Software Engineering Tools Summary

Chapter 1:

- Which two of the following are benefits of open source software for the user?
  - Code can survive the loss of the original developer or distributor.
  - You can learn from real-world code and develop more effective applications.
- Which two of the following are ways in which Red Hat develops their products for the future and interacts with the community?
  - Sponsor and integrate open source projects into the community-driven Fedora project.
  - Participate in upstream projects.
- Which two statements describe the benefits of Linux?
  - Linux is modular and can be configured as a full graphical desktop or a small appliance.
  - Linux includes a powerful and scriptable command-line interface, enabling easier automation and provisioning.

Chapter 2:

- The Linux command line is provided by a program called the **shell**.
- When a shell is used interactively, it displays a string when it is waiting for a command from the user. This is called the **shell prompt**.
- When a regular user starts a shell, the default prompt ends with a **$** character, and is replaced by a **#** character if the shell is running as the superuser "root".
- Commands entered at the shell prompt have three basic parts:
  - **Command** to run. "ls"
  - **Options** to adjust the behavior of the command. "-all"
  - **Arguments**, which are typically targets of the command. "fci/sw"
- Each virtual console supports an independent login session. You can switch between them by pressing **Ctrl+Alt** and a function key (**F1** through **F6**) at the same time.
- In Linux, the most common way to get a shell prompt on a remote system is to use Secure Shell (**SSH**).
- The **ssh** command encrypts the connection to secure the communication against eavesdropping or hijacking of the passwords and content.
- **Shell** is the interpreter that executes commands typed as strings.
- **Prompt** is the visual cue that indicates an interactive shell is waiting for the user to type a command.
- **Command** is the name of the program to run.
- **Option** is the part of the command line that adjusts the behavior of a command.

- **Argument** is the part of the command line that specifies the target that the command should operate on.
- **Physical console** is the hardware display and keyboard used to interact with a system.
- **Virtual console** is one of multiple logical consoles that can each support an independent login session.
- **Terminal** is an interface that provides a display for output and a keyboard for input to a shell session.
- Introduction to the GNOME Desktop Environment:
    - Top bar
    - Activities overview
    - System menu
    - Dash
    - Windows overview
    - Workspace selector
    - Message tray
- Workspaces are separate desktop screens that have different application windows. These can be used to organize the working environment by grouping open application windows by task.
- There are two simple methods for switching between workspaces. One method, perhaps the fastest, is to press **Ctrl+Alt+UpArrow** or **Ctrl+Alt+DownArrow** to switch between workspaces sequentially. The second is to switch to the Activities overview and click the desired workspace.
- Starting a terminal:
    - From the **Activities** overview, select Terminal from the **dash**
    - Or press the **Alt+F2** key combination to open the Enter a Command and enter **gnome terminal**.
- 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 **(User) → Log Out**. A window displays that offers the option to **Cancel** or confirm the **Log Out** action.
- To shut down the system, from the system menu in the upper-right corner, click the power button at the bottom of the menu or press **Ctrl+Alt+Del**. In the dialog box that displays, you can choose to **Power Off** or **Restart** the machine or **Cancel** the operation. If you do not make a choice, the system automatically shuts down after 60 seconds.
- Commands are the names of **programs** that are installed on the system. Each command has its own options and arguments.
- If you want to type more than one command on a single line, use the semicolon (**;**) as a command separator.

- Examples of simple commands:
    - date: (+) sign argument specifies a format string.
        - **date**
        - **date +%R, +%x, +%r, +%W, +%d**
    - **whoami**
    - **passwd**
    - **file filepath** → displays what type it is.
- Viewing the contents of files:
    - **cat filepath** → view the contents of the file.
    - **cat file1 file2** → display the contents of multiple files.
    - **less file** → The less command allows you to page forward and backward through files that are longer than can fit on one terminal window. Use the **UpArrow** key and the **DownArrow** key to scroll up and down. Press **q** to exit the command.
    - **head file** or **tail file** → commands display the beginning and end of a file, respectively. By default, these commands display 10 lines of the file, but they both have a -**n** option that allows a different number of lines to be specified. The file to display is passed as an argument to these commands.
    - **wc file** → The wc command counts lines, words, and characters in a file. It takes a -l, -w, or -c option to display only the number of lines, words, or characters, respectively.
- Tab completion allows a user to quickly complete commands or file names after they have typed enough at the prompt to make it unique. If the characters typed are not unique, pressing the **Tab** key twice displays all commands that begin with the characters already typed.
- 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, to make command readability easier, you can type a long command using more than one line, to do this, you will use a backslash character (****).
- The **history** command displays a list of previously executed commands prefixed with a command number.
    - The exclamation point character (**!**) is a metacharacter that is used to expand previous commands without having to retype them. The **!number** command expands to the command matching the number specified. The **!string** command expands to the most recent command that begins with the string specified.
- Esc+ (the keys **Esc** and **.** pressed at the same time) copies the last argument of previous commands.
- !! command repeats the previous command exactly.

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.

Chapter 3:

- All files on a Linux system are stored on file systems, which are organized into a single inverted tree of directories, known as a **file-system hierarchy**. This tree is inverted because the root of the tree is said to be at the top of the hierarchy, and the branches of directories and subdirectories stretch below the root.
- The / directory is the **root directory** at the top of the file-system hierarchy. The / character is also used as a directory separator in file names. For example, if etc is a subdirectory of the / directory, you could refer to that directory as /etc. Likewise, if the /etc directory contained a file named issue, you could refer to that file as /etc/issue.

Location	Purpose
/usr	**Installed software**, shared libraries, include files, and read-only program data. Important subdirectories include: • **/usr/bin: User commands**. • /usr/sbin: System administration commands.
/etc	Configuration files specific to this system. (persistent data)
/var	Variable data specific to this system that should **persist** between boots. Files that dynamically change, such as databases, cache directories, log files, printer-spooled documents, and website content may be found under /var.
/run	Runtime data for processes started since the last boot.
/home	*Home directories* are where regular **users store their personal data** and configuration files.
/root	Home directory for the administrative superuser, root.
/tmp	A world-writable space for **temporary files**. Files which have not been accessed, changed, or modified for 10 days are deleted from this directory automatically.

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

- An **absolute path** is a fully qualified name, specifying the files exact location in the file system hierarchy. It begins at the **root** (/) **directory** and specifies each subdirectory that must be traversed to reach the specific file. Every file in a file system has a unique absolute path name, recognized with a simple rule: A path name with a forward slash (/) as the first character is an absolute path name.
- Like an absolute path, a **relative path** identifies a unique file, specifying **only** the path necessary to reach the file from the working directory. Recognizing relative path names follows a simple rule: A path name with anything other than a forward slash as the first character is a relative path name.

