AMMINISTRAZIONE DI SISTEMI IT E CLOUD

Docente: Luca Cavatorta

Contatti: luca.cavatorta@unipr.it

Orari di Ricevimento:
In call Teams su prenotazione via mail

Modalità corso:

- **Slide scaricabili da piattaforma Elly 2022 del corso**
- **Esercizi pratici sui sistemi virtuali locali**
- **Lezioni registrate disponibili sul portale Elly**

Modalità di Esame:

- **Domande di teoria**
- **Esercizi sui sistemi forniti in sede di esame**

Prerequisiti:

- **PC personale; accesso ad internet**

Testi (FACOLTATIVI!!!!)

Linux:
“Linux BIBLE”
https://www.wiley.com/en-us/Linux+Bible%2C+10th+Edition-p-9781119578895

Docker:
“Learn Docker”
https://www.packtpub.com/product/learn-docker-fundamentals-of-docker-19-x-second-edition/9781838827472
https://docker.com

Kubernetes:
https://kubernetes.io/

Install Linux Fedora Server in VirtualBox:

VirtualBox:
https://www.virtualbox.org
User Manual
https://www.virtualbox.org/manual/UserManual.html
Operating System:
Fedora : https://getfedora.org/it/server/download/

Lab:

https://elly2022.dia.unipr.it

THE BASH SHELL

A command line is a text-based interface which can be used to input instructions to a computer system. The Linux command line is provided by a program called the shell. Over the long history of UNIX-like systems, many shells have been developed. The default shell for users in Fedora Linux is the GNU Bourne-Again Shell (bash). Bash is an improved version of one of the most successful shells used on UNIX-like systems, the Bourne Shell (sh)

.

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

```
[student@desktop ~]$
```

The $ is replaced by a # if the shell is running as the superuser, root. This makes it more obvious that it is a superuser shell, which helps to avoid accidents and mistakes in the privileged account.

```
[root@desktop ~]#
```

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

SHELL BASICS

Commands entered at the shell prompt have three basic parts:

- Command to run
- Options to adjust the behavior of the command
- Arguments, which are typically targets of the command

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

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

Most commands have a **--help** option. This causes the command to print a description of what it does, a "usage statement" that describes the command's syntax, and a list of the options it accepts and what they do.

Usage statements may seem complicated and difficult to read. They become much simpler to understand once a user becomes familiar with a few basic conventions:
Squarebrackets, **[]**, surround optional items.
Anything followed by… represents an arbitrary-length list of items of that type.
Multiple items separated by pipes, |, means only one of them can be specified.
Text in angle brackets, <>, represents variable data. For example, <filename> means “insert the filename you wish to use here”. Sometimes these variables are simply written in capital letters (e.g., FILENAME).

Consider the first usage statement for the date command:
[student@desktop ~]$ date --help

BASIC COMMAND SYNTAX

The GNU Bourne-Again Shell (bash) is a program that interprets commands typed in by the user. Each string typed into the shell can have up to three parts: the command, options (that begin with a - or --), and arguments. Each word typed into the shell is separated from each other with spaces. Commands are the names of programs that are installed on the system. Each command has its own options and arguments.
The Enter key is pressed when a user is ready to execute a command. Each command is typed on a separate line and the output from each command displays before the shell displays a prompt.

If a user wants to type more than one command on a single line, a semicolon, **;**, can be used as a command separator. A semicolon is a member of a class of characters called metacharacters that has special meanings for bash.

EXAMPLES OF SIMPLE COMMANDS

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

```
[teacher@desktop ~]$ date
Tue Sep 17 09:05:07 CEST 2019
[teacher@desktop ~]$ date
Tue Sep 17 09:05:07 CEST 2019
[teacher@desktop ~]$ date +%R
09:05
[teacher@desktop ~]$ date +%x
09/17/2019
```

- The **passwd** command changes a user's own password. The original password for the account must be specified before a change will be allowed. By default, passwd is configured to require a strong password, consisting of lowercase letters, uppercase letters, numbers, and symbols, and is not based on a dictionary word. The superuser can use the passwd command to change other users' passwords.

```
[root@desktop ~]# passwd teacher
Changing password for user teacher.
```

```
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

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

```
[root@desktop ~]# file /bin/passwd
/bin/passwd: setuid ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=0a16a7915f7f9b01d96442755257e22067ce5b2c, stripped
[root@desktop ~]# file /etc/shadow
/etc/shadow: ASCII text
```

- The **head** and **tail** commands display the beginning and end of a file respectively. By default, these commands display 10 lines, 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.

```
[root@desktop ~]# head -2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
```

```
[root@desktop ~]# tail -2 /etc/passwd
minnie:x:1004:1005::/home/minnie:/bin/bash
goofy:x:1005:1006::/home/goofy:/bin/bash
```

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

```
[root@desktop ~]# wc -l /etc/passwd
25 /etc/passwd
[root@desktop ~]# wc -w /etc/passwd
33 /etc/passwd
[root@desktop ~]# wc -c /etc/passwd
1106 /etc/passwd
[root@desktop ~]# wc /etc/passwd
  25   33 1106 /etc/passwd
```

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

```
[root@desktop ~]# history
  365  wc -l /etc/passwd
  366  wc -w /etc/passwd
  367  wc -c /etc/passwd
  [root@desktop ~]# !367
```

```
wc -c /etc/passwd
1106 /etc/passwd
```

TAB COMPLETION

Tab completion allows a user to quickly complete commands or file names once 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.

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

EDITING THE COMMAND LINE

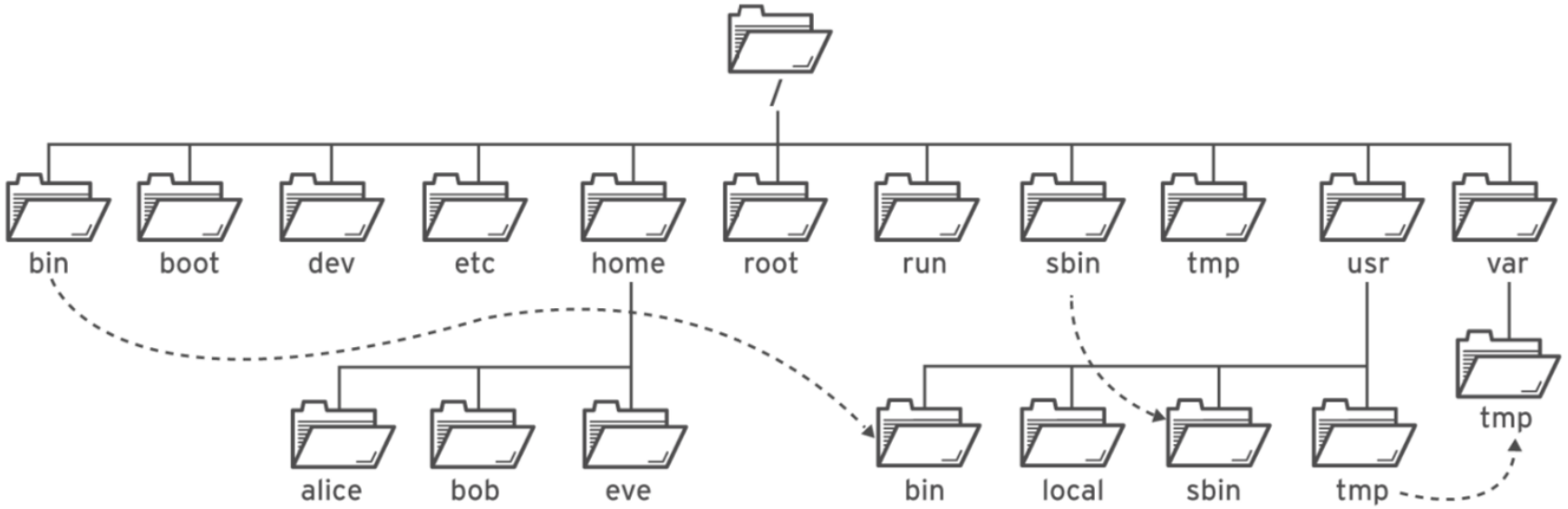
When used interactively, bash has a command line-editing feature. This allows the user to use text editor commands to move around within and modify the current command being typed. Using the arrow keys to move within the current command and to step through the command history was introduced earlier in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

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+Left Arrow	Jump to the beginning of the previous word on the command line.
Ctrl+Right Arrow	Jump to the end of the next word on the command line.
Ctrl+r	Search the history list of commands for a pattern.

THE LINUX FILE SYSTEM HIERARCHY

THE FILE SYSTEM HIERARCHY

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



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, we could call that directory **/etc**. Likewise, if the **/etc** directory contained a file named issue, we could refer to that file as **/etc/issue**.

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

LOCATION	PURPOSE
/usr	Installed software, shared libraries, include files, and static read-only program data. Important subdirectories include: - /usr/bin: User commands. - /usr/sbin: System administration commands. - /usr/local: Locally customized software.
/etc	Configuration files specific to this system.
/var	Variable data specific to this system that should persist between boots. Files that dynamically change (e.g. databases, cache directories, log files, printer- spooled

	documents, and website content) may be found under /var.
/run	Runtime data for processes started since the last boot. This includes process ID files and lock files, among other things. The contents of this directory are recreated on reboot. (This directory consolidates /var/run and /var/lock from older versions of Linux.)
/home	Home directories where regular users store their personal data and configuration files.
/root	Home directory for the administrative superuser, root.
/tmp	A world-writable space for temporary files. Files which have not been accessed, changed, or modified for 10 days are deleted from this directory automatically. Another temporary directory exists, /var/tmp, in which files that have not been accessed, changed, or modified in more than 30 days are deleted automatically.
/boot	Files needed in order to start the boot process.
/dev	Contains special device files which are used by the system to access hardware.

ABSOLUTE PATHS AND RELATIVE PATHS

Absolute paths

An absolute path is a fully qualified name, beginning at the root (/) directory and specifying each subdirectory traversed to reach and uniquely represent a single file. Every file in a file system has a unique absolute path name, recognized with a simple rule: A path name with a forward slash (/) as the first character is an absolute path name. For example, the absolute path name for the system message log file is **/var/log/messages**.

Relative paths

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 a first character is a relative path name. A user in the **/var** directory could refer to the message log file relatively as **log/messages.**

NAVIGATING PATHS

The **pwd** command displays the full path name of the current location, which helps determine appropriate syntax for reaching files using relative path names. The **ls** command lists directory contents for the specified directory or, if no directory is given, for the current directory.

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

Use the **cd** command to change directories.

