (the terminal window, so no change) and *then* redirects only standard output to the `output.log` file.

```
2>&1 > output.log
```

For this reason, some people prefer to use the merging redirection operators:

- `&> output.log` instead of `> output.log 2>&1`
- `&>> output.log` instead of `>> output.log 2>&1` (in Bash 4 or RHEL 6 and later)

However, system administrators and programmers prefer to avoid the newer merging redirection operators when using alternative shells to `bash` (known as Bourne-compatible shells) for scripting commands. These new redirection operators are not standardized or implemented in those shells, and have other limitations.

Examples for Output Redirection

Simplify many routine administration tasks by using redirection. Use the previous table to assist, and consider the following examples:

Save a time stamp in the `/tmp/saved-timestamp` file for later reference.

```
[user@host ~]$ date > /tmp/saved-timestamp
```

Copy the last 100 lines from the `/var/log/secure` file to the `/tmp/last-100-log-secure` file.

```
[user@host ~]$ tail -n 100 /var/log/secure > /tmp/last-100-log-secure
```

Concatenate all four `step` files into one file in the `tmp` directory.

```
[user@host ~]$ cat step1.sh step2.log step3 step4 > /tmp/all-four-steps-in-one
```

List the home directory's hidden and regular file names, and save the output to the `my-file-names` file.

```
[user@host ~]$ ls -a > my-file-names
```

Append a line to the existing `/tmp/many-lines-of-information` file.

```
[user@host ~]$ echo "new line of information" >> /tmp/many-lines-of-information
```

The next few commands generate error messages because some system directories are inaccessible to normal users. Observe the error message redirection.

Redirect errors from the `find` command to the `/tmp/errors` file when viewing normal command output on the terminal.

```
[user@host ~]$ find /etc -name passwd 2> /tmp/errors
```

Save process output to the `/tmp/output` file and error messages to the `/tmp/errors` file.

```
[user@host ~]$ find /etc -name passwd > /tmp/output 2> /tmp/errors
```

Save process output to the `/tmp/output` file and discard error messages.

```
[user@host ~]$ find /etc -name passwd > /tmp/output 2> /dev/null
```

Store output and generated errors together to the `/tmp/all-message-output` file.

```
[user@host ~]$ find /etc -name passwd &> /tmp/all-message-output
```

Append output and generated errors to the `/tmp/all-message-output` file.

```
[user@host ~]$ find /etc -name passwd >> /tmp/all-message-output 2>&1
```

Construct Pipelines

A *pipeline* is a sequence of one or more commands that are separated by the vertical bar character (|). A pipeline connects the standard output of the first command to the standard input of the next command.

Figure 5.8: Process I/O piping

Use pipelines to manipulate and format the output of a process by other processes before it is output to the terminal. Imagine that data "flows" through the pipeline from one process to another, and is altered by each command in the pipeline that it flows through.

Note

Pipelines and I/O redirection both manipulate standard output and standard input. Pipelines send the standard output from one process to the standard input of another process. Redirection sends standard output to files, or gets standard input from files.

Pipeline Examples

The following list shows some pipeline examples:

Redirect the output of the `ls` command to the `less` command to display it on the terminal one screen at a time.

```
[user@host ~]$ ls -l /usr/bin | less
```

Redirect the output of the `ls` command to the `wc -l` command, which counts the number of received lines from `ls` and prints that value to the terminal.

```
[user@host ~]$ ls | wc -l
```

Redirect the output of the `ls -t` command to the `head` command to display the first 10 lines, with the final result redirected to the `/tmp/first-ten-changed-files` file.

```
[user@host ~]$ ls -t | head -n 10 > /tmp/first-ten-changed-files
```

Pipelines, Redirection, and Appending to a File

When you combine redirection with a pipeline, the shell sets up the entire pipeline first, and then it redirects the input/output. If you use output redirection in the *middle* of a pipeline, then the output goes to the file and not to the next command in the pipeline.

In the next example, the output of the `ls` command goes to the `/tmp/saved-output` file, and the `less` command displays nothing on the terminal.

```
[user@host ~]$ ls > /tmp/saved-output | less
```

The `tee` command overcomes this limitation. In a pipeline, `tee` copies its standard input to its standard output and also redirects its standard output to the files that are given as arguments to the command. If you imagine data as water that flows through a pipeline, then you can visualize `tee` as a "T" joint in the pipe that directs output in two directions.