Command	**Purpose**
pwd	Displays the full path name of the current working directory.
cd **cd ~**	Change your shell's current working directory. If you do not specify any arguments to the command, it will change to your home directory.
ls	Lists directory contents for the specified directory or, if no directory is given, for the current working directory. -l (long listing format), -a (all files, including *hidden files*), -R (recursive, to include the contents of all subdirectories).
touch	Normally updates a file's timestamp to the current date and time without otherwise modifying it.
cd .	Refer to the **current** directory.
cd ..	Refer to the **parent** directory.
cd ../..	changes the working directory up **two** levels from the current location.
cd -	Changes to the previous directory.

- Linux file systems are **case-sensitive**, creating FileCase.txt and filecase.txt in the same directory results in two unique files.
- File names beginning with a dot (**.**) indicate hidden files; you cannot see them in the normal view using **ls** and other commands.

Activity	**Command Syntax**
Create a directory	**mkdir *directory***
Creates missing parent directories for the requested destination.	**mkdir -p *directory***
Copy a file	**cp *file new-file***
Copy a directory and its contents	**cp -r *directory new-directory***
Move or rename a file or directory (cut)	**mv *file new-file***
Remove a file	**rm *file***
Remove a directory containing files	**rm -r *directory / rm --recursive***
Remove file with interactive prompt for confirmation	**rm -ri**
Remove file without interactive prompt for confirmation	**rm -f**
Remove an empty directory	**rmdir *dir / rm -d dir***

- Hard Links and Soft Links (like c++ pointers):
    - It is possible to create multiple names that point to the same file. There are two ways to do this: by creating a **hard link** to the file, or by creating a **soft link** (sometimes called a symbolic link) to the file.
- Creating hard link:
    - Every file starts with a single hard link, from its initial name to the data on the file system. When you create a new 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. Once created, you cannot tell the difference between the new hard link and the original name of the file.
    - You can use the **ln** command to create a new hard link (another 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.
        - `ln newfile.txt /tmp/newfile-hlink2.txt`
    - If you want to find out whether two files are hard links of each other, one way is to use the **-i** option with the **ls** command to list the files' *inode* number. If the files are on the same file system (discussed in a moment) and their *inode* numbers are the same, the files are hard links pointing to the same data.
    - All hard links that reference the same file will have the same link count, access permissions, user and group ownerships, time stamps, and file content.
    - Even if the original file gets deleted, the contents of the file are still available if at least one hard link exists. Data is only deleted from storage when the last hard link is deleted.
    - If you delete a hard link and then use normal tools (rather than ln) to create a new file with the same name, the new file will not be linked to the old file.
    - A hard link points a name to data on a **storage device**.
- Hard links can only be used with regular files. You cannot use **ln** to create a hard link to a directory or special file, also hard links can only be used if both files are on the same file system.
- Creating soft link:
    - Soft links have some advantages over hard links:
        - They can link two files on different file systems.
        - They can point to a directory or special file, not just a regular file.
    - The **ln -s** command is used to create a new soft link.
    - When the original regular file gets deleted, the soft link will still point to the file, but the target is gone. A soft link pointing to a missing file is called a "dangling soft link."
    - One side-effect of the dangling soft link in the preceding example is that if you later create a new file with the same name as the deleted file, the soft link will no longer be "dangling" and will point to the new file.

- A soft link points a name to **another name**, that points to data on a storage device.

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

- Tilde Expansion:
  - The tilde character (~) matches the current user's home directory. If it starts a string of characters other than a slash (/), the shell will interpret the string up to that slash as a user name, if one matches, and replace the string with the absolute path to that user's home directory. If no username matches, then an actual tilde followed by the string of characters will be used instead.
- Brace Expansion:
  - Brace expansion is used to generate discretionary strings of characters. Braces contain a comma-separated list of strings, or a sequence expression. The result includes the text preceding or following the brace definition. Brace expansions may be nested, one inside another. Also, double-dot syntax (..) expands to a sequence such that {m..p} will expand to m n o p.
- Variable Expansion:
  - A variable act like a named container that can store a value in memory. Variables make it easy to access and modify the stored data either from the command line or within a shell script.
  - You can use variable expansion to convert the variable name to its value on the command line. If a string starts with a dollar sign ($), then the shell will try to use the rest of that string as a variable name and replace it with whatever value the variable has.

- Command Substitution:
    - Command substitution allows the output of a command to replace the command itself on the command line. Command substitution occurs when a command is enclosed in parentheses and preceded by a dollar sign (**$**). The **$(command)** form can nest multiple command expansions inside each other.
- Protecting Arguments from Expansion:
    - Many characters have special meaning in the Bash shell. To keep the shell from performing shell expansions on parts of your command line, you can quote and escape characters and strings.
    - The backslash (****) is an **escape character** in the Bash shell. It will protect the character immediately following it from expansion.
    - To protect longer character strings, single quotes (') or double quotes (") are used to enclose strings. They have slightly different effects. Single quotes stop all shell expansion. Double quotes stop most shell expansion.

Chapter 4:

- One source of documentation that is generally available on the local system are system manual pages or man pages These pages are shipped as part of the software packages for which they provide documentation, and can be accessed from the command line by using the **man** command.
- To read specific man pages, use **man** topic. Contents are displayed 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, include the section number argument: **man 5 passwd** displays passwd(5).
- Each topic is separated into several parts. Most topics share the same headings and are presented in the same order. Typically, a topic does not feature all headings, because not all headings apply for all topics.
- A keyword search of man pages is performed with man -k keyword, which displays a list of keyword-matching man page topics with section numbers.
- Man pages have a format useful as a command reference, but less useful as general documentation. For such documents, the GNU Project developed a different online documentation system, known as GNU Info.
- Man pages have a much more formal format, and typically document a specific command or function from a software package, and are structured as individual text files. Info documents typically cover particular software packages as a whole, tend to have more practical examples of how to use the software, and are structured as hypertext documents.

- Info documentation is comprehensive and hyperlinked. It is possible to output info pages to multiple formats. By contrast, man pages are optimized for printed output. The Info format is more flexible than man pages, allowing thorough discussion of complex commands and concepts. Like man pages, Info nodes are read from the command line, using the pinfo command.
- A typical man page has a small amount of content focusing on one particular topic, command, tool, or file. The Info documentation is a comprehensive document. Info provides the following improvements:
    - One single document for a large system containing all the necessary information for that system
    - Hyperlinks
    - A complete browsable document index
    - A full text search of the entire document
- Some commands and utilities have both man pages and info documentation; usually, the Info documentation is more in depth.