```
[student@desktop ~]$cd Videos
[student@desktop Videos]$pwd
/home/student/Videos
[student@desktop Videos]$ cd /home/student/Documents
[student@desktop Documents]$ pwd
/home/student/Documents
[student@desktop Documents]$ cd
[student@desktop ~]$ pwd
/home/student
[student@desktop ~]$
```

The **cd ..** command uses the **..** hidden directory to move up one level to the parent directory, without needing to know the exact parent name

```
[student@desktopX Videos]$pwd
/home/student/Videos
[student@desktopX Videos]$ cd .
[student@desktopX Videos]$ pwd
/home/student/Videos
[student@desktopX Videos]$ cd ..
[student@desktopX ~]$ pwd
/home/student
[student@desktopX ~]$ cd ..
[student@desktopX home]$ pwd
/home
[student@desktopX home]$ cd ..
[student@desktopX /]$ pwd
/
[student@desktopX /]$ cd
[student@desktopX ~]$ pwd
/home/student
```

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

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

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

```
[teacher@desktop ~]$ ls -l
total 4
drwxr-xr-x. 3 teacher teachers 18 Sep 17 09:00 dir
-rw-r--r--. 1 teacher teachers  0 Sep 17 08:59 file1
-rw-r--r--. 1 teacher teachers  5 Sep 17 09:00 file2

[teacher@desktop ~]$ ls -la
total 20
drwx------. 3 teacher teachers 120 Sep 17 09:00 .
drwxr-xr-x. 9 root    root     107 Sep  4 17:19 ..
-rw-------. 1 teacher teachers  66 Aug 21 01:12 .bash_history
-rw-r--r--. 1 teacher teachers  18 May 22  2018 .bash_logout
```

```
-rw-r--r--. 1 teacher teachers 193 May 22  2018 .bash_profile
-rw-r--r--. 1 teacher teachers 231 May 22  2018 .bashrc
drwxr-xr-x. 3 teacher teachers  18 Sep 17 09:00 dir
-rw-r--r--. 1 teacher teachers   0 Sep 17 08:59 file1
-rw-r--r--. 1 teacher teachers   5 Sep 17 09:00 file2

[teacher@desktop ~]$ ls -R
.:
dir  file1  file2

./dir:
dir1

./dir/dir1:
dir2

./dir/dir1/dir2:
file2
```

COMMAND-LINE FILE MANAGEMENT

File management involves **creating**, **deleting**, **copying**, and **moving** files. Additionally, directories can be **created**, **deleted**, **copied**, and **moved** to help organize files logically. When working at the command line, file management requires awareness of the current working directory to choose either absolute or relative path syntax as most efficient for the immediate task.

ACTIVITY	SINGLE SOURCE	MULTIPLE SOURCE
Copy file	cp file1 file2	cp file1 file2 file3 dir1
Move file	mv file1 file2	mv file1 file2 file3 dir
Remove file	rm file1	rm -f file1 file2 file3
Create directory	mkdir dir1	mkdir -p par1/par2/dir1
Copy directory	cp -r dir1 dir2	cp -r dir1 dir2 dir3 dir4
Move directory	mv dir1 dir2	mv dir1 dir2 dir3 dir4

Remove directory	rm -r dir1	rm -rf dir1 dir2 dir3
Remove empty directory	rmdir dir1	rmdir -p par1/par2/dir1

MATCHING FILE NAMES USING PATH NAME EXPANSION

FILE GLOBBING: PATH NAME EXPANSION

The Bash shell has a path name-matching capability historically called globbing, abbreviated from the “global command” file path expansion program of early UNIX. The Bash globbing feature, commonly called pattern matching or “wildcards”, makes managing large numbers of files easier. Using meta-characters that “expand” to match file and path names being sought, commands perform on a focused set of files at once.

Pattern matching

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

PATTERN	MATCHES
*	Any string of zero or more characters.
?	Any single character.
~	The current user's home directory.
~username	User username's home directory.
~+	The current working directory.

~-	The previous working directory.
[abc...]	Any one character in the enclosed class.
[!abc...]	Any one character not in the enclosed class.
[^abc...]	Any one character not in the enclosed class.
[[:alpha:]]	Any alphabetic character.
[[:lower:]]	Any lower-case character.
[[:upper:]]	Any upper-case character.
[[:alnum:]]	Any alphabetic character or digit.
[[:punct:]]	Any printable character not a space or alphanumeric.
[[:digit:]]	Any digit, 0-9.
[[:space:]]	Any one whitespace character; may include tabs, newline, or carriage returns, and form feeds as well as space.

EXAMPLE

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

```
[student@desktop glob]$ls
able  alfa  baker  bravo  cast  charlie  delta  dog  easy  echo
[student@desktop glob]$
```

First, simple pattern matches using * and ?.

```
[student@desktop glob]$ls a*
able alfa
[student@desktop glob]$ ls *a*
able alfa baker bravo cast charlie delta easy
[student@desktop glob]$ ls [ac]*
able alfa cast charlie
```

Tilde expansion

The tilde character (~), when followed by a slash delimiter, matches the current user's home directory. When followed by a string of characters up to a slash, it will be interpreted as a username, if one matches. If no username matches, then an actual tilde followed by the string of characters will be returned.

```
[student@desktop glob]$ls ~/glob
able alfa baker bravo cast charlie delta dog easy echo
```

```
[student@desktop glob]$ echo ~/glob
/home/student/glob
```

Brace expansion

Brace expansion is used to generate discretionary strings of characters. Braces contain a comma separated list of strings, or a sequence expression. The result includes the text preceding or following the brace definition. Brace expansions may be nested, one inside another.

```
[student@desktop glob]$echo {Sunday,Monday,Tuesday,Wednesday}.log
Sunday.log Monday.log Tuesday.log Wednesday.log
[student@desktop glob]$ echo file{1..3}.txt
file1.txt file2.txt file3.txt
[student@desktop glob]$ echo file{a..c}.txt
filea.txt fileb.txt filec.txt
[student@desktop glob]$ echo file{a,b}{1,2}.txt
filea1.txt filea2.txt fileb1.txt fileb2.txt
[student@desktop glob]$ echo file{a{1,2},b,c}.txt
filea1.txt filea2.txt fileb.txt filec.txt
```

Command substitution

Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed with a beginning dollar sign and parenthesis, **$(command)**, or with backticks, `command`. The form with backticks is older, and has two disadvantages:
1) it can be easy to visually confuse backticks with single quote marks, and
2) backticks cannot be nested inside backticks. The $(command) form can nest multiple command expansions inside each other.

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

Protecting arguments from expansion

Many characters have special meaning in the Bash shell. To ignore meta-character special meanings, quoting and escaping are used to protect them from shell expansion. The backslash (\) is an escape character in Bash, protecting the single following character from special interpretation. To protect longer character strings, single (') or double quotes (") are used to enclose strings.

Use double quotation marks to suppress globbing and shell expansion, but still allow command and variable substitution. Variable substitution is conceptually identical to command substitution, but may use optional brace syntax.

```
[student@desktop glob]$host=$(hostname -s); echo $host
desktop
[student@desktop glob]$ echo "***** hostname is ${host} *****"
***** hostname is desktop *****
[student@desktopX glob]$ echo Your username variable is \$USER.
Your username variable is $USER.
```

Use single quotation marks to interpret all text literally. Observe the difference, on both screen and keyboard, between the single quote (') and the command substitution backtick (`). Besides suppressing globbing and shell expansion, quotations direct the shell to additionally suppress command and variable substitution. The question mark is a meta-character that also needed protection from expansion.

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

LAB - MANAGING FILES WITH SHELL EXPANSION

Perform the following steps on server. Log in as student user and begin the lab in the home directory.

1. Create your student user.

```
[root@server ~]$ useradd student
[root@server ~]$ su - student  ##switch to student user from root
```

2. To begin, create sets of empty practice files to use in this lab. If an intended shell expansion shortcut is not immediately recognized, students are expected to use the solution to learn and practice. Use shell tab completion to locate file path names easily. Create a total of 12 files with names tv_seasonX_episodeY.ogg. Replace X with the season number and Y with that season's episode, for two seasons of six episodes each.

```
[student@server ~]$
touch tv_season{1..2}_episode{1..6}.ogg
[student@server ~]$
ls -l
```

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

```
[student@server ~]$
touch mystery_chapter{1..8}.odf
[student@server ~]$
ls -l
```

4. To organize the TV episodes, create two subdirectories named season1 and season2 under the Videos directory (create it). Use one command.

```
[student@server ~]$
mkdir -p Videos/season{1..2}
[student@server ~]$
ls -lR
```

5. Move the appropriate TV episodes in to the season subdirectories. Use only two commands, specifying destinations using relative syntax.

```
[student@server ~]$
mv tv_season1* Videos/season1
[student@server ~]$
mv tv_season2* Videos/season2
[student@server ~]$
```

```
ls -lR
```

6. To organize the my sterybook chapters, create a two-level directory hierarchy with one command. Create my_bestseller under the Documents directory (create it), and chapters beneath the new my_bestseller directory.

```
[student@server ~]$
mkdir -p Documents/my_bestseller/chapters
[student@server ~]$
ls -lR
```

7. Using one command, create three more subdirectories directly under themy_bestseller directory. Name these subdirectories editor, plot_change, and vacation. The create parent option is not needed since the my_bestseller parent directory already exists.

```
[student@server ~]$
mkdir -p Documents/my_bestseller/{editor,plot_change,vacation}
[student@server ~]$
ls -lR
```

8. Change to the chapters directory. Using the home directory shortcut to specify the source files, move all book chapters into the chapters directory, which is now your current directory. What is the simplest syntax to specify the destination directory?

```
[student@server ~]$
```

```
cd Documents/my_bestseller/chapters
[student@server chapters]$
mv ~/mystery_chapter* .
[student@server chapters]$
ls -l
```

9. The first two chapters are sent to the editor for review. To remember to not modify these chapters during the review, move those two chapters only to the editor directory. Use relative syntax starting from the chapters subdirectory.