Figure 5.9: Process I/O piping with tee

Pipeline Examples with the tee Command

The next example redirects the output of the `ls` command to the `/tmp/saved-output` file and passes it to the `less` command, so it is displayed on the terminal one screen at a time.

```
[user@host ~]$ ls -l | tee /tmp/saved-output | less
```

If you use the `tee` command at the end of a pipeline, then the terminal shows the output of the commands in the pipeline and saves it to a file at the same time.

```
[user@host ~]$ ls -t | head -n 10 | tee /tmp/ten-last-changed-files
```

Use the `tee` command `-a` option to append the content to a file instead of overwriting it.

```
[user@host ~]$ ls -l | tee -a /tmp/append-files
```

Important

You can redirect standard error through a pipeline, but you cannot use the merging redirection operators (`&>` and `&>>`). The following example is the correct way to redirect both standard output and standard error through a pipeline:

```
[user@host ~]$ find / -name passwd 2>&1 | less
```

References

`info bash` (*The GNU Bash Reference Manual*)

- Section 3.2.3: Pipelines
- Section 3.6: Redirections

`info coreutils 'tee invocation'` (*The GNU coreutils Manual*)

- Section 17.1: Redirect output to multiple files or processes

`bash`(1), `cat`(1), `head`(1), `less`(1), `mail`(1), `tee`(1), `tty`(1), `wc`(1) man pages

Next

Quiz: Redirect Output to a File or Program

Choose the correct answer to the following questions:

1.

2.

1. Which output redirection operator displays output to a terminal and discards all error messages?

A	&> file
B	2> &> file
C	2> /dev/null
D	1> /dev/null

3. CheckResetShow Solution

4.

5.

2. Which output redirection

operator sends output to a file and sends errors to a different file?

A	> file 2> file2
B	> file 1> file2
C	> file &2> file2
D	| tee file

6. CheckResetShow Solution

7.

8.

3. Which output redirection operator sends both output and errors to a file, and creates it or overwrites its contents?

A	| tee file
B	2 &> file
C	1 &> file

D &> file

9. CheckResetShow Solution

10.

11.

4. Which output redirection operator sends output and errors to the same file and preserves the file content if it exists?

A > file 2> file2

B &> file

C >> file 2>&1

D >> file 1>&1

12. CheckResetShow Solution

13.

14.

5. Which output redirection operator discards all messages that are

normally sent to the terminal?

A > file 2> file2

B &> /dev/null

C &> /dev/null 2> file

D &> file

15. CheckResetShow Solution

16.

17.

6. Which output redirection operator sends output to both the screen and a file at the same time?

A &> /dev/null

B > file 2> file2

C | tee file

D | < file

18. CheckResetShow Solution

19.

20.

7. Which output redirection operator saves output to a file and discards all error messages?

A	&> file
B	| tee file 2> /dev/null
C	> file 1> /dev/null
D	> file 2> /dev/null

21. CheckResetShow Solution

Previous Next

Edit Text Files from the Shell Prompt

Objectives

Create and edit text files from the command line with the `vim` editor.

Edit Files with Vim

The fundamental design principle of Linux is that it supports storage of the information and configuration settings in text-based files. These files follow various structures such as lists of settings, INI-like formats, structured XML or YAML, and others. The advantage of storing files in a text-based structure is that they are edited with any text editor.

Vim is an improved version of the `vi` editor, which is distributed with Linux and UNIX systems. Vim is a highly configurable and efficient editor that provides split-screen editing, color formatting, and highlighting for editing text.

Benefits of the Vim Editor

When a system uses a text-only shell prompt, you should know how to use at least one text editor for editing files. You can then edit text-based configuration files from a terminal window or remote logins through the `ssh` command or the Web Console. You also do not need access to a graphical desktop to edit files on a server, and that server might not need to run a graphical desktop environment.

The key reason to learn Vim is that it is almost always installed by default on a server for editing text-based files. The *Portable Operating System Interface* or *POSIX* standard specified the `vi` editor on Linux, and many other UNIX-like operating systems largely do likewise.