Navigation	pinfo	man
Scroll forward (down) one screen	**PageDown or Space**	**PageDown or Space**
Scroll backward (up) one screen	**PageUp or b**	**PageUp or b**
Display the directory of topics	**D**	**-**
Scroll forward (down) one half-screen	**-**	**D**
Display the parent node of a topic	**U**	**-**
Display the top (up) of a topic	**HOME**	**G**
Scroll backward (up) one half-screen	**-**	**U**
Scroll forward (down) to next hyperlink	**DownArrow**	**-**
Open topic at cursor location	**Enter**	**-**
Scroll forward (down) one line	**-**	**DownArrow or Enter**
Scroll backward (up) to previous hyperlink	**UpArrow**	**-**
Scroll backward (up) one line	**-**	**UpArrow**
Search for a pattern	**/string**	**/string**
Display next node (chapter) in topic	**N**	**-**
Repeat previous search forward (down)	**/ then Enter**	**n**
Display previous node (chapter) in topic	**P**	**-**
Repeat previous search backward (up)	**-**	**ShiftN**
Quit the program	**Q**	**Q**

Chapter 5:

Channel name	Description	Default connection	Usage
stdin	Standard input	Keyboard	read only
stdout	Standard output	Terminal	write only
stderr	Standard error	Terminal	write only
filename	Other files	none	read and/or write

Usage	Explanation
> file	redirect stdout to overwrite a file
>> file	redirect stdout to append to a file
2> file	redirect stderr to overwrite a file
2> /dev/null	discard stderr error messages by redirecting to /dev/null
> file 2>&1	redirect stdout and stderr to overwrite the same file
&> file	
>> file 2>&1	redirect stdout and stderr to append to the same file
&>> file	

- A pipeline is a sequence of one or more commands separated by the pipe character (**|**). A pipe connects the standard output of the first command to the standard input of the next command.
- One useful mental image is to imagine that data is "flowing" through the pipeline from one process to another, being altered slightly by each command in the pipeline through which it flows.
- When redirection is combined with a pipeline, the shell sets up the entire pipeline first, then it redirects input/output. If output redirection is used in the middle of a pipeline, the output will go to the file and not to the next command in the pipeline.
- In a pipeline, **tee** command copies its standard input to its standard output and also redirects its standard output to the files named as arguments to the command. If you imagine data as water flowing through a pipeline, **tee** can be visualized as a "T" joint in the pipe which directs output in two directions.
- Standard error can be redirected through a pipe, but the merging redirection operators (&> and &>>) cannot be used to do this.
- An unusual characteristic of Vim is that it has several modes of operation, including *command mode, extended command mode, edit mode, and visual mode*. Depending on the mode, you may be issuing commands, editing text, or working with blocks of text. As a new Vim user, you should always be aware of your current mode as keystrokes have different effects in different modes.

- When you first open Vim, it starts in *command mode*, which is used for navigation, cut and paste, and other text manipulation. Enter each of the other modes with single character keystrokes to access specific editing functionality:
    - An **i** keystroke enters *insert mode*, where all text typed becomes file content. Pressing **Esc** returns to *command mode*.
    - A **v** keystroke enters *visual mode*, where multiple characters may be selected for text manipulation. Use **Shift+V** for multiline and **Ctrl+V** for block selection. The same keystroke used to enter visual mode (**v**, **Shift+V** or **Ctrl+V**) is used to exit.
    - The **:** keystroke begins *extended command mode* for tasks such as writing the file (to save it) and quitting the Vim editor.
- The **i** key puts Vim into *insert mode*. All text entered after this is treated as file contents until you exit *insert mode*. The **Esc** key exits insert mode and returns Vim to *command mode*. The **u** key will undo the most recent edit. Press the **x** key to delete a single character. The **:w** command writes (saves) the file and remains in *command mode* for more editing. The **:wq** command writes (saves) the file and quits Vim. The **:q!** command quits Vim, discarding all file changes since the last write.
- In Vim, copy and paste is known as *yank and put*, using command characters **y** and **p**. Begin by positioning the cursor on the first character to be selected, and then enter visual mode. Use the arrow keys to expand the visual selection. When ready, press **y** to *yank* the selection into memory. Position the cursor at the new location, and then press **p** to *put* the selection at the cursor. Use **dd** command to cut text.
- 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, you are running two shells. Each shell has its own set of values for its shell variables.
- Variable names can contain uppercase or lowercase letters, digits, and the underscore character (_).
- You can use the **set** command to list all shell variables that are currently set.
- You can use variable expansion to refer to the value of a variable that you have set. To do this, precede the name of the variable with a dollar sign (**$**).
- If there are any trailing characters adjacent to the variable name, you might need to protect the variable name with **curly braces.**
- You can make any variable defined in the shell into an environment variable by marking it for export with the **export** command.
- To list all the environment variables for a particular shell, run the **env** command.
- Environment variables are not shared as they are a special case from the shell variables.
- To unset and unexport a variable entirely, use the **unset** command.

Chapter 6:

- There are three main types of user account: *the superuser, system users, and regular users*.
    - The *superuser* account is for administration of the system. The name of the superuser is root and the account has UID 0. The superuser has full access to the system.
    - The system has *system user* accounts which are used by processes that provide supporting services. These processes, or daemons, usually do not need to run as the superuser. They are assigned non-privileged accounts that allow them to secure their files and other resources from each other and from regular users on the system. Users do not interactively log in using a system user account.
    - Most users have *regular user* accounts which they use for their day-to-day work. Like system users, regular users have limited access to the system.
- Specific UID numbers and ranges of numbers are used for specific purposes by Red Hat Enterprise Linux.
    - **UID 0** is always assigned to the **superuser** account, root.
    - **UID 1-200** is a range of "**system users**" assigned statically to system processes by Red Hat.
    - **UID 201-999** is a range of "**system users**" used by system processes that do not own files on the file system.
    - **UID 1000+** is the range available for assignment to **regular users**.
- You can use the **id** command to show information about the currently logged-in user.
- To view the owner of a file, use the ls **-l** command. To view the owner of a directory, use the **ls -ld** command.
- To view process information, use the **ps** command.
- Each line in the **/etc/passwd** file contains information about one user.
- A group is a collection of users that need to share access to files and other system resources. Groups can be used to grant access to files to a set of users instead of just a single user.
- Each line in the **/etc/group** file contains information about one group.
- Normally, when you create a new regular user, a new group with the same name as that user is created. That group is used as the primary group for the new user, and that user is the only member of this *User Private Group*.
- Users may also have supplementary groups. Membership in supplementary groups is determined by the /etc/group file. Users are granted access to files based on whether any of their groups have access.
- Most operating systems have some sort of superuser, a user that has all power over the system. In Red Hat Enterprise Linux this is the root user. This user has the power to override normal privileges on the file system and is used to manage and administer the system. To perform tasks such as installing or removing software and

to manage system files and directories, users must escalate their privileges to the root user.