```
[student@server chapters]$
mv mystery_chapter1.odf mystery_chapter2.odf ../editor
[student@server chapters]$
ls -l
[student@server chapters]$
ls -l ../editor
```

10. Chapters 7 and 8 will be written while on vacation. Move the files from chapters to vacation. Use one command without wildcard characters.

```
[student@server chapters]$
mv mystery_chapter7.odf mystery_chapter8.odf ../vacation
[student@server chapters]$
ls -l
```

```
[student@server chapters]$
ls -l ../vacation
```

11. With one command, change the working directory to the season 2 TV episodes location, then copy the first episode of the season to the vacation directory.

```
[student@server chapters]$
cd ~/Videos/season2
[student@server season2]$
cp tv_season2_episode1.ogg ~/Documents/my_bestseller/vacation
```

12. With one command, change the working directory to vacation, then list its files. Episode 2 is also needed. Return to the season2 directory using the previous working directory shortcut. This will succeed if the last directory change was accomplished with one command. Copy the episode 2 file into vacation. Return to vacation using the shortcut again.

```
[student@server season2]$
cd ~/Documents/my_bestseller/vacation
[student@server vacation]$
ls -l
[student@server vacation]$
cd -
[student@server season2]$
cp tv_season2_episode2.ogg ~/Documents/my_bestseller/vacation
```

```
[student@server vacation]$
cd -
[student@server vacation]$
ls -l
```

13. Chapters 5 and 6 may need a plotchange. To prevent these changes from modifying original files, copy both files into plot_change. Move up one directory to vacation's parent directory, then use one command from there.

```
[student@server vacation]$
cd ..
[student@server my_bestseller]$
cp chapters/mystery_chapter[56].odf plot_change
[student@server my_bestseller]$
ls -l chapters
[student@server my_bestseller]$
ls -l plot_change
```

14. To track changes, make three backups of chapter 5. Change to the plot_change directory. Copy mystery_chapter5.odf as a new file name to include the full date (Year-Mo-Da). Make another copy appending the current timestamp (as the number of seconds since the epoch) to ensure a unique file name. Also make a copy appending the current user ($USER) to the file name. See the solution for the

syntax of any you are unsure of (like what arguments to pass to the date command). Note, we could also make the same backups of the chapter 6 files too.

```
[student@server my_bestseller]$
cd plot_change
[student@server plot_change]$
cp mystery_chapter5.odf mystery_chapter5_$(date +%F).odf
[student@server plot_change]$
cp mystery_chapter5.odf mystery_chapter5_$(date +%s).odf
[student@server plot_change]$
cp mystery_chapter5.odf mystery_chapter5_$USER.odf
[student@server plot_change]$
ls -l
```

15. The plot changes were not successful. Delete the plot_change directory. First,delete all of the files in the plot_change directory. Change directory up one level because the directory cannot be deleted while it is the working directory. Try to delete the directory using the rm command without the recursive option. This attempt should fail. Now use the rmdir command, which will succeed.

```
[student@server plot_change]$ rm mystery*
[student@server plot_change]$ cd ..
[student@server my_bestseller]$ rm plot_change
rm: cannot remove 'plot_change': Is a directory
[student@server my_bestseller]$ rmdir plot_change
```

```
[student@server my_bestseller]$ ls -l
```

16. When the vacation is over, the vacation directory is no longer needed. Delete it using therm command with the recursive option.

```
[student@server my_bestseller]$
rm -r vacation
[student@server my_bestseller]$
ls -l
[student@server my_bestseller]$
cd
```

When finished, return to the home directory.

READING DOCUMENTATION USING MAN COMMAND

INTRODUCING THE MAN COMMAND

The historical Linux Programmer's Manual, from which man pages originate, was large enough to be multiple printed books. Each contained information for specific types of files, which have become the sections listed below. Articles are referred to as topics, as pages no longer applies.

SECTION	CONTENT TYPE
1	User commands (both executable and shell programs)
2	System calls (kernel routines 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 (historical section for amusing programs)

7	Conventions, standards, and miscellaneous (protocols, file systems)
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, while **passwd**(5) explains the /etc/passwd file format for storing local user accounts.

SEARCHING FOR MAN PAGES BY KEYWORD

A keyword search of man pages is performed using man -k keyword, which displays a list of keyword-matching man page topics with section numbers.

```
[student@server ~]$ man -k passwd
chgpasswd (8)         - update group passwords in batch mode
chpasswd (8)          - update passwords in batch mode
gpasswd (1)           - administer /etc/group and /etc/gshadow
mkpasswd (1)          - Overfeatured front end to crypt(3)
openssl-passwd (1ssl) - compute password hashes
pam_localuser (8)     - require users to be listed in /etc/passwd
passwd (1)            - change user password
passwd (1ssl)         - compute password hashes
update-passwd (8)     - safely update /etc/passwd, /etc/shadow and /etc/group
```

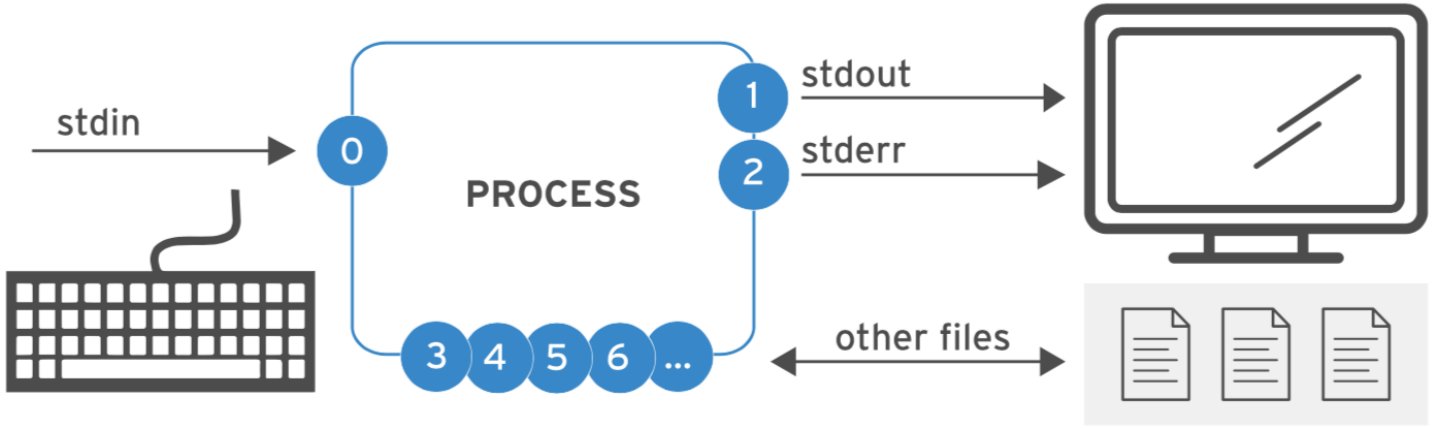
Popular system administration topics are in sections 1 (user commands), 5 (file formats), and 8 (administrative commands). Administrators using certain troubleshooting tools also use section 2 (system calls). The remaining sections are commonly for programmer reference or advanced administration.

REDIRECTING OUTPUT TO A FILE OR PROGRAM

STANDARD INPUT, STANDARD OUTPUT, AND STANDARD ERROR

A running program, or process, needs to read input from somewhere and write output to the screen or to files. A command run from the shell prompt normally reads its input from the keyboard and sends its output to its terminal window.

A process uses numbered channels called file descriptors to get input and send output. All processes will have at least three file descriptors to start with. 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, it may use higher-numbered file descriptors.



REDIRECTING OUTPUT TO A FILE

I/O redirection replaces the default channel destinations with file names representing either output files or devices. Using redirection, process output and error messages normally sent to the terminal window can be captured as file contents, sent to a device, or discarded.

Redirecting **stdout** suppresses process output from appearing on the terminal. As seen in the following table, redirecting only **stdout** does not suppress **stderr** error messages from displaying on the terminal. If the file does not exist, it will be created. If the file does exist and the redirection is not one that appends to the file, the file's contents will be overwritten. The special file **/dev/null** quietly discards channel output redirected to it and is always an empty file.

Output Redirection Operators

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	
>>fiel 2>&1	redirect **stdout** and stderr to append to the same file
&>>file	

Examples for output redirection

Many routine administration tasks are simplified by using redirection. Use the previous table to assist while considering the following examples:

1 - Save a timestamp for later reference.

```
[student@desktop ~]$date > /tmp/saved-timestamp
```

2 - Copy the last 100 lines from a log file to another file.

```
[student@desktop ~]$tail -n 100 /var/log/dmesg > /tmp/last-100-boot-messages
```

3 - Concatenate four files into one.

```
[student@desktop ~]$cat file1 file2 file3 file4 > /tmp/all-four-in-one
```

4 - List the home directory's hidden and regular filenames into a file.

```
[student@desktop ~]$ls -a > /tmp/my-file-names
```

5 - Append output to an existing file.

In the next examples, errors are generated since normal users are denied access to system directories. Redirect errors to a file while viewing normal command output on the terminal.

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

6 - Save process output and error messages to separate files.

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

7 - Ignore and discard error messages.

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

8 - Store output and generated errors to get her.

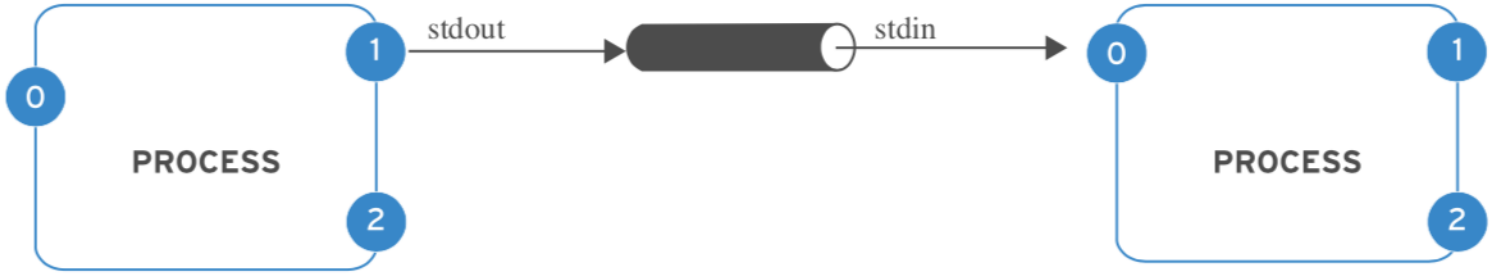
```
[student@desktop ~]$find /etc -name passwd &> /tmp/save-both
```

9 - Append output and generated errors to an existing file.