Vim is also used often as the `vi` implementation on other standard operating systems or distributions. For example, macOS currently includes a lightweight installation of Vim by default. So, Vim skills that are learned for Linux might also prove useful elsewhere.

Get Started with Vim

You can install the Vim editor in Red Hat Enterprise Linux by using either of two packages. These two packages provide different features and Vim commands for editing text-based files.

With the `vim-minimal` package, you might install the `vi` editor with core features. This lightweight installation includes only the core features and the basic `vi` command. You can open a file for editing by using the `vi` command:

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

Alternatively, you can use the `vim-enhanced` package to install the Vim editor. This package provides a more comprehensive set of features, an online help system, and a tutorial program. Use the `vim` command to start Vim in this enhanced mode:

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

The core features of the Vim editor are available in both commands.

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

If `vim-enhanced` is installed and a regular user wants to use the `vi` command, then they might have to use the `\vi` command to override the alias temporarily. You can use `\vi --version` and `vim --version` to compare the feature sets of the two commands.

Vim Operating Modes

The Vim editor offers various modes of operation such as *command mode*, *extended command mode*, *edit mode*, and *visual mode*. As a Vim user, always verify the current mode, because the effect of keystrokes varies between modes.

Figure 5.10: 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 modification. Pressing the required keystroke accesses specific editing functions.

- An **i** keystroke enters *insert mode*, where all typed text becomes file content. Pressing **Esc** returns to command mode.
- A **v** keystroke enters *visual mode*, where multiple characters might be selected for text manipulation. Use **Shift**+**V** for multiline and **Ctrl**+**V** for block selection. To exit the visual mode, use the **v**, **Shift**+**V**, or **Ctrl**+**V** keystrokes.
- 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 unsure which mode Vim is using, then press **Esc** a few times to get back into command mode. It is safe to press the **Esc** key in command mode repeatedly.

The Minimum, Basic Vim Workflow

Vim has efficient, coordinated keystrokes for advanced editing tasks. Although considered beneficial with practice, the capabilities of Vim can overwhelm new users.

Red Hat recommends that you learn the following Vim keys and commands:

- The **u** key undoes the most recent edit.
- The **x** key deletes 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, and discards all file changes since the last write.

Learning these commands helps a Vim user to accomplish any editing task.

Rearrange Existing Text

In Vim, you can *yank* and *put* (copy and paste), by using the **y** and **p** command characters. Position the cursor on the first character to select, 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 useful to highlight and manipulate text in different lines and columns. You can enter various visual modes in Vim by using the following key combinations.

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

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

Visual block mode is perfect for manipulating data files. Press the **Ctrl**+**v** keystroke to enter the visual block from the cursor. `VISUAL BLOCK` appears at the bottom of the screen. Use the arrow keys to highlight the section to change.

Note

First, take the time to familiarize yourself with the basic Vim capabilities. Then, expand your Vim vocabulary by learning more Vim keystrokes.

The exercise for this section uses the `vimtutor` command. This tutorial, from the `vim-enhanced` package, is an excellent way to learn the core Vim functions.

Vim Configuration Files

The `/etc/vimrc` and `~/.vimrc` configuration files alter the behavior of the `vim` editor for the entire system or for a specific user respectively. Within these configuration files, you can specify behavior such as the default tab spacing, syntax highlighting, color schemes, and more. Modifying the behavior of the `vim` editor is particularly useful when working with languages such as YAML, which have strict syntax requirements. Consider the following `~/.vimrc` file, which sets the default tab stop (denoted by the `ts` characters) to two spaces while editing YAML files. The file also includes the `set number` parameter to display line numbers while editing all files.

```
[user@host ~]$ cat ~/.vimrc
```

```
autocmd FileType yaml setlocal ts=2
set number
```

A complete list of `vimrc` configuration options is available in the references.

References

`vim`(1) man page

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

Vim Reference Manual: Vim Options

Previous Next

Guided Exercise: Edit Text Files from the Shell Prompt

In this exercise, you use the `vimtutor` command to practice basic editing techniques in the `vim` editor.

Outcomes

- Edit files with Vim.
- Gain competency in Vim by using the `vimtutor` command.