- The **su** command allows users to switch to a different user account. If you run **su** from a regular user account, you will be prompted for the password of the account to which you want to switch. When root runs **su**, you do not need to enter the user's password.
- The command **su** starts a non-login shell, while the command **su -** (with the dash option) starts a login shell. The main distinction between the two commands is that **su -** sets up the shell environment as if it were a new login as that user, while **su** just starts a shell as that user, but uses the original user's environment settings.
- One tool that can be used to get root access in this case is **sudo**, unlike **su**, **sudo** normally requires users to enter their own password for authentication, not the password of the user account they are trying to access. That is, users who use **sudo** to run commands as root do not need to know the root password. Instead, they use their own passwords to authenticate access.
- If there is a nonadministrative user account on the system that can use **sudo** to run the su command, you can run **sudo su -** from that account to get an interactive root user shell. This works because **sudo** will run **su -** as root, and root does not need to enter a password to use **su**. Another way to access the root account with **sudo** is to use the **sudo -i** command.
- The main configuration file for **sudo** is **/etc/sudoers**.
- For example, the following line from the **/etc/sudoers** file enables sudo access for members of group **wheel**.

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

- In this line, **%wheel** is the user or group to whom the rule applies. A **%** specifies that this is a group, group wheel. The **ALL=(ALL)** specifies that on any host that might have this file, wheel can run any command. The final **ALL** specifies that wheel can run those commands as any user on the system.
- Using supplementary files under the **/etc/sudoers.d** directory is convenient and simple. You can enable or disable **sudo** access simply by copying a file into the directory or removing it from the directory.
- To enable full **sudo** access for the user **user01**, you could create **/etc/sudoers.d/user01** with the following content:
    - **user01 ALL=(ALL) ALL**
- To enable full **sudo** access for the group **group01**, you could create **/etc/sudoers.d/group01** with the following content:
    - **%group01 ALL=(ALL) ALL**
- **sudo** may or may not prompt you for the student password, depending on the time-out period of **sudo**. The default time-out period is five minutes. If you have

authenticated to **sudo** within the last five minutes, **sudo** will not prompt you for the password. If it has been more than five minutes since you authenticated to **sudo**, you need to enter student as the password to get authenticated to **sudo**.

- The **useradd** *username* command creates a new user named username.

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

- When a user is removed with **userdel** without the **-r** option specified, the system will have files that are owned by an unassigned UID, the old user's UID will get reassigned to the new user, giving the new user ownership of the old user's remaining files.
- The **passwd** *username* command sets the initial password or changes the existing password of *username*.
- The **groupadd** command creates groups. Without options the **groupadd** command uses the next available GID from the range specified in the **/etc/login.defs** file while creating the groups.
- The **-g** option specifies a particular GID for the group to use.
- The **-r** option creates a system group using a GID from the range of valid system GIDs.
- The **groupmod** command changes the properties of an existing group. The **-n** option specifies a new name for the group. The **-g** option specifies a new GID.
- The **groupdel** command removes groups.
- Use the **usermod -aG** command to add a user to a supplementary group.
- At one time, encrypted passwords were stored in the world-readable **/etc/passwd** file. This was thought to be reasonably secure until dictionary attacks on encrypted passwords became common. At that point, the encrypted passwords were moved to a separate **/etc/shadow** file which is readable only by root. This new file also allowed password aging and expiration features to be implemented.

- The preceding **chage** command uses the **-m**, **-M**, **-W**, and **-I** options to set the minimum age, maximum age, warning period, and inactivity period of the user's password, respectively.
- The **chage -d 0 user03** command forces the *user03* user to update its password on the next login.
- The **chage -l user03** command displays the password aging details of *user03*.
- The **chage -E 2019-08-05 user03** command causes the *user03* user's account to expire on 2019-08-05 (in YYYY-MM-DD format).

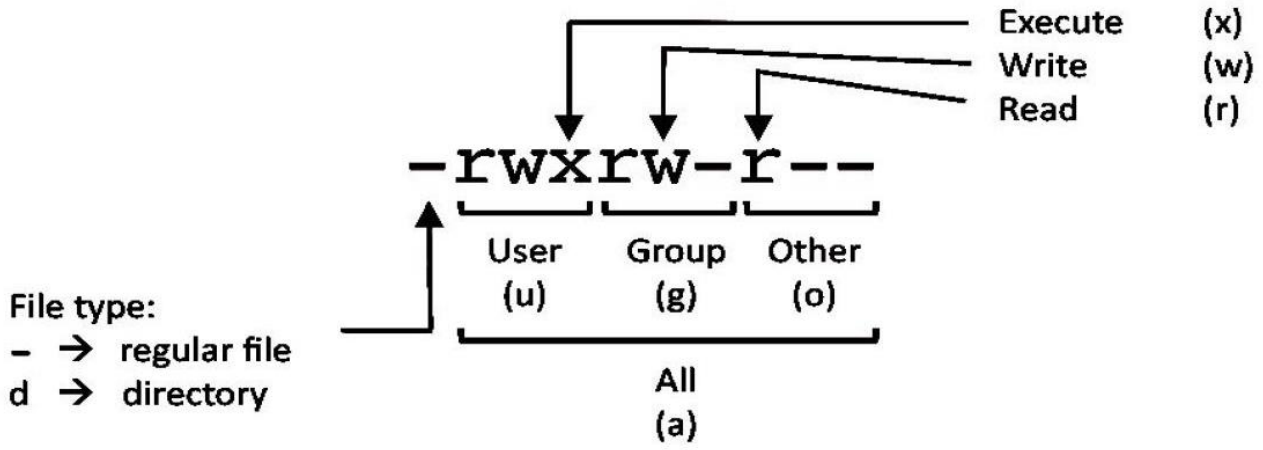
Chapter 7:

- File permissions control access to files. Files have three user categories to which permissions apply. The file is owned by a **user**, normally the one who created the file. The file is also owned by a single **group**, usually the primary group of the user who created the file, but this can be changed. Different permissions can be set for the owning user, the owning group, and for all other users on the system that are not the user or a member of the owning group.

Permission	Effect on files	Effect on directories
r (read)	File contents can be read.	Contents of the directory (the file names) can be listed. (**ls** command)
w (write)	File contents can be changed.	Any file in the directory can be created or deleted.
x (execute)	Files can be executed as commands.	The directory can become the current working directory. (You can **cd** into it, but also require read permission to list files found there.)

- Users normally have both **read** and **execute** permissions on read-only directories so that they can list the directory and have full read-only access to its contents. If a user only has **read** access on a directory, the names of the files in it can be listed, but no other information, including permissions or time stamps, are available, nor can they be accessed. If a user only has **execute** access on a directory, they cannot list file names in the directory. If they know the name of a file that they have permission to read, they can access the contents of that file from outside the directory by explicitly specifying the relative file name.
- A file may be removed by anyone who has ownership of, or write permission to, the directory in which the file resides, regardless of the ownership or permissions on the file itself. This can be overridden with a special permission, the **sticky bit**.
- The Linux root user has the equivalent of the Windows **Full Control** permission on all files.
- The **-l** option of the **ls** command shows detailed information about permissions and ownership.