```
[student@desktop ~]$find /etc -name passwd >> /tmp/save-both 2>&1
```

CONSTRUCTING PIPELINES

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.



Pipelines allow the output of a process to be manipulated and formatted by other processes before it is output to the terminal. One useful mental image is to imagine that data is "flowing" through the pipeline from one process to another, being altered in slight ways by each command in the pipeline through which it passes.

Pipeline examples

This example takes the output of the ls command and uses less to display it on the terminal one screen at a time.

```
[student@desktop ~]$ls -l /usr/bin | less
```

The output of the ls command is piped to **wc -l,** which counts the number of lines received from ls and prints that to the terminal.

```
[student@desktop ~]$ls | wc -l
```

In this pipeline, head will output the first 10 lines of output from ls -t, with the final result redirected to a file.

```
[student@desktop ~]$ls -t | head -n 10 > /tmp/ten-last-changed-files
```

Pipelines, redirection, and **tee**

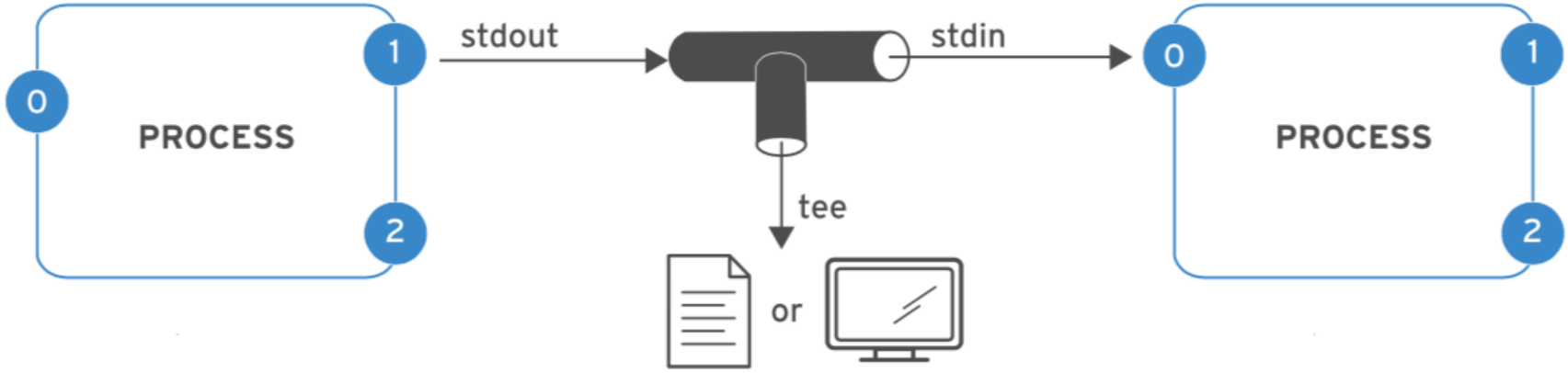
When redirection is combined with a pipeline, the shell first sets up the entire pipeline, then it redirects input/output.
What this means is that if output redirection is used in the middle of a pipeline, the output will go to the file and not to the next command in the pipeline.

In this example, the output of the ls command will go to the file, and less will display nothing on the terminal.

```
[student@desktop ~]$ls > /tmp/saved-output | less
```

The **tee** command is used to work around this. In a pipeline, tee will copy its standard input to its standard output and will also redirect its standard output to the files named as arguments to the command. If data is imagined to

be like water flowing through a pipeline, tee can be visualized as a **"T"** joint in the pipe which directs output in two directions.



Pipeline examples using the tee command

This example will redirect the output of the ls command to the file and will pass it to less to be displayed on the terminal a screen at a time.

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

If tee is used at the end of a pipeline, then the final output of a command can be saved and output to the terminal at the same time.

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

This more sophisticated example takes advantage of the fact that a special device file exists that represents the terminal. The name of the device file for a particular terminal can be determined by running the tty command at its shell prompt. Then tee can be used to redirect output to that file to display it on the terminal window, while standard output can be passed to some other program through a pipe. In this case, mail will e-mail the output to student@desktop1.example.com.

```
[student@desktop ~]$tty
/dev/pts/0
[student@desktop ~]$ ls -l | tee /dev/pts/0 | mailx student@desktop.example.com
```

EDITING FILES WITH VIM

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

Install VIM editor

```
[student@desktop ~]$ yum install -y vim
```

If needed in italian try to find how to with:

```
[student@desktop ~]$ man vimtutor
```

User and Group in Linux

What is a user?

Every process (running program) on the system runs as a particular user.
Every file is owned by a particular user.
Access to files and directories are restricted by user.
The user associated with a running process determines the files and directories accessible to that process.

The **id** command is used to show information about the current logged-in user.

```
[student@desktop ~]$ id
uid=1001(student) gid=1001(students) groups=1001(students),100(users)
```

Basic information about another user can also be requested by passing in the username of that user as the first argument to the **id** command.

```
[student@desktop ~]$ id teacher
```

```
uid=1002(teacher) gid=1002(teachers) groups=1002(teachers),1001(students)
```

To view the user associated with a file or directory, use the **ls - l** command.
The third column shows the username:

```
[student@desktop ~]$ ls -l /tmp/student/
-rw-r--r--. 1 teacher teachers 0 Aug 21 01:12 exam.txt
-rw-r--r--. 1 student students 0 Aug 21 01:11 text.txt
```

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

```
[student@desktop ~]$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
student    30289  0.0  0.1 115840  2508 pts/0    Ss   19:40   0:00 -bash
student    30512  0.0  0.0 155360  1912 pts/0    R+   19:42   0:00 ps au
```

The output of the previous commands displays users by name, but internally the operating system tracks users by a **UID** number. The mapping of names to numbers is defined in databases - of account information. By default, systems use a simple "flat file," the **/etc/passwd** file, to

store information about local users.

```
[student@desktop ~]$ grep student /etc/passwd
1student:2x:31001:41001:5Mario Rossi:6/home/student:7/bin/bash
```

1 is a mapping of a UID to a name for the benefit of human users
2 password is where, historically, passwords were kept in an encrypted format. Today, they are stored in a separate file called /etc/shadow.
3 UID is a user ID, a number that identifies the user at the most fundamental level.
4 GID is the user's primary group ID number. Groups will be discussed in a moment.
5 GECOS field is arbitrary text, which usually includes the user's real name.
6 /home/student is the location of the user's personal data and configuration files.
7 shell is a program that runs as the user logs in. For a regular user, this is normally the program that provides the user's command line prompt.

What is a group?

Like users, groups have a name and a number (GID). Local groups are defined in /etc/group.

Primary group

- Every user has exactly one primary group.
- For local users, the primary group is defined by the GID number of the group listed in the third field of /etc/passwd.
- Normally, the primary group owns new files created by the user.
- Normally, the primary group of a newly created user is a newly created group with the same name as the user. The user is the only member of this User Private Group (UPG).

Supplementary group

Users may be a member of zero or more supplementary groups.

The users that are supplementary members of local groups are listed in the last field of the group's entry in /etc/group. For local groups, user membership is determined by a comma separated list of users found in the last field of the group's entry in /etc/group:

```
[student@desktop ~]$ grep students /etc/group
students:x:1001:teacher,student
```

Supplementary group membership is used to help ensure that users have access permissions to files and other resources on the system.

LAB - Guided Exercise

Explore characteristics of the current user login environment.

-- View the user and group information and display the current working directory

```
[student@desktop ~]$ id
uid=1001(student) gid=1001(students) groups=1001(students),100(users)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[student@desktop ~]$ pwd
/home/student
```

-- View the variables which specify the home directory and the locations searched for executable files.

```
[student@desktop ~]$ echo $HOME
/home/student
[student@desktop ~]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/student/.local/bin:/
home/student/bin
```

Gaining Superuser Access

The **root** user

Most operating systems have some sort of superuser, a user that has all power over the system. This user in Linux 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.
In order to perform tasks such as installing or removing software and to manage system files and directories, a user must escalate privileges to the root user.
Most devices can only be controlled by root, but there are a few exceptions. For instance, removable devices, such as USB devices, are allowed to be controlled by a normal user. Thus, a non-root user is allowed to add and remove files and otherwise manage a removable device, but only root is allowed to manage "fixed" hard drives by default.
This unlimited privilege, however, comes with responsibility. root has unlimited power to damage the system: remove files and directories, remove user accounts, add backdoors, etc.
If the root account is compromised, someone else would have administrative control of the system. The root account on Linux is roughly equivalent to the local Administrator account on Windows.
In Linux, most system administrators log into an unprivileged user account and use various tools to temporarily gain root privileges.

Switching users with su

The su command allows a user to switch to a different user account. If a username is not specified, the root account is implied. When invoked as a regular user, a prompt will display asking for the password of the account you are switching to; when invoked as root, there is no need to enter the account password.

```
su [ - ] <username>
```

```
[student@desktop ~]$ su -
Password: ########
Last login: Fri Aug 23 15:53:45 CEST 2019 on pts/0
```

The command su username starts a non-login shell, while the command su -
starts a login shell. The main distinction is su - sets up the shell environment as if this were a clean login as that user, while su just starts a shell as that user with the current environment settings.
In most cases, administrators want to run su - to get the user's normal settings. For more information, see the bash(1) man page.

Running commands as root with sudo

Fundamentally, Linux implements a very coarse-grained permissions model: root can do everything, other users can do nothing (systems-related). The common solution previously discussed is to allow standard users to temporarily "become root" using the su command. The disadvantage is that while acting as root, all the privileges (and responsibilities) of root are granted. Not only can the user restart the web server, but they can also remove the entire /etc directory. Additionally, all users requiring superuser privilege in this manner must know the root password.
The **sudo** command allows a user to be permitted to run a command as root, or as another user, based on settings in the **/etc/sudoers** file. Unlike other tools such as **su**, **sudo** requires users to enter their own password for authentication, not the password of the account they are trying to access. This allows an administrator to hand out fine-grained permissions to users to delegate system administration tasks, without having to hand out the root password.

For example, when **sudo** has been configured to allow the user student to run the command **usermod** as root, student could run the following command to lock a user account:

```
[student@desktop ~]$ sudo usermod -L username
```

```
[sudo] password for student: #######
```

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

Add string "`student ALL = NOPASSWD: /bin/tail /var/log/secure`" to **/etc/sudoers** file as user root.