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares your environment and ensures that all required resources are available.

```
[student@workstation ~]$ lab start edit-editfile
```

Instructions

1. Use the `ssh` command to log in to the `servera` machine as the `student` user.

```
2. [student@workstation ~]$ ssh student@servera
```

```
3. ...output omitted...
```

```
[student@servera ~]$
```

4. Run the `vimtutor` command. Read the Welcome screen and perform *Lesson 1.1*.

   In the presentation, keyboard arrow keys help to navigate the window. Initially, when the `vi` editor was developed, users could not rely on having arrow keys or working keyboard mappings for arrow keys to move the cursor. Therefore, the `vi` editor was initially designed to move the cursor by using commands with standard character keys, such as the conveniently grouped **h**, **j**, **k**, and **l**.

   Here is a way to remember them:

   `hang` **back**, `jump` **down**, `kick` **up**, `leap` **forward**.

```
[student@servera ~]$ vimtutor
```

5. In the `vimtutor` window, perform *Lesson 1.2*.

   This lesson teaches how to quit without keeping unwanted changes. All changes are lost. Sometimes it is preferable to lose changes than to leave a critical file in an incorrect state.

6. In the `vimtutor` window, perform *Lesson 1.3*.

   Vim has fast, efficient keystrokes to delete an exact number of words, lines, sentences, or paragraphs. Any editing is possible with the `x` key for single-character deletion.

7. In the `vimtutor` window, perform *Lesson 1.4*.

   For most editing tasks, the `i` (insert) key is pressed first.

8. In the `vimtutor` window, perform *Lesson 1.5*.

   The previous lecture taught only the `i` command to enter edit mode. This lesson demonstrates other available keystrokes to change the cursor placement in insert mode. In insert mode, all typed text changes the file content.

9. In the `vimtutor` window, perform *Lesson 1.6*.

   Type `:wq` to save the file and quit the editor.

10. In the `vimtutor` window, read the *Lesson 1 Summary*.

    The `vimtutor` command includes six more multistep lessons. These lessons are not assigned as part of this course, but feel free to explore them to learn more.

11. Return to the `workstation` system as the `student` user.

```
[student@servera ~]$ exit
logout
Connection to servera closed.
```

```
[student@workstation ~]$
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish edit-editfile
```

This concludes the section.

Previous Next

Change the Shell Environment

Objectives

Set shell variables to run commands, and edit Bash startup scripts to set shell and environment variables to modify the behavior of the shell and programs that are run from the shell.

Shell Variable Usage

With the Bash shell, you can set *shell variables* to help to run commands or to modify the behavior of the shell. You can also export shell variables as environment variables, which are automatically copied to programs that are run from that shell. You can use variables for ease of running a command with a long argument, or to apply a common setting to commands that are run from that shell.

Shell variables are unique to a particular shell session. If you have two terminal windows open, or two independent login sessions to the same remote server, then you are running two shells. Each shell has its own set of values for its shell variables.

Assign Values to Variables

Assign a value to a shell variable with the following syntax:

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

Variable names can contain uppercase or lowercase letters, digits, and the underscore character (_). For example, the following commands set shell variables:

```
[user@host ~]$ COUNT=40
[user@host ~]$ first_name=John
[user@host ~]$ file1=/tmp/abc
[user@host ~]$ _ID=RH123
```

Remember, this change affects only the shell that you run the command in, not any other shells that you might be running on that server.

You can use the `set` command to list all shell variables that are currently set. (It also lists all shell functions, which you can ignore.) To improve readability, you can pipe the output to the `less` command so that you can view it one page at a time.

```
[user@host ~]$ set | less
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote\
:force_fignore:globasciiranges:histappend:interactive_comments:login_shell:progc\
omp:promptvars:sourcepath
BASHRCSOURCED=Y
```

```
...output omitted...
```

Retrieve Values with Variable Expansion

You can use *variable expansion* to refer to the value of a variable that you set. To use variable expansion, precede the name of the variable with a dollar sign (`$`). In the following examples, the variable expansion occurs first and then the `echo` command prints the rest of the command line that is entered.

For example, the following command sets the variable `COUNT` to `40`.

```
[user@host ~]$ COUNT=40
```

If you enter the `echo COUNT` command, then it prints the `COUNT` string.