- Use the **-ld** option of the **ls** command to show detailed information about a directory itself, and not its contents.
- The **-la** option of the **ls** command shows the permissions of hidden files, including the special files used to represent the directory and its parent.
- The first character of the long listing is the file type, interpreted like this:
    - **-** is a regular file.
    - **d** is a directory.
    - **l** is a soft link.
    - Other characters represent hardware devices (**b** and **c**) or other special-purpose files (**p** and **s**).




```
PERMISSION      EXAMPLE

 U   G   W
rwx rwx rwx     chmod 777 filename  # Use sparingly!
rwx rwx r-x     chmod 775 filename
rwx r-x r-x     chmod 755 filename
rw- rw- r--     chmod 664 filename
rw- r-- r--     chmod 644 filename
```

- The next nine characters are the file permissions. These are in three sets of three characters: permissions that apply to the user that owns the file, the group that owns the file, and all other users. If the set shows **rwx**, that category has all three permissions, **read**, **write**, and **execute**. If a letter has been replaced by **-**, then that category does not have that permission.
- **The group permissions take precedence. (Important example in section 7.1)**
- The command used to change permissions from the command line is **chmod**, which means "change mode" (permissions are also called the *mode of a file*). The **chmod** command takes a permission instruction followed by a list of files or directories to change.

- Changing Permissions with the **Symbolic Method**:
    - *chmod WhoWhatWhich file|directory*
        - Who is u, g, o, a (for user, group, other, all)
        - What is +, -, = (for add, remove, set exactly)
        - Which is r, w, x (for read, write, execute)
- When using **chmod** to change permissions with the symbolic method, using a capital **X** as the permission flag will add execute permission only if the file is a directory or already has execute set for user, group, or other.
- The **chmod** command supports the **-R** option to recursively set permissions on the files in an entire directory tree.
- Changing Permissions with the **Numeric Method**:
    - *chmod ### file|directory*
        - Each digit (**#**) represents permissions for an access level: user, group, other.
        - The digit is calculated by adding together numbers for each permission you want to add, **4 for read**, **2 for write**, and **1 for execute**.
        - Permissions are represented by a **3-digit octal number**. A single octal digit can represent any single value from **0-7**.
- A newly created file is owned by the **user** who creates that file. By default, new files have a group ownership that is the **primary group** of the user creating the file. In Red Hat Enterprise Linux, a user's primary group is usually a **private group** with only that user as a member. To grant access to a file based on group membership, the group that owns the file may need to be changed.
- File ownership can be changed with the **chown** (change owner) command, followed by **username** and **file name**. (for example, **chown** *student test_file*)
- The **chown** can be used with the **-R** option to recursively change the ownership of an entire directory tree. (for example, **chown -R** *student test_dir*)
- The **chown** command can also be used to change group ownership of a file by preceding the group name with a colon (**:**). (for example, **chown :***admins test_dir*)
- The **chown** command can also be used to change both owner and group at the same time by using the *owner:group* syntax. (for example, **chown** *owner:group test_dir*)

Special permission	Effect on files	Effect on directories
u+s (suid)	File executes as the user that **owns** the file, not the user that **ran** the file.	No effect.
g+s (sgid)	File executes as the group that **owns** the file.	Files newly created in the directory have their group owner set to match the group owner of the directory.
o+t (sticky)	No effect.	Users with write access to the directory can only remove files that they own; they cannot remove, or force saves to files owned by other users.

- The **setuid** permission on an executable **file** means that commands run as the user owning the file, not as the user that ran the command. In a long listing, you can identify the **setuid** permissions by a lowercase **s** where you would normally expect the **x** to be. If the owner does not have execute permissions, this is replaced by an uppercase **S**.
- The special permission **setgid** on a **directory** means that files created in the directory inherit their group ownership from the directory, rather than inheriting it from the creating user. This is commonly used on group collaborative directories to automatically change a file from the default private group to the shared group.
- If **setgid** is set on an executable **file**, commands run as the group that owns that file, not as the user that ran the command, in a similar way to **setuid** works. In a long listing, you can identify the **setgid** permissions by a **lowercase s** where you would normally expect the **x** to be. If the group does not have execute permissions, this is replaced by an **uppercase S**.
- Lastly, the **sticky bit** for a directory sets a special restriction on deletion of files. Only the **owner** of the file (and **root**) can delete files within the directory. In a long listing, you can identify the **sticky** permissions by a **lowercase t** where you would normally expect the **x** to be. If other does not have execute permissions, this is replaced by an **uppercase T**.
- Setting Special Permissions:
    - Symbolically: setuid = **u+s**; setgid = **g+s**; sticky = **o+t**
    - Numerically (fourth preceding digit): **setuid** = **4**; **setgid** = **2**; **sticky** = **1**
- Examples:
    - Add the **setgid** bit on directory:

```
[user@host ~]# chmod g+s directory
```

    - Set the **setgid** bit and add read/write/execute permissions for user and group, with no access for others, on directory:

```
[user@host ~]# chmod 2770 directory
```

- If you create a new directory, the operating system starts by assigning it octal permissions **0777** (*drwxrwxrwx*). If you create a new regular file, the operating system assigns it octal permissions **0666** (*-rw-rw-rw-*). You always must explicitly add execute permission to a regular file. This makes it harder for an attacker to compromise a network service so that it creates a new file and immediately executes it as a program.
- However, the shell session will also set a **umask** to further restrict the permissions that are initially set.

- The system's default **umask** values for Bash shell users are defined in the **/etc/profile** and **/etc/bashrc** files. Users can override the system defaults in the **.bash_profile** and **.bashrc** files in their home directories.
- By setting the **umask** value to **0**, the **file** permissions for **other** change from read to **read and write**. The **directory** permissions for **other** changes from read and execute to **read, write, and execute**.
- To **mask** all **file** and **directory** permissions for other, set the **umask** value to **007**.
- A **umask** of **027** ensures that new **files** have **read** and **write** permissions for **user** and **read** permission for **group**. New **directories** have **read** and **write** access for **group** and **no permissions** for **other**.
- The default umask for users is set by the shell startup scripts. By default, if your account's UID is 200 or more and your username and primary group name are the same, you will be assigned a umask of 002. Otherwise, your umask will be 022.

Chapter 8:

- A process is a running instance of a launched, executable program. Consists of:
  - An **address** space of allocated memory.
  - **Security properties** including ownership credentials and privileges.
  - One or more execution **threads** of program code.
  - **Process state**.
- The environment of a process includes:
  - **Local** and **global variables**.
  - A current **scheduling context**.
  - **Allocated system resources**, such as file descriptors and network ports.
- An existing (**parent**) process duplicates its own address space (**fork**) to create a new (**child**) process structure. Every new process is assigned a unique process ID (**PID**) for tracking and security. The **PID** and the parent's process ID (**PPID**) are elements of the new process environment. Any process may create a child process. All processes are descendants of the first system process, **systemd**.
- A child process may then **exec** its own program code. Normally, a parent process **sleeps** while the child process runs, setting a request (**wait**) to be signaled when the child completes. Upon **exit**, the child process has already closed or discarded its resources and environment. The only remaining resource, called a **zombie**, is an entry in the process table. The parent, signaled **awake** when the child exited, cleans the process table of the child's entry, thus freeing the last resource of the child process. The parent process then continues with its own program code execution.