```
[student@desktop ~]$ sudo tail /var/log/secure
Aug 24 18:06:37 desktop sudo: student : TTY=pts/0 ; PWD=/home/student ; USER=root
; COMMAND=/bin/tail /var/log/secure
Aug 24 18:06:37 desktop sudo: pam_unix(sudo:session): session opened for user
root by student(uid=0)
```

All members of group **wheel** can use **sudo** to run commands as any user, including root. The user will be prompted for their own password.

LAB - Running Commands as root

Outcomes

Use the **su** with and without login scripts to switch users. Use **sudo** to run commands with privilege.

Log into the system as **student** with a password of student.

1. Explore characteristics of the current student login environment.
    a. View the user and group information and display the current working directory.

```
[student@desktop ~]$ id
uid=1001(student) gid=1001(students) gruppi=1001(students),100(users)
contesto=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[student@desktop ~]$ pwd
/home/student
```

b. View the variables which specify the home directory and the locations searched for executable files.

```
[student@desktop ~]$ echo $HOME
/home/student
[student@desktop ~]$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/student/.local/bi
n:/home/student/bin
```

2. Switch to root without the dash and explore characteristics of the new environment.
   a. Become the root user at the shell prompt.

```
[student@desktop ~]$ su
Password:
```

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

```
[root@desktop student]# id
uid=0(root) gid=0(root) gruppi=0(root)
[root@desktop student]# pwd
/home/student
```

c. View the variables which specify the home directory and the locations searched for executable files. Look for references to the student and root accounts.

```
[root@desktop student]# echo $HOME
/root
[root@desktop student]# echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/student/.local/bi
n:/home/student/bin
```

d. Exit the shell to return to **student** user.

```
[root@desktop student]# exit
exit
[student@desktop ~]$
```

3. Switch to root with the dash and explore characteristics of the new environment.
   a. Become the root user at the shell prompt. Be sure all the login scripts are also executed.

```
[student@desktop ~]$ su -
Password:
```

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

```
[root@desktop ~]# id
uid=0(root) gid=0(root) groups=0(root)
[root@desktop ~]# pwd
/root
```

c. View the variables which specify the home directory and the locations searched for executable files. Look for references to the student and root accounts.

```
[root@desktop ~]# echo $HOME
/root
[root@desktop ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
```

d. Exit the shell to return to **student** user.

```
[root@desktop student]# exit
exit
[student@desktop ~]$
```

4. Run several commands as student which require root access.
    a. View the last lines of the /var/log/messages with **tail** command

```
[student@desktop ~]$ tail /var/log/messages
tail: impossibile aprire "/var/log/messages" per la lettura: Permesso
negato
```

    b. Switch user to root

```
[student@desktop ~]$ su - root
Password:
```

    c. Add to /etc/sudoers “student ALL = NOPASSWD: /sbin/usermod, /bin/tail /var/log/messages”

```
vim /etc/sudoers
## Allows members of the users group to mount and unmount the
## cdrom as root
# %users  ALL=/sbin/mount /mnt/cdrom, /sbin/umount /mnt/cdrom

student ALL = NOPASSWD: /sbin/usermod, /bin/tail /var/log/messages
```

```
## Allows members of the users group to shutdown this system
# %users  localhost=/sbin/shutdown -h now
```

d. Switch back to student user

```
[root@desktop ~]# exit
logout
[student@desktop ~]$
```

e. View the last lines of the /var/log/messages with **sudo tail** command

```
[student@desktop ~]$ sudo tail /var/log/messages
Aug 24 18:49:17 desktop dbus[6706]: [system] Successfully activated
service 'org.freedesktop.nm_dispatcher'
Aug 24 18:53:54 desktop su: (to root) student on pts/0
```

Managing Local User Accounts

Managing local users

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

useradd creates users

useradd username sets reasonable defaults for all fields in **/etc/passwd** when run without options.
The **useradd** command does not set any valid password by default, and the user cannot log in until a password is set.
• **useradd** - -help will display the basic options that can be used to override the defaults.
In most cases, the same options can be used with the **usermod** command to modify an existing user.
• Some defaults, such as the range of valid **UID** numbers and default password aging rules, are read from the **/etc/login** . defs file. Values in this file are only used when creating new users. A change to this file will not have an effect on any existing users.

usermod modifies existing users

usermod - -help will display the basic options that can be used to modify an account. Some common options include:

-c, - -comment COMMENT	Add a value, such as a full name, to the GECOS field.
-g, - -gid GROUP	Specify the primary group for the user account.
-G, - -groups GROUPS	Specify a list of supplementary groups for the user account.
-a, - -append	Used with the -G option to append the user to the supplemental groups mentioned without removing the user from other groups.
-d, - -home HOME_DIR	Specify a new home directory for the user account.
-m, - -move-home	Move a user home directory to a new location. Must be used. with the - d option.
-s, --shell SHELL	Specify a new login shell fort heuseraccount.
-L, --lock	Lock a user account.

-U, - -unlock	Unlock a user account.

userdel deletes users

userdel username removes the user from **/etc/passwd**, but leaves the home directory intact by default.
userdel - r username removes the user and the user's home directory.

id displays user information

id will display user information, including the user's UID number and group memberships.
id username will display user information for username, including the user's UID number

passwd sets passwords

passwd username can be used to either set the user's initial password or change that user's password.

Change password as root

```
[root@desktop ~]# passwd student
Changing password for user student.
New password: student
BAD PASSWORD: The password is shorter than 8 characters
Retype new password: student
passwd: all authentication tokens updated successfully.
```

The **root** user can set a password to any value. A message will be displayed if the password does not meet the minimum recommended criteria, but is followed by a prompt to retype the new password and all tokens are updated successfully.

Change password as student

```
[student@desktop ~]$ passwd
Cambio password per l'utente student.
Cambio password per student.
Password UNIX (corrente): student
Nuova password: goofy
PASSWORD ERRATA: La password è più corta di 8 caratteri
```

A regular user must choose a password which is at least 8 characters in length and is not based on a dictionary word, the username, or the previous password.

UID ranges

Specific UID numbers and ranges of numbers are used for specific purposes

- 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 Liniux.
- UID 201-999 is a range of "system users" used by system processes that do not own files on the file system. They are typically assigned dynamically from the available pool when the software that needs them is installed. Programs run as these "unprivileged" system users in order to limit their access to just the resources they need to function.
- UID 1000+ is the range available for assignment to regular users.

LAB - Creating Users Using Command-line Tools

1. Become the **root** user at the shell prompt.
```
[student@desktop ~]$ su -
Password:
```

2. Add the user juliet.
```
[root@desktop ~]# useradd juliet
```

3. Changing password for user juliet.
```
[root@desktop ~]# passwd juliet
New password: juliet
BAD PASSWORD: The password is shorter than 8 characters
Retype new password: juliet
passwd: all authentication tokens updated successfully.
```

4. Confirm that juliet has been added by examining the **/etc/passwd** file and **/etc/shadow** file.
```
[root@desktop ~]# grep juliet /etc/passwd
juliet:x:1003:1003::/home/juliet:/bin/bash
[root@desktop ~]# grep juliet /etc/shadow
```

```
juliet:$6$ncIV78id$tKzpm/aYfRB9xh9q3hIHq2yZUJ5SExylOKJGnJ5psfrTQ54.WKQs.n0J4.
4mejb/sqcteBs8HSTE9dNfj/p5T0:18133:0:99999:7:::
```

5. Continue adding the remaining users in the steps below and set initial passwords.
   a. romeo

```
[root@desktop ~]# useradd romeo
[root@desktop ~]# passwd romeo
Changing password for user romeo.
New password: romeo
BAD PASSWORD: The password is shorter than 8 characters
Retype new password: hamlet
passwd: all authentication tokens updated successfully.
```

   b. hamlet

```
[root@desktop ~]# useradd hamlet
[root@desktop ~]# passwd hamlet
```

   c. reba

```
[root@desktop ~]# useradd hamlet
[root@desktop ~]# passwd hamlet
```

d. dolly

```
[root@desktop ~]# useradd dolly
[root@desktop ~]# passwd dolly
```

e. elvis

```
[root@desktop ~]# useradd elvis
[root@desktop ~]# passwd elvis
```

Managing Local Group Accounts

Managing supplementary groups

A group must exist before a user can be added to that group. Several command-line tools are used to manage local group accounts.

groupadd creates groups

groupadd groupname without options uses the next available GID from the range specified
in the **/etc/login** . defs file.
The **-g GID** option is used to specify a specific **GID**.

```
[root@desktop ~]# groupadd -g 5000 ateam
```

Given the automatic creation of user private groups (**GID 1000+**), it is generally recommended to set aside a range of **GID** numbers to be used for supplementary groups. A higher range will avoid a collision with a system group (**GID 0-999**).

The - r option will create a system group using a GID from the range of valid system GID numbers listed in the /etc/login . defs file.

```
[root@desktop ~]# groupadd -r appusers
```

groupmod modifies existing groups

The groupmod command is used to change a group name to a GID mapping. The -n option is used to specify a new name.

```
[root@desktop ~]# groupmod -r javaapps appusers
```

The -g option is used to specify a new GID

```
[root@desktop ~]# groupmod -g 6000 ateam
```

groupdel delete group

```
[root@desktop ~]# groupdel javaapps
```

A group may not be removed if it is the primary group of any existing user. As with userdel, check all file systems to ensure that no files remain owned by the group.

usermod alters group membership

The membership of a group is controlled with user management. Change a user's primary group with **usermod** -g groupname.