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

If you enter instead the `echo $COUNT` command, then it prints the value of the `COUNT` variable.

```
[user@host ~]$ echo $COUNT
40
```

You can also use a variable to refer to a long file name for multiple commands.

```
[user@host ~]$ file1=/tmp/tmp.z9pXW0HqcC
[user@host ~]$ ls -l $file1
-rw-------. 1 student student 1452 Jan 22 14:39 /tmp/tmp.z9pXW0HqcC
[user@host ~]$ rm $file1
[user@host ~]$ ls -l $file1
total 0
```

Important

You can always use curly braces in variable expansion, although they are often unnecessary.

In the following example, the `echo` command tries to expand the nonexistent variable `COUNTx`, but returns nothing. The command does not report any errors either.

```
[user@host ~]$ echo Repeat $COUNTx
Repeat
```

If any trailing characters are next to a variable name, then delimit the variable name with curly braces. In the following example, the `echo` command now expands the `COUNT` variable.

```
[user@host ~]$ echo Repeat ${COUNT}x
Repeat 40x
```

Configure Bash with Shell Variables

Some shell variables are set when Bash starts. You can modify them to adjust the shell's behavior.

For example, the `HISTFILE`, `HISTFILESIZE`, and `HISTTIMEFORMAT` shell variables affect the shell history and the `history` command. The `HISTFILE` variable specifies which file to save the shell history to, and defaults to the `~/.bash_history` file.
The `HISTFILESIZE` variable specifies how many commands to save in that file from the history. The `HISTTIMEFORMAT` variable defines the time stamp format for every command in the history. This variable does not exist by default.

```
[user@host ~]$ history
...output omitted...
    6  ls /etc
    7  uptime
    8  ls -l
    9  date
   10  history
[user@host ~]$ HISTTIMEFORMAT="%F %T "
[user@host ~]$ history
...output omitted...
    6  2022-05-03 04:58:11 ls /etc
```

```
    7  2022-05-03 04:58:13 uptime
    8  2022-05-03 04:58:15 ls -l
    9  2022-05-03 04:58:16 date
   10  2022-05-03 04:58:18 history
   11  2022-05-03 04:59:10 HISTTIMEFORMAT="%F %T "
   12  2022-05-03 04:59:12 history
```

Another example is the PS1 variable, which controls the appearance of the shell prompt. If you change this value, then it changes the appearance of your shell prompt. Various special character expansions that the prompt supports are listed in the "PROMPTING" section of the bash(1) man page.

```
[user@host ~]$ PS1="bash\$ "
bash$ PS1="[\u@\h \W]\$ "
[user@host ~]$
```

Because the value that the PS1 variable sets is a prompt, Red Hat recommends ending the prompt with a trailing space. Also, whenever a variable value contains some form of space, including a space, a tab, or a return, the value must be enclosed in either single or double quotation marks. Unexpected results might occur if the quotation marks are omitted. The previous PS1 variable conforms to both the trailing space recommendation and the quotation marks rule.

Configure Programs with Environment Variables

The shell provides an *environment* for the programs that you run from that shell. Among other items, this environment includes information about the current working directory on the file system, the command-line options that are passed to the program, and the values of *environment variables*. The programs might use these environment variables to change their behavior or their default settings.

If a shell variable is not an environment variable, then only the shell can use it. However, if a shell variable is an environment variable, then the shell and any programs that run from that shell can use the variable.

Note

The HISTFILE, HISTFILESIZE, and PS1 variables from the previous section do not need to be exported as environment variables, because only the shell itself uses them, not the programs that you run from the shell.

You can assign any variable that is defined in the shell as an environment variable by marking it for export with the export command.

```
[user@host ~]$ EDITOR=vim
[user@host ~]$ export EDITOR
```

You can set and export a variable in one step:

```
[user@host ~]$ export EDITOR=vim
```

Applications and sessions use these variables to determine their behavior. For example, the shell automatically sets the HOME variable to the file name of the user's home directory when it starts. You can use this variable to help programs to determine where to save files.