Name	**Kernel-defined state name and description**
Running	(R) The process is either executing on a CPU or waiting to run. Process can be executing user routines or kernel routines (system calls) or be queued and ready when in the Running (or Runnable) state.
Sleeping	(S) The process is waiting for some condition: a hardware request, system resource access, or signal.
	(D) Used only when process interruption may cause an unpredictable device state.
	(K) Identical to the uninterruptible D state but modified to allow a waiting task to respond to the signal that it should be killed (exit completely).
	(I) The kernel does not count these processes when calculating load average. Used for kernel threads.
Stopped	(T) The process has been stopped (suspended), usually by being signaled by a user or another process.
	(T) A process that is being debugged is also temporarily Stopped
Zombie	(Z) A child process signals its parent as it exits. All resources except for the process identity (**PID**) are released.
	(X) When the parent cleans up (reaps) the remaining child process structure, the process is now released completely.

- The **S** column of the **top** command or the **STAT** column of the **ps aux** command show the state of each process.
- On a single CPU system, only **one** process can run at a time. It is possible to see several processes with a state of **R**. However, not all of them will be running consecutively, some of them will be in status waiting.
- The **ps** command is used for listing current processes. It can provide detailed process information.
- Perhaps the most common set of options, **aux**, displays all processes including processes without a controlling terminal. A long listing (options **lax**) provides more technical detail but may display faster by avoiding username lookups. The similar UNIX syntax uses the options **-ef** to display all processes(including PPID).
- By default, **ps** with no options selects all processes with the same effective user ID (**EUID**) as the current user, and which are associated with the same terminal where **ps** was invoked.
- Any command or pipeline can be started in the background by appending an ampersand (**&**) to the end of the command line.
- When a command line containing a pipe is sent to the background using an ampersand, the **PID** of the **last** command in the pipeline is used as output. All processes in the pipeline are still members of that job.
- You can display the list of jobs that Bash is tracking for a particular session with the **jobs** command.
- A background job can be brought to the foreground by using the **fg** command with its **job ID** (*%job number*).
- To send a foreground process to the background, first press the keyboard generated suspend request **(Ctrl+z)** in the terminal.
- The **ps j** command displays information relating to jobs.
- To start the suspended process running in the background, use the **bg** command with its **job ID** (*%job number*).
- The **+** sign indicates that this job is the current default job. That is, if a command is used that expects a *%job number* argument and a job number is not provided, then the action is taken on the job with the **+** indicator.
- Use the **ps** command with the **jT** option to view the remaining jobs.
- You signal their current foreground process by pressing a keyboard control sequence to suspend/stop (**Ctrl+z**), kill (**Ctrl+c**), or core dump (**Ctrl+**) the process.
- The **kill** command sends a signal to a process by PID number.
- Use the **kill -l** command to list the names and numbers of all available signals.
- The **killall** command can signal multiple processes.
- Use **pkill** to send a signal to one or more processes which match selection criteria.
- Use the **w** command to list user logins and current running processes.
- Use the **pstree** command to view a process tree for the system or a single user.

- The **uptime** command is one way to display the current load average. It prints the current time, how long the machine has been up, how many user sessions are running, and the current load average.
- The **lscpu** command can help you determine how many CPUs a system has.
- top then d: to change delay
- top then m: to order by memory usage (process)
- top then t: to order by cpu usage (process)
- top then h: to help
- top then L: to search
- echo $$ -> current PID
- ls /proc/ -> all process IDs
- ps aux | grep *process ID* = pidof *process name* = pgrep *process name*
- kill systemd -> shutdown
- uptime: first line in top command

Chapter 9:

- The **systemd** daemon manages startup for Linux, including service startup and service management in general. Daemons are processes that either wait or run in the background, performing various tasks.
- You use the **systemctl** command to explore the current state of the system that are both loaded and active.
- By default, the **systemctl list-units --type=service** command lists only the service units with **active** activation states. The **--all** option lists all service units regardless of the activation states. Use the **--state=** option to filter by the values in the LOAD, ACTIVE, or SUB fields.
- The **systemctl list-units** command displays units that the **systemd** service attempts to parse and load into memory.
- **systemctl list-unit-files --type=service:** List the enabled or disabled states of all service units.
- View the status of a specific unit with **systemctl status name.type**. If the unit type is not provided, **systemctl** will show the status of a service unit if one exists.
- **systemctl is-active name.type:** to verify that a service unit is currently active (running).
- **systemctl is-enabled name.type:** to verify whether a service unit is enabled to start automatically during system boot.
- **systemctl is-failed name.type:** To verify whether the unit failed during startup.
- To start a service, first verify that it is not running with **systemctl status**. Then, use the **systemctl start** command as the root user.
- To stop a currently running service, use the **stop** argument with the **systemctl** command.
- To restart a running service, use the **restart** argument with the **systemctl** command.

- . To reload a running service, use the **reload** argument with the **systemctl** command.
- In case you are not sure whether the service has the functionality to reload the configuration file changes, use the **reload-or-restart** argument with the **systemctl** command.
- The **systemctl list-dependencies UNIT** command displays a hierarchy mapping of dependencies to start the service unit. To list reverse dependencies (units that depend on the specified unit), use the --reverse option with the command.
- **systemctl mask UNIT:** Masking a service prevents an administrator from accidentally starting a service that conflicts with others.
- Use the **systemctl unmask NAME** command to unmask the service unit.
- To start a service at boot, use the **systemctl enable UNIT** command
- To disable the service from starting automatically, use the **systemctl disable UNIT** command.

Task	**Command**
View detailed information about a unit state.	**systemctl status UNIT**
Stop a service on a running system.	**systemctl stop UNIT**
Start a service on a running system.	**systemctl start UNIT**
Restart a service on a running system.	**systemctl restart UNIT**
Reload the configuration file of a running service.	**systemctl reload UNIT**
Completely disable a service from being started, both manually and at boot.	**systemctl mask UNIT**
Make a masked service available.	**systemctl unmask UNIT**
Configure a service to start at boot time.	**systemctl enable UNIT**
Disable a service from starting at boot time.	**systemctl disable UNIT**
List units required and wanted by the specified unit.	**systemctl list-dependencies UNIT**

Chapter 10:

- You can use the **ssh** command to create a secure connection to a remote system, authenticate as a specific user, and get an interactive shell session on the remote system as that user. You may also use the **ssh** command to run an individual command on the remote system without running an interactive shell.
- The **ssh** command would log you in on the remote server *remotehost* using the same username as the current local user.
- Public keys are stored in the */etc/ssh/ssh_known_hosts* and each users' *~/.ssh/known_hosts* file on the SSH client.
- Each remote SSH server that you conect to stores its public key in the */etc/ssh* directory in files with the extension *.pub*.
- To create a private key and matching public key for authentication, use the **ssh-keygen** command. By default, your private and public keys are saved in your *~/.ssh/id_rsa* and *~/.ssh/id_rsa.pub* files, respectively.
- The **ssh-copy-id** command copies the public key of the SSH keypair to the destination system.
- **ssh-agent** will automatically provide the passphrase for you(**eval $(ssh-agent)**).
- The **ssh-add** commands add the private keys from **/home/user/.ssh/id_rsa** (the default) and **/home/user/.ssh/key-with-pass** files.
- OpenSSH service is provided by a daemon called **sshd**. Its main configuration file is */etc/ssh/sshd_config*.
- With the **PermitRootLogin** parameter to *yes*, as it is by default, people are permitted to log in as root. To prevent this, set the value to *no*. Alternatively, to prevent password-based authentication but allow private key-based authentication for root, set the **PermitRootLogin** parameter to *without-password*.
- The SSH server (**sshd**) must be reloaded for any changes to take effect.
- The default value of yes for the **PasswordAuthentication** parameter in the */etc/ssh/sshd_config* configuration file causes the SSH server to allow users to use password-based authentication while logging in. The value of no for **PasswordAuthentication** prevents users from using password-based authentication.

Chapter 11:

- Processes and the operating system kernel record a log of events that happen. These logs are used to audit the system and troubleshoot problems.
- Many systems record logs of events in text files which are kept in the **/var/log** directory. These logs can be inspected using normal text utilities such as less and tail.
- The **systemd-journald** and **rsyslog** services handle the syslog messages in Red Hat Enterprise Linux 8.
- The **rsyslog** service sorts and writes syslog messages to the log files that do persist across reboots in /var/log.
- The **systemd-journald** service is stored on a file system that does not persist across reboots.

Log file	Type of Messages Stored
/var/log/messages	Most syslog messages are logged here. Exceptions include messages related to authentication and email processing, scheduled job execution, and those which are purely debugging-related.
/var/log/secure	Syslog messages related to **security and authentication events**.
/var/log/maillog	Syslog messages related to **mail server**.
/var/log/cron	Syslog messages related to **scheduled job execution**.
/var/log/boot.log	Non-syslog console messages related to **system startup**.

- Many programs use the syslog protocol to log events to the system. Each log message is categorized by a **facility** (the type of message) and a **priority** (the severity of the message).
- The rsyslog service uses the facility and priority of log messages to determine how to handle them. This is configured by rules in the **/etc/rsyslog.conf** file and any file in the **/etc/rsyslog.d** directory that has a file name extension of **.conf**.
- Each rule that controls how to sort syslog messages is a line in one of the configuration files. The left side of each line indicates the facility and severity of the syslog messages the rule matches. The right side of each line indicates what file to save the log message in (or where else to deliver the message). An asterisk (*****) is a wildcard that matches all values.
- Log messages sometimes match more than one rule in **rsyslog.conf**. In such cases, one message is stored in more than one log file. To limit messages stored, the key word **none** in the priority field indicates that no messages for the indicated facility should be stored in the given file.
- The **logrotate** tool rotates log files to keep them from taking up too much space in the file system containing the /var/log directory.
- After a certain number of rotations, typically after four weeks, the oldest log file is discarded to free disk space. A scheduled job runs the **logrotate** program daily to see if any logs need to be rotated. Most log files are rotated weekly, but **logrotate** rotates some faster, or slower, or when they reach a certain size.

- Log messages start with the oldest message on top and the newest message at the end of the log file.
- Monitoring one or more log files for events is helpful to reproduce problems and issues. The **tail -f /path/to/file** command outputs the last 10 lines of the file specified and continues to output new lines in the file as they get written.
- The **logger** command can send messages to the **rsyslog** service. By default, it sends the message to the **user facility** with the **notice** priority (**user.notice**) unless specified otherwise with the **-p** option.
- To send a message to the **rsyslog** service that gets recorded in the **/var/log/boot.log** log file, execute the following logger command:
    - [root@host ~]# logger -p local7.notice "Log entry created on host"
- To retrieve log messages from the journal, use the **journalctl** command.
- The **journalctl** command highlights important log messages: messages at notice or warning priority are in bold text while messages at the error priority or higher are in red text.
- By default, **journalctl -n** shows the last 10 log entries.
- The **journalctl -f** command outputs the last 10 lines of the system journal and continues to output new journal entries as they get written to the journal.
- The **journalctl -p** takes either the name or the number of a priority level and shows the journal entries for entries at that priority and above. The **journalctl** command understands the debug, info, notice, warning, err, crit, alert, and emerg priority levels.
- When looking for specific events, you can limit the output to a specific time frame. The **journalctl** command has two options to limit the output to a specific time range, the **--since** and **--until** options. Both options take a time argument in the format **"YYYY-MM-DD hh:mm:ss"** (the double-quotes are required to preserve the space in the option). If the date is omitted, the command assumes the current day, and if the time is omitted, the command assumes the whole day starting at 00:00:00. Both options take **yesterday**, **today**, and **tomorrow** as valid arguments in addition to the date and time field.
- Run the following **journalctl** command to list all journal entries ranging from 2019-02-10 20:30:00 to 2019-02-13 12:00:00:
    - [root@host ~]# journalctl --since "2019-02-10 20:30:00" --until "2019-02-13 12:00:00"
- To specify all entries in the last hour:
    - [root@host ~]# journalctl --since "-1 hour"
- There are fields attached to the log entries that can only be seen when verbose output is turned on:
    - [root@host ~]# journalctl -o verbose

- The following list gives the common fields of the system journal that can be used to search for lines relevant to a particular process or event:
    - _COMM is the name of the command
    - _EXE is the path to the executable for the process
    - _PID is the PID of the process
    - _UID is the UID of the user running the process
    - _SYSTEMD_UNIT is the systemd unit that started the process
- The following **journalctl** command shows all journal entries related to the sshd.service systemd unit from a process with PID 1182:
    - [root@host ~]# journalctl _SYSTEMD_UNIT=sshd.service _PID=1182
- The Storage parameter in the **/etc/systemd/journald.conf** file defines whether to store system journals in a volatile manner or persistently across reboot. Set this parameter to **persistent**, **volatile**, or **auto** as follows:
    - **persistent**: stores journals in the /var/log/journal directory which persists across reboots.
    - **volatile**: stores journals in the volatile /run/log/journal directory.
    - **auto**: rsyslog determines whether to use persistent or volatile storage. If the **/var/log/journal** directory exists, then rsyslog uses persistent storage, otherwise it uses volatile storage.
- The following command output shows the journal entries that reflect the current size limits:
    - [user@host ~]$ journalctl | grep -E 'Runtime|System journal'
- To configure the **systemd-journald** service to preserve system journals persistently across reboot, set Storage to persistent in the **/etc/systemd/journald.conf** file. After editing the configuration file, restart the **systemd-journald** service to bring the configuration changes into effect.
    - [root@host ~]# systemctl restart systemd-journald
- The following **journalctl** command retrieves the entries limited to the first system boot:
    - [root@host ~]# journalctl -b 1
- The following **journalctl** command retrieves the entries limited to the current system boot:
    - [root@host ~]# journalctl -b
- Use the **systemctl reboot** command to restart server.
- The **timedatectl** command shows an overview of the current time-related system settings (and to verify that the time zone has been updated).
- A database of time zones is available and can be listed with the **timedatectl list-timezones** command.
- The command **tzselect** is useful for identifying correct **zoneinfo** time zone names.