```
[root@desktop ~]# usermod -g ateam romeo
```

Add a user to a supplementary group with **usermod -aG** groupname username.

```
[root@desktop ~]# usermod -aG wheel romeo
```

LAB - Managing Groups Using Command line Tools

In this lab, you will add users to newly created supplementary groups.

Outcomes
The shakespeare group consists of juliet, romeo, and hamlet. The artists group contains reba, dolly, and elvis.

1. Become the root user at the shell prompt.
   ```
   [student@desktop ~]$ su -
   Password:
   ```

2. Create a supplementary group called shakespeare with a group ID of 30000.
   ```
   [root@desktop ~]# groupadd -g 30000 shakespeare
   ```

3. Create a supplementary group called artists
   ```
   [root@desktop ~]# groupadd artists
   ```

4. Confirm that shakespeare and artists have been added by examining the /etc/group file.
   ```
   [root@desktop ~]# tail -2 /etc/group
   shakespeare:x:30000:
   artists:x:30001:
   ```

5. Add the juliet user to the shakespeare group as a supplementary group.

```
[root@desktop ~]# usermod -G shakespeare juliet
```

6. Confirm that juliet has been added using the id command.

```
[root@desktop ~]# id juliet
```

7. Continue adding the remaining user to group as follow:
    a. add romeo and hamlet to shakespeare group

```
[root@desktop ~]# usermod -G shakespeare romeo
[root@desktop ~]# usermod -G shakespeare hamlet
```

    b. Add reba, dolly and elvis to the artists group

```
[root@desktop ~]# usermod -G artists reba
[root@desktop ~]# usermod -G artists dolly
[root@desktop ~]# usermod -G artists elvis
```

    c. Verify the supplemental group memberships by examining the /etc/group file.

```
[root@desktop ~]# tail -2 /etc/group
```

Managing user Password

Shadow passwords and password policy

There are three pieces of information stored in a modern password hash:

1gCjLa2/Z$6Pu0EK0AzfCjxjv2hoLOB/

1. **1:** The hashing algorithm. The number 1 indicates an MD5 hash. The number 6 appears when a SHA-512 hash is used.
2. **gCjLa2/Z**: The salt used to encrypt the hash. This is originally chosen at random. The salt and the unencrypted password are combined and encrypted to create the encrypted password hash. The use of a salt prevents two users with the same password from having identical entries in the /etc/shadow file.
3. **6Pu0EK0AzfCjxjv2hoLOB/**: The encrypted hash.

When a user tries to log in, the system looks up the entry for the user in /etc/shadow, combines the salt for the user with the unencrypted password that was typed in, and encrypts them using the hashing algorithm specified. If the result matches the encrypted hash, the user typed in the right password. If the result doesn't match the

encrypted hash, the user typed in the wrong password and the login attempt fails. This method allows the system to determine if the user typed in the correct password without storing that password in a form usable for logging in.

/etc/shadow format

The format of **/etc/shadow** follows (nine colon-separated fields):

[1]name:**[2]password**:**[3]astchange**:**[4]minage**:**[5]maxage**:**[6]warning**:**[7]inactive**:**[8]expire**:**[9]b1ank**

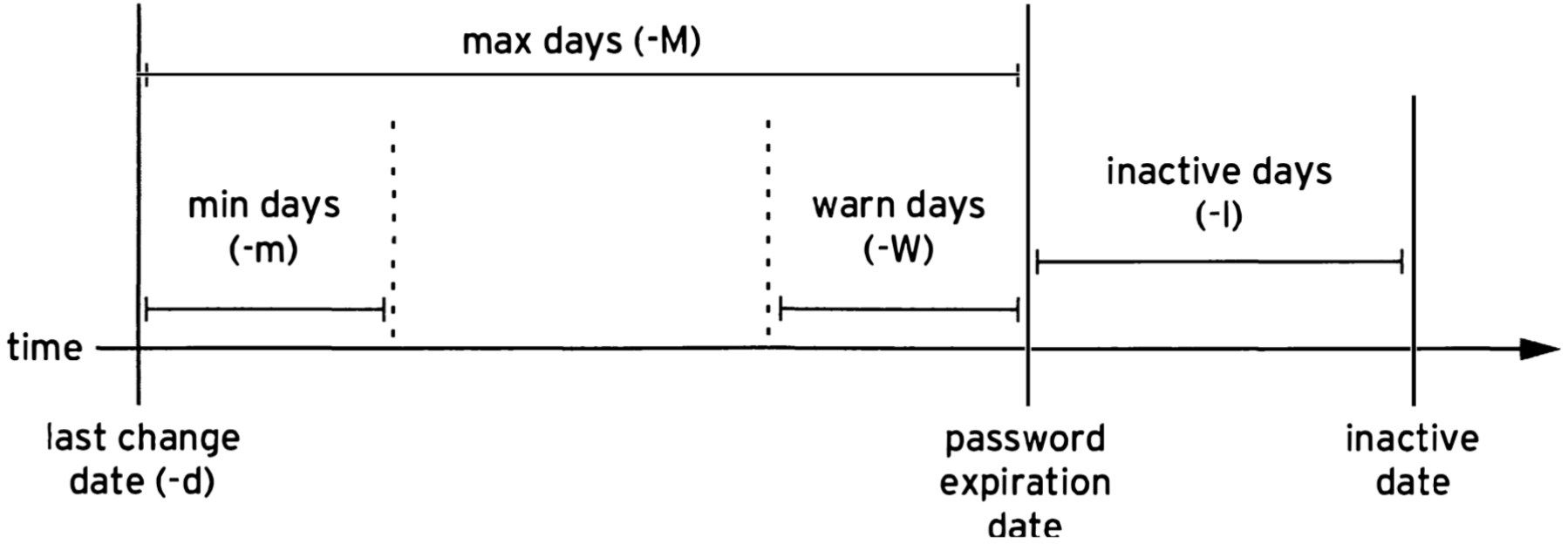
```
[root@desktop ~]# grep student /etc/shadow
student:$6$f8F43V2D$TbnKZVvEWuCKed8NTXdPBjljcurZk0ghVmwfTci0QsW2BrT1Ofmw33V3QYjO0
X53Zna3x22Xxf/7yFDr7WBXy1:18133:0:99999:7:::
```

1. The login name. This must be a valid account name on the system.
2. The encrypted password. A password field which starts with a exclamation mark means that the password is locked.
3. The date of the last password change, represented as the number of days since 1970.01.01.
4. The minimum number of days before a password may be changed, where O means "no minimum age requirement."

5. The maximum number of days before a password must be changed.
6. The warning period that a password is about to expire. Represented in days, where 0 means "no warning given."
7. The number of days an account remains active after a password has expired. A user may still log into the system and change the password during this period. After the specified number of days, the account is locked, becoming inactive.
8. The account expiration date, represented as the number of days since 1970.01.01.
9. This blank field is reserved for future use.

Password aging

The following diagram relates the relevant password-aging parameters, which can be adjusted using **chage** to implement a password-aging policy.



 # chage -m 0 -M 90 -W 7 -I 14 username

chage -d 0 username will force a password update on next login.

chage -l username will list a username's current settings.

chage -E YY-MM-DD will expire an account on a specific day.

The **date** command can be used to calculate a date in the future.

```
[root@desktop ~]# date -d "+45 days"
Wed Oct  9 22:54:47 CEST 2019
```

LAB - Managing User Password Aging

Outcomes

The password for romeo must be changed when the user first logs into the system, every 90 days thereafter, and the account expires in 180 days.

1. Lock the **romeo** account

```
[root@desktop ~]# usermod -L romeo
[root@desktop ~]# exit
logout
[student@desktop ~]$ su - romeo
Password: romeo
su: Autenticazione fallita
[student@desktop ~]$ su -
Password:
[root@desktop ~]# usermod -U romeo
[root@desktop ~]# exit
logout
[student@desktop ~]$ su - romeo
Password: romeo
```

```
[romeo@desktop ~]$
```

2. Change the password policy for romeo to require a new password every 90 days.

```
[root@desktop ~]# chage -M 90 romeo
[root@desktop ~]# chage -l romeo
Last password change                                    : Aug 25, 2019
Password expires                                        : Nov 23, 2019
Password inactive                                       : never
Account expires                                         : never
Minimum number of days between password change          : 0
Maximum number of days between password change          : 90
Number of days of warning before password expires       : 7
```

3. Additionally, force a password change on the first login for the romeo account.

```
[root@desktop ~]# chage -d 0 romeo
[root@desktop ~]# exit
logout
[student@desktop ~]$ su - romeo
Password: romeo
È richiesta la modifica immediata della password (imposto
dall'amministratore)
Cambio password per romeo.
```

```
Password UNIX (corrente): romeo
Nuova password: 45hbf9R@!
Reimmettere la nuova password: 45hbf9R@!
[romeo@desktop ~]$ exit
logout
[student@desktop ~]$
```

4. Expire accounts in the future.
    a. Determine a date 180 days in the future.

```
[root@desktop ~]# date -d "+180 days"
Fri Feb 21 22:12:50 CET 2020
```

    b. Set accounts to expire on that date.