Another example is the LANG variable, which sets the locale encoding. This variable adjusts the preferred language for program output; the character set; the formatting of dates, numbers, and currency; and the sort order for programs. If it is set to en_US.UTF-8, then the locale uses US English with UTF-8 Unicode character encoding. If it is set, for example, to fr_FR.UTF-8, then it uses French UTF-8 Unicode encoding.

```
[user@host ~]$ date
Tue Jan 22 16:37:45 CST 2019
[user@host ~]$ export LANG=fr_FR.UTF-8
[user@host ~]$ date
mar. janv. 22 16:38:14 CST 2019
```

Another important environment variable is PATH. The PATH variable contains a list of colon-separated directories that contain programs:

```
[user@host ~]$ echo $PATH
/home/user/.local/bin:/home/user/bin:/usr/share/Modules/bin:/usr/local/bin:/usr/bin:/
usr/local/sbin:/usr/sbin
```

When you run a command such as the `ls` command, the shell looks for the `ls` executable file in each of those directories in order, and runs the first matching file that it finds. (On a typical system, this file is `/usr/bin/ls`.)

You can append directories to your `PATH` variable. For example, perhaps you want to run some executable programs or scripts such as regular commands in the `/home/user/sbin` directory. You can append the `/home/user/sbin` directory to your `PATH` for the current session as follows:

```
[user@host ~]$ export PATH=${PATH}:/home/user/sbin
```

To list all the environment variables for a shell, run the `env` command:

```
[user@host ~]$ env
...output omitted...
LANG=en_US.UTF-8
HISTCONTROL=ignoredups
HOSTNAME=host.example.com
XDG_SESSION_ID=4
...output omitted...
```

Set the Default Text Editor

The `EDITOR` environment variable specifies your default text editor for command-line programs. Many programs use the `vi` or `vim` editor if it is not specified, and you can override this preference:

```
[user@host ~]$ export EDITOR=nano
```

Important

By convention, environment variables and shell variables that are automatically set by the shell have names with all uppercase characters. If you are setting your own variables, then you might want to use names with lowercase characters to prevent naming collisions.

Set Variables Automatically

When Bash starts, several text files run with shell commands that initialize the shell environment. To set shell or environment variables automatically when your shell starts, you can edit these Bash startup scripts.

The exact scripts that run depend on whether the shell is interactive or non-interactive, and a login or non-login shell. A user directly enters commands into an interactive shell, whereas a non-interactive shell, such as a script, runs in the background without user intervention. A login shell is invoked when a user logs in locally via the terminal or remotely via the SSH protocol. A non-login shell is invoked from an existing session, such as to open a terminal from the GNOME GUI.

For interactive login shells, the `/etc/profile` and `~/.bash_profile` files configure the Bash environment. The `/etc/profile` and `~/.bash_profile` files also source the `/etc/bashrc` and `~/.bashrc` files respectively. For interactive non-login shells, only the `/etc/bashrc` and `~/.bashrc` files configure the Bash environment. Whereas the `/etc/profile` and `/etc/bashrc` files apply to the whole system, the `~/.bash_profile` and `~/.bashrc` files are user-specific. Non-interactive shells invoke any files that the `BASH_ENV` variable defines. This variable is not defined by default.

To create a variable that is available to all of your interactive shells, edit the `~/.bashrc` file. To apply a variable only once after the user logs in, define it in the `~/.bash_profile` file.

For example, to change the default editor when you log in via SSH, you can modify the `EDITOR` variable in your `~/.bash_profile` file:

```
# .bash_profile


# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi


# User specific environment and startup programs
export EDITOR=nano
```

Note

The best way to adjust settings that affect all user accounts is to add a file with a `.sh` extension that contains the changes to the `/etc/profile.d` directory. To create the files in the `/etc/profile.d` directory, log in as the `root` user.

Bash Aliases

Bash aliases are shortcuts to other Bash commands. For example, if you must often type a long command, then you can create a shorter alias to invoke it. You use the `alias` command to create aliases. Consider the following example, which creates a `hello` alias for an `echo` command.

```
[user@host ~]$ alias hello='echo "Hello, this is a long string."'
```

You can then run the `hello` command and it invokes the `echo` command.

```
[user@host ~]$ hello
Hello, this is a long string.
```

Add aliases to a user's `~/.bashrc` file so they are available in any interactive shell.