- The superuser can change the system setting to update the current time zone using the **timedatectl set-timezone *Continent/Country*** command.
- Use the **timedatectl set-time** command to change the system's current time.
- The **timedatectl set-ntp** command enables or disables NTP synchronization for automatic time adjustment. The option requires either a **true** or **false** argument to turn it on or off.
- If no network connectivity is available, **chronyd** calculates the RTC clock drift, which is recorded in the **driftfile** specified in the **/etc/chrony.conf** configuration file.
- The **chronyc** command acts as a client to the **chronyd** service. After setting up NTP synchronization, you should verify that the local system is seamlessly using the NTP server to synchronize the system clock using the **chrony sources** command. For more verbose output with additional explanations about the output, use the **chronyc sources -v** command.

Chapter 13:

- With **tar** command, users can gather large sets of files into a single file (archive). A tar archive is a structured sequence of file data mixed in with metadata about each file and an index so that individual files can be extracted.

Option	Description
-c, --create	Create a new archive.
-x, --extract	Extract from an existing archive.
-t, --list	List the table of contents of an archive.

Option	Description
-v, --verbose	Verbose. Shows which files get archived or extracted.
-f, --file=	File name. This option must be followed by the file name of the archive to use or create.
-p, --preserve-permissions	Preserve the permissions of files and directories when extracting an archive, without subtracting the umask.

- The first option to use when creating a new archive is the **c** option, followed by the **f** option, then a **single space**, then the **file name** of the archive to be created, and finally the **list of files and directories** that should get added to the archive.
    - [user@host ~]$ tar -cf archive.tar file1 file2 file3
    - [user@host ~]$ tar --file=archive.tar --create file1 file2 file3
- The **tar** command overwrites an existing archive without warning.
- For **tar** to be able to archive the selected files, it is mandatory that the user executing the tar command can read the files.
- The **t** option directs tar to list the contents (table of contents, hence t) of the archive. Use the **f** option with the name of the archive to be queried.
- A **tar** archive should usually be extracted in an empty directory to ensure it does not overwrite any existing files. When root extracts an archive, the tar command preserves the original user and group ownership of the files. If a regular user extracts files using tar, the file ownership belongs to the user extracting the files from the archive.
- By default, when files get extracted from an archive, the umask is subtracted from the permissions of archive content. To preserve the permissions of an archived file, the **p** option when extracting an archive.
- The **tar** command supports three compression methods. There are three different compression methods supported by the **tar** command. The **gzip** compression is the fastest and oldest one and is most widely available across distributions and even across platforms. **bzip2** compression creates smaller archive files compared to **gzip** but is less widely available than **gzip**, while the **xz** compression method is relatively new, but usually offers the best compression ratio of the methods available.

- Use one of the following options to create a compressed tar archive:
    - -z or --gzip for gzip compression (filename.tar.gz or filename.tgz)
    - -j or --bzip2 for bzip2 compression (filename.tar.bz2)
    - -J or -xz for xz compression (filename.tar.xz)
- To create a **gzip** compressed archive named **/root/etcbackup.tar.gz**, with the contents from the **/etc** directory on host:
    - [root@host ~]# tar -czf /root/etcbackup.tar.gz /etc
- To create a **bzip2** compressed archive named **/root/logbackup.tar.bz2**, with the contents from the **/var/log** directory on host:
    - [root@host ~]$ tar -cjf /root/logbackup.tar.bz2 /var/log
- To create a **xz** compressed archive named **/root/sshconfig.tar.xz**, with the contents from the **/etc/ssh** directory on host:
    - [root@host ~]$ tar -cJf /root/sshconfig.tar.xz /etc/ssh
- To extract the contents of a **gzip** compressed archive named **/root/etcbackup.tar.gz** in the **/tmp/etcbackup** directory:
    - [root@host etcbackup]# tar -xzf /root/etcbackup.tar.gz
- To extract the contents of a **bzip2** compressed archive named **/root/logbackup.tar.bz2** in the **/tmp/logbackup** directory:
    - [root@host logbackup]# tar -xjf /root/logbackup.tar.bz2
- To extract the contents of a **xz** compressed archive named **/root/sshbackup.tar.xz** in the **/tmp/sshbackup** directory:
    - [root@host sshbackup]# tar -xJf /root/sshbackup.tar.xz
- The Secure Copy command, **scp**, which is part of the OpenSSH suite, copies files from a remote system to the local system or from the local system to a remote system.
- The following example demonstrates how to copy the local **/etc/yum.conf** and **/etc/hosts** files on host, to the **remoteuser**'s home directory on the **remotehost** remote system:
    - [user@host ~]$ scp /etc/yum.conf /etc/hosts remoteuser@remotehost:/home/remoteuser
- You can also copy a file in the other direction, from a remote system to the local file system. In this example, the file **/etc/hostname** on **remotehost** is copied to the local directory **/home/user**:
    - [user@host ~]$ scp remoteuser@remotehost:/etc/hostname /home/user
- To copy a whole directory tree recursively, use the **-r**.
- To interactively upload or download files from a SSH server, use the Secure File Transfer Program, **sftp**.
- The interactive **sftp** session accepts various commands that work the same way on the remote file system as they do in the local file system, such as **ls, cd, mkdir, rmdir, and pwd**. The **put** command uploads a file to the remote system. The **get** command downloads a file from the remote system. The **exit** command exits the sftp session.

- The **rsync** command is another way to securely copy files from one system to another. It differs from **scp** in that if two files or directories are similar between two servers, **rsync** copies only the differences between the file systems, while scp would still copy everything.
- An important option of **rsync** is the **-n** option to perform a dry run. A dry run is a simulation of what happens when the command gets executed. The dry run shows the changes **rsync** would perform when the command is run normally. Perform a dry run before the actual **rsync** operation to ensure no important files get overwritten or deleted.
- The **-a** or **--archive** option enables "archive mode". This enables recursive copying and turns on many useful options that preserve most characteristics of the files.
- Archive mode does not preserve **hard links**, because this can add significant time to the synchronization. If you want to preserve hard links too, add the **-H** option.