```
[root@desktop ~]# chage -E 2020-02-21 romeo
[root@desktop ~]# chage -l romeo
Last password change                    : Aug 25, 2019
Password expires                        : Nov 23, 2019
Password inactive                       : never
Account expires                         : Feb 21, 2020
Minimum number of days between password change    : 0
Maximum number of days between password change    : 90
Number of days of warning before password expires : 7
```

LAB - Managing local Linux Users and Groups

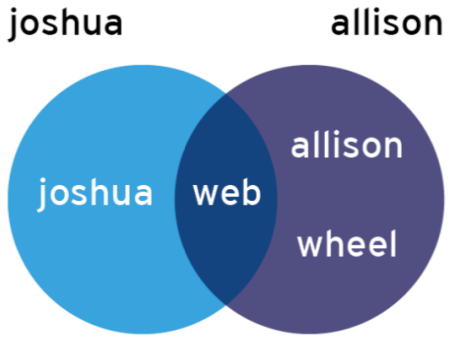
Outcome:

A new group called **consultants** including three new users : **Sam Spade,Betty Boop and Dick Tracy**.
All new account should required that password be changed at first login and every 30 days thereafter.
The consultants accounts should expire at the end of the 90-day contract, and **Betty Boop** must change her password every 15 days.

Before you begin.
Ensure that newly created users have password which must be changed every 30 days
Create new group named **consultants** with a **GID** of **40000**
create three new users: **sspade**, **bboop** and **dtracy**, with password of default and add them to the supplementary group consultants. The primary group should remain as the user private group.
Determinate the date 90 days in the future and set each of the three accounts to expire on that date.
Change the password policy for **bboop** account to require a new password every 15 days.
Force all new user to change their password on first login

Linux File System Permissions

Access to files by users are controlled by file permissions.
Files have just three categories of user to which permissions apply. The file is owned by a user, normally the one who created the file. The file is also owned by a single group, usually the primary group of the user who created the file, but this can be changed. Different permissions can be set for the owning user, the owning group, and for all other users on the system that are not the user or a member of the owning group.
The most specific permissions apply. So, user permissions override group permissions, which override other permissions.
In the example image, joshua is a member of the groups joshua and web, while allison is a member of allison, wheel, and web. When joshua and allison have the need to collaborate, the files should be associated with the group web and the group permissions should allow the desired access.



There are also just three categories of permissions which apply: read, write, and execute. These permissions affect access to files and directories as follows:

Effects of permissions on files and directories

Permission	Effect on files	Effect on directories
r (read)	Content on the file can be read	Contents of the directory (file names) can be listed.
w (write)	Content on the file can be changed	Any file in the directory may be created or deleted.
x (execute)	File can be executed as commands	Contents of the directory can be accessed (dependent on the permissions of the files in the directory).

Note that users normally have both **read** and **exec** on read-only directories, so that they can list the directory and access 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 **exec** access on a directory, they cannot

list the names of the files in the directory, but if they already know the name of a file which they have permission to read, then they can access the contents of that file by explicitly specifying the file name.
A file may be removed by anyone who has 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, which will be discussed at the end of the unit.).

Viewing file/directory permissions and ownership

The **-l** option of the **ls** command will expand the file listing to include both the permissions of a file and the ownership:U

```
[root@desktop ~]# ls -la test
-rw-r--r--. 1 root root 0  3 set 18.16 test
```

The command **ls - l** directory name will show the expanded listing of all of the files that reside inside the directory. To prevent the descent into the directory and see the expanded listing of the directory itself, add the **-d** option to ls:

```
[student@desktop ~]$ ls -ld /home
drwxr-xr-x. 6 root root 61 Aug 25 23:03 /home
```

Examples: Linux user, group, other concepts

Users and their groups:

```
lucy      lucy,ricardo
ricky     ricky,ricardo
ethel     ethel,mertz
fred      fred,mertz
```

File attributes (permissions, user & group ownership, name):

```
drwxrwxr-x   ricky   ricardo   dir (which contains the following files)
-rw-rw-r--   lucy    lucy      lfile1
-rw-r--rw-   lucy    ricardo   lfile2
-rw-rw-r--   ricky   ricardo   rfile1
-rw-r-----   ricky   ricardo   rfile2
```

lucy is the only person who can change the contents of **lfile1**.
ricky can view the contents of **lfile2**

ricky can delete **lfile2**.
ethel can change the contents of **lfile2**.
lucy can change the contents of **rfile1**.
ricky can view and modify the contents of **rfile2**.
lucy can view but not modify the contents of **rfile2**.
ethel and **fred** do not have any access to the contents of **rfile2**.

Managing File System Permissions from the Command Line

Changing file/directory permissions

The command used to change permissions from the command line is **chmod**, short for "change mode" (permissions are also called the mode of a file). The **chmod** command takes a permission instruction followed by a list of files or directories to change. The permission instruction can be issued either symbolically (the symbolic method) or numerically (the numeric method).

Symbolic method keywords:

chmod WhoWhatWhich file l 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, executable)

The symbolic method of changing file permissions uses letters to represent the different groups of permissions: **u** for user, **g** for group, **o** for other, and a for all.
With the symbolic method, it is not necessary to set a complete new group of permissions. Instead, it is possible to change one or more of the existing permissions. In order to accomplish this, use three symbols: **+** to add permissions to a set, **-** to remove permissions from a set, and **=** to replace the entire set for a group of permissions.

The permissions themselves are represented by a single letter: **r** for read, **w** for write, and **x** for execute.

Numeric method:

chmod ### file|directory

Each digit represents an access level: **user**, **group**, **other**
is sum of r=4, w=2, and x=1

Using the numeric method, permissions are represented by a three-digit (or four, when setting advanced permissions) octal number. A single octal digit can represent the numbers **0-7**, exactly the number of possibilities for a three-bit number.

To convert between symbolic and numeric representation of permissions, we need to know how the mapping is done. In the three-digit octal (numeric) representation, each digit stands for one group of permissions, from left to right: user, group, and other. In each of these groups, start with 0. If the read permission is present, add **4**. Add **2** if write is present, and **1** for execute.

Numeric permissions are often used by advanced administrators since they are shorter to type and pronounce, while still giving full control over all permissions.

Examine the permissions **-rwxr-x---**. For the user, rwx is calculated as **4+2+1=7**. For the group, **r-x** is calculated as **4+0+1=5**, and for other users, **---** is represented with **0**. Putting these three together, the numeric representation of those permissions is **750**.

This calculation can also be performed in the opposite direction. Look at the permissions **640**. For the user permissions, **6** represents read (4) and write (2), which displays as rw-. For the group part, **4** only includes read (4) and displays as **r--**. The **0** for other provides no permissions (---) and the final set of symbolic permissions for this file is **-rw-r-----**.

Examples

Remove read and write permission for group and other on file1:

```
[student@desktop ~]$ ls -la test
-rwxrwxrwx. 1 student students 0 Sep  3 21:59 test
[student@desktop ~]$ chmod go -rw test
chmod: cannot access ‘go’: No such file or directory
chmod: test: new permissions are --x-wx-wx, not --x--x--x
[student@desktop ~]$ ls -la test
---x--x-wx. 1 student students 0 Sep  3 21:59 test
```

Remove execute permission for everyone on **file2**

```
[student@desktop ~]$ chmod a-x file1
[student@desktop ~]$ ls -la file1
-----w--w-. 1 student students 0 Sep  3 21:59 file1
```

Setread, write, and execute permission for user, read, and execute for group, and no permission for other on studentdir:

```
[student@desktop ~]$ chmod 750 studentdir
[student@desktop ~]$ ls -lad
drwxr-x---. 2 student students 6 Sep  3 21:37 studentdir
```

N.B. The **chmod** command supports the **-R** option to recursively set permissions on the files in an entire directory tree

```
[student@desktop ~]$ ls -la studentdir/
total 0
drwxr-x---. 2 student students  45 Sep  4 10:44 .
drwx------. 3 student students 130 Sep  4 10:37 ..
-rw-r--r--. 1 student students   0 Sep  4 10:44 file1
-rw-r--r--. 1 student students   0 Sep  4 10:44 file2
-rw-r--r--. 1 student students   0 Sep  4 10:44 file3
[student@desktop ~]$ chmod -R ugo=rwx studentdir/ ###chmod 777 studentdir
[student@desktop ~]$ ls -la studentdir/
total 0
drwxrwxrwx. 2 student students  45 Sep  4 10:44 .
drwx------. 3 student students 130 Sep  4 10:37 ..
-rwxrwxrwx. 1 student students   0 Sep  4 10:44 file1
```

```
-rwxrwxrwx. 1 student students   0 Sep  4 10:44 file2
-rwxrwxrwx. 1 student students   0 Sep  4 10:44 file3
```

CHANGING FILE/DIRECTORY USER OR GROUP OWNERSHIP

A newly created file is owned by the user who creates the file.
By default, the new file has a group ownership which is the primary group of the user creating the file. Since Linux uses user private groups, this group is often a group with only that user as a member.
To grant access based on group membership, the owner or the group of a file may need to be changed.
File ownership can be changed with the **chown** command .
For example, to grant ownership of the file foofile to student, the following command could be used:

```
[root@desktop ~]# chown student foofile
```

chown can be used with the **-R** option to recursively change the ownership of an entire directory tree.
The following command would grant ownership of foodir and all files and subdirectories within it to student:

```
[root@desktop ~]# chown -R student foodir
```

The **chown** command can also be used to change group ownership of a file by preceding the group name with a colon **(:)**. For example, the following command will change the group foodir to admins:

```
[root@desktop ~]# chown -R :admins foodir
```

The **chown** command can also be used to change both **owner** and **group** at the same time by using the syntax **owner:group**. For example, to change the ownership of foodir to visitor and the group to guests, use:

```
[root@desktop ~]# chown -R visitor:guest foodir
```

Only **root** can change the ownership of a file. Group ownership, however, can be set by root or the file's owner. root can grant ownership to any group, while non-root users can grant ownership only to groups they belong to.

LAB: MANAGING FILE SECURITY FROM THE COMMAND LINE

1. Become the root user at the shell prompt.

```
[student@desktop ~]$ su -
Password:
```

2. Create group **disney**

```
[root@desktop ~]# groupadd disney
[root@desktop ~]# grep disney /etc/group
disney:x:1004:
[root@desktop ~]#
```

3. Create users minnie and goofy with password=password and add secondary group disney to users

```
[root@desktop ~]# useradd minnie
[root@desktop ~]# passwd minnie ##set password to “password”
[root@desktop ~]# useradd goofy
[root@desktop ~]# passwd goofy ##set password to “password”
[root@desktop ~]# usermod minnie -G minnie,disney
[root@desktop ~]# id minnie
uid=1004(minnie) gid=1005(minnie) groups=1005(minnie),1004(disney)
[root@desktop ~]# usermod goofy -G goofy,disney
[root@desktop ~]# id goofy
uid=1005(goofy) gid=1006(goofy) groups=1006(goofy),1004(disney)
```

4. Create directory /home/disney-team

```
[root@desktop ~]# mkdir /home/disney-team
```

5. Change the group ownership of the disney-team directory to disney

```
[root@desktop ~]# chown :disney /home/disney-team
```

6. Ensure the permissions of /home/disney-team allows group members to create and delete files.