Unset and Unexport Variables and Aliases

To unset and unexport a variable, use the `unset` command:

```
[user@host ~]$ echo $file1
/tmp/tmp.z9pXW0HqcC
[user@host ~]$ unset file1
[user@host ~]$ echo $file1


[user@host ~]$
```

To unexport a variable without unsetting it, use the `export -n` command:

```
[user@host ~]$ export -n PS1
```

To unset an alias, use the `unalias` command:

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

References

`bash`(1), `env`(1), and `builtins`(1) man pages

Previous Next

Guided Exercise: Change the Shell Environment

In this exercise, you use shell variables and variable expansion to run commands, and set an environment variable to adjust the default editor for new shells.

Outcomes

- Edit a user profile.
- Create a shell variable.
- Create an environment variable.

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command ensures that all required resources are available.

```
[student@workstation ~]$ lab start edit-bashconfig
```

Instructions

1. Change the `student` user's `PS1` shell variable to `[\u@\h \t \w]$` (remember to put the value of `PS1` in quotation marks and to include a trailing space after the dollar sign). This change adds the time to the prompt.
    1. Use the `ssh` command to log in to `servera` as the `student` user.

```
2. [student@workstation ~]$ ssh student@servera
3. ...output omitted...
```

```
[student@servera ~]$
```

    4. Use Vim to edit the `~/.bashrc` configuration file.

```
[student@servera ~]$ vim ~/.bashrc
```

5. Add the `PS1` shell variable and its value to the `~/.bashrc` file. Set the value of the shell variable, including a trailing space at the end, inside quotation marks.

```
...output omitted...
export PATH
```

```
PS1='[\u@\h \t \w]$ '
```

8. Exit from `servera` and log in again by using the `ssh` command to update the command prompt, or execute the `~/.bashrc` file by using the `source ~/.bashrc` command.

```
[student@servera ~]$ exit
logout
Connection to servera closed.
[student@workstation ~]$ ssh student@servera
...output omitted...
```

```
[student@servera 14:45:05 ~]$
```

2. Assign a value to a local shell variable. Variable names can contain uppercase or lowercase letters, digits, and the underscore character. Retrieve the variable value.
    1. Create a variable called `file` with a value of `tmp.zdkei083`. The `tmp.zdkei083` file exists in the `student` home directory.

```
[student@servera 14:47:05 ~]$ file=tmp.zdkei083
```

    2. Retrieve the value of the `file` variable.

```
[student@servera 14:48:35 ~]$ echo $file
```

```
tmp.zdkei083
```

    4. Use the `file` variable name and the `ls -l` command to list the `tmp.zdkei083` file. Use the `rm` command and the `file` variable name to delete the `tmp.zdkei083` file. Verify that the file is deleted.

```
[student@servera 14:59:07 ~]$ ls -l $file
-rw-r--r--. 1 student student 0 Jan 23 14:59 tmp.zdkei083
[student@servera 14:59:10 ~]$ rm $file
[student@servera 14:59:15 ~]$ ls -l $file
```

```
ls: cannot access 'tmp.zdkei083': No such file or directory
```

3. Assign a value to the `EDITOR` variable. Use one command to assign the variable as an environment variable.

```
[student@servera 14:46:40 ~]$ export EDITOR=vim
[student@servera 14:46:55 ~]$ echo $EDITOR
```

```
vim
```

6. Return to the `workstation` system as the `student` user.

```
[student@servera 14:47:11 ~]$ exit
logout
Connection to servera closed.
```

```
[student@workstation ~]$
```

Finish

On the `workstation` machine, change to the `student` user home directory and use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish edit-bashconfig
```

This concludes the section.

Previous Next

Summary

- Running programs, or processes, have three standard communication channels: standard input, standard output, and standard error.
- You can use I/O redirection to read standard input from a file or to write the output or errors from a process to a file.
- Pipelines can connect standard output from one process to the standard input of another process, and can format output or build complex commands.
- Know how to use at least one command-line text editor, and Vim is the recommended option because it is commonly installed by default in Linux distributions.
- Shell variables can help you to run commands, and are unique to a shell session.
- You can modify the behavior of the shell or the processes with environment variables.

Previous Next