```
[root@desktop ~]# chmod g+w /home/disney-team/
```

7. Ensure the permissions of disney-team forbids others from accessing its files.

```
[root@desktop ~]# chmod o-rx /home/disney-team/
[root@desktop ~]# ls -lad /home/disney-team
drwxrwx---. 2 root disney 6 Sep  4 17:19 /home/disney-team
```

8. Exit the root shell and switch to the user minnie with password of password.

```
[root@desktop ~]# exit
logout
[student@desktop ~]$ su - minnie
Password:
```

9. Navigate to the /home/disney-team and create an empty file called minniefile

```
[student@desktop ~]$ su - minnie
Password:
[minnie@desktop ~]$ cd /home/disney-team/
[minnie@desktop disney-team]$ touch minniefile
[minnie@desktop disney-team]$ ls -la
total 0
drwxrwx---. 2 root   disney  24 Sep  4 17:36 .
drwxr-xr-x. 9 root   root   107 Sep  4 17:19 ..
-rw-rw-r--. 1 minnie minnie   0 Sep  4 17:36 minniefile
```

10. Change the group ownership of the new file to disney

```
[minnie@desktop disney-team]$ chown :disney minniefile
[minnie@desktop disney-team]$ ls -la minniefile
-rw-rw-r--. 1 minnie disney 0 Sep  4 17:36 minniefile
```

11. Exit the shell and switch to the user goofy with a password of password.

```
[minnie@desktop disney-team]$ exit
logout
[student@desktop ~]$ su - goofy
Password:
```

12. Navigate to the /home/disney folder; Determine minnie's privileges to access and/or modify minniefile

```
[goofy@desktop ~]$ cd /home/disney-team/
[goofy@desktop disney-team]$ ls -la minniefile
-rw-rw-r--. 1 minnie disney 0 Sep  4 17:36 minniefile
[goofy@desktop disney-team]$ echo "hello" >> minniefile
[goofy@desktop disney-team]$ cat minniefile
hello
```

MANAGING DEFAULT PERMISSIONS AND FILE ACCESS

SPECIAL PERMISSIONS

The setuid (or setgid) permission on an executable file means that the command will run as the user (or group) of the file, not as the user that ran the command. One example is the passwd command:

```
[student@desktop ~]$ ls -la /usr/bin/passwd
-rwsr-xr-x. 1 root root 27832 Jan 30  2014 /usr/bin/passwd
```

In a long listing, you can spot the setuid permissions by a lowercase s where you would normally expect the x (owner execute permissions) to be. If the owner does not have execute permissions, this will be replaced by an uppercase S.
The sticky bit for a directory sets a special restriction on deletion of files: Only the owner of the file (and root) can delete files within the directory. An example is /tmp:

```
[student@desktop ~]$ ls -ld /tmp/
drwxrwxrwt. 9 root root 154 Sep  6 13:41 /tmp/
```

In a long listing, you can spot the sticky permissions by a lowercase t where you would normally expect the x (other execute permissions) to be. If the other does not have execute permissions, this will be replaced by an uppercase T.

Lastly, setgid on a directory means that files created in the directory will inherit the group affiliation 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.
In a long listing, you can spot the setgid permissions by a lowercase s where you would normally expect the x (group execute permissions) to be. If the group does not have execute permissions, this will be replaced by an uppercase S.

Effects of special permissions on files and directories

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

Setting special permissions

- Symbolically: setuid = u+s; setgid = g+s; sticky = o+t
- Numerically(fourthprecedingdigit):setuid=4;setgid=2;sticky=1

Examples

- Add the setgid bit on directory:

```
[student@desktop ~]$ ls -ld directory/
drwxr-sr-x. 2 student students 6 Sep  6 13:50 directory/
```

Set the setgid bit, and read/write/execute for user and group on directory:

```
[student@desktop ~]$ chmod 2770 directory
[student@desktop ~]$ ls -ld directory/
drwxrws---. 2 student students 6 Sep  6 13:50 directory/
```

DEFAULT FILE PERMISSIONS

The default permissions for files are set by the processes that create them. For example, text editors create files so they are readable and writeable, but not executable, by everyone. The same goes for shell redirection. Additionally, binary executables are created executable by the compilers that create them. The **mkdir** command creates new directories with all permissions set **read**, **write**, and **execute**.
Experience shows that these permissions are not typically set when new files and directories are created. This is because some of the permissions are cleared by the **umask** of the shell process. The **umask** command without arguments will display the current value of the shell's umask:

```
[student@desktop ~]$ umask
0022
```

Every process on the system has a umask, which is an octal bitmask that is used to clear the permissions of new files and directories that are created by the process. If a bit is set in the umask, then the corresponding permission is cleared in new files. For example, the previous umask, 0002, clears the write bit for other users. The leading zeros indicate the special, user, and group permissions are not cleared. A umask of 077 clears all the group and other permissions of newly created files.
Use the **umask** command with a single numeric argument to change the umask of the current shell. The numeric argument should be an octal value corresponding to the new umask value. If it is less than 3 digits, leading zeros are assumed.

The system default umask values for Bash shell users are defined in the **/etc/profile** and **/etc/bashrc** files.
Users can override the system defaults in their **.bash_profile** and **.bashrc files.**
In this example, please follow along with the next steps while your instructor demonstrates the effects of umask on new files and directories.

Create a new file and directory to see how the default **umask** affects permissions.

```
[student@desktop ~]$ touch newfile
[student@desktop ~]$ ls -la newfile
-rw-r--r--. 1 student students 0 Sep  4 17:49 newfile
[student@desktop ~]$ mkdir newdir
[student@desktop ~]$ ls -lad newdir/
drwxr-xr-x. 2 student students 6 Sep  4 17:49 newdir/
```

Set the **umask** value to 0.This setting will not mask any of the permissions of new files.
Create a new file and directory to see how this new umask affects permissions.

```
[student@desktop ~]$ umask 0
[student@desktop ~]$ touch newfile2
[student@desktop ~]$ ls -la newfile2
-rw-rw-rw-. 1 student students 0 Sep  4 17:50 newfile2
[student@desktop ~]$ mkdir newdir2
```

```
[student@desktop ~]$ ls -lad newdir2
drwxrwxrwx. 2 student students 6 Sep  4 17:51 newdir2
```

Set the **umask** value to 007. This setting will mask all of the “other” permissions of new files

```
[student@desktop ~]$ umask 007
[student@desktop ~]$ touch newfile3
[student@desktop ~]$ ls -la newfile3
-rw-rw----. 1 student students 0 Sep  4 17:52 newfile3
[student@desktop ~]$ mkdir newdir3
[student@desktop ~]$ ls -lad newdir/
drwxr-xr-x. 2 student students 6 Sep  4 17:49 newdir/
```

Set the **umask** value to 027. This setting will mask write access for group members and all of the “other” permissions of new files.

```
[student@desktop ~]$ umask 027
[student@desktop ~]$ touch newfile4
[student@desktop ~]$ ls -la newfile4
-rw-r-----. 1 student students 0 Sep  4 17:54 newfile4
[student@desktop ~]$ mkdir newdir4
[student@desktop ~]$ ls -lad newdir4
```

```
drwxr-x---. 2 student students 6 Sep  4 17:54 newdir4
```

Log in as **root** to change the default umask for unprivileged users to prohibit all access for users not in their group.
Modify **/etc/bashrc** and **/etc/profile** to change the default umask for Bash shell users. Since the default umask for unprivileged users is 0002, look for the umask command in these files that sets the umask to that value. Change them to set the umask to 007.

```
[root@desktop ~]# less /etc/bashrc
# You could check uidgid reservation validity in
# /usr/share/doc/setup-*/uidgid file
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
        umask 002
    else
umask 022 fi
# Only display echos from profile.d scripts if we are no login shell

[root@desktop ~]# vim /etc/bashrc
[root@desktop ~]# less /etc/bashrc
    # You could check uidgid reservation validity in
    # /usr/share/doc/setup-*/uidgid file
    if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
```

```
umask 007
    else
       umask 022
fi
# Only display echos from profile.d scripts if we are no login shell

[root@desktop ~]# less /etc/profile
    # You could check uidgid reservation validity in
    # /usr/share/doc/setup-*/uidgid file
    if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
        umask 002
    else
umask 022 fi
for i in /etc/profile.d/*.sh ; do
[root@desktopX ~]# vim /etc/profile

[root@desktopX ~]# vim /etc/profile
[root@desktopX ~]# less /etc/profile
    # You could check uidgid reservation validity in
    # /usr/share/doc/setup-*/uidgid file
    if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
umask 007
```

```
    else
        umask 022
    fi
    for i in /etc/profile.d/*.sh ; do
```

Log back in as student and confirm that the umask changes you made are persistent.

```
[root@desktop ~]# exit
logout
[student@desktop ~]$ umask
0002
```

LAB: CONTROLLING ACCESS TO FILES WITH LINUX FILE SYSTEM PERMISSIONS

OUTCOMES

- A directory on server called **/home/stooges** where these three users can work collaboratively on files.
- Only the user and group access, create, and delete files in **/home/stooges**. Files created in this directory should automatically be assigned a group ownership of stooges.
- New files created by users will not be accessible outside of the group.

1. Create three new account **curly**, **larry**, and **moe**, who are members of a group called **stooges**. The password for each account is **password**.
2. become **root** on server.
3. Create the **/home/stooges** directory.
4. Change group permissions on the **/home/stooges** directory so it belongs to the **stooges** group.
5. Set permissions on the **/home/stooges** directory so it is a set GID bit directory **(2)**, the owner **(7)** and group **(7)** have full read/write/execute permissions, and other users have no permission **(0)** to the directory.
6. Check that the permissions were set properly.
7. Modify the global login scripts so that normal users have a umask setting which prevents others from viewing or modifying new files and directories.

No solution for this lab enjoy yourself…..

```
groupadd stooges
useradd curly,larry,moe
passwd curly,larry,moe
usermod curly,larry,moe

mkdir /home/stooges
chown :stooges /home/stooges
chmod 2770 /home/stooges
ls -ld /home/stooges
vim /etc/bashrc
vim /etc/profile

su - curly
cd /home/stooges
touch curlyfile1
ls -la curlyfile1
```

MONITORING AND MANAGING LINUX PROCESSES

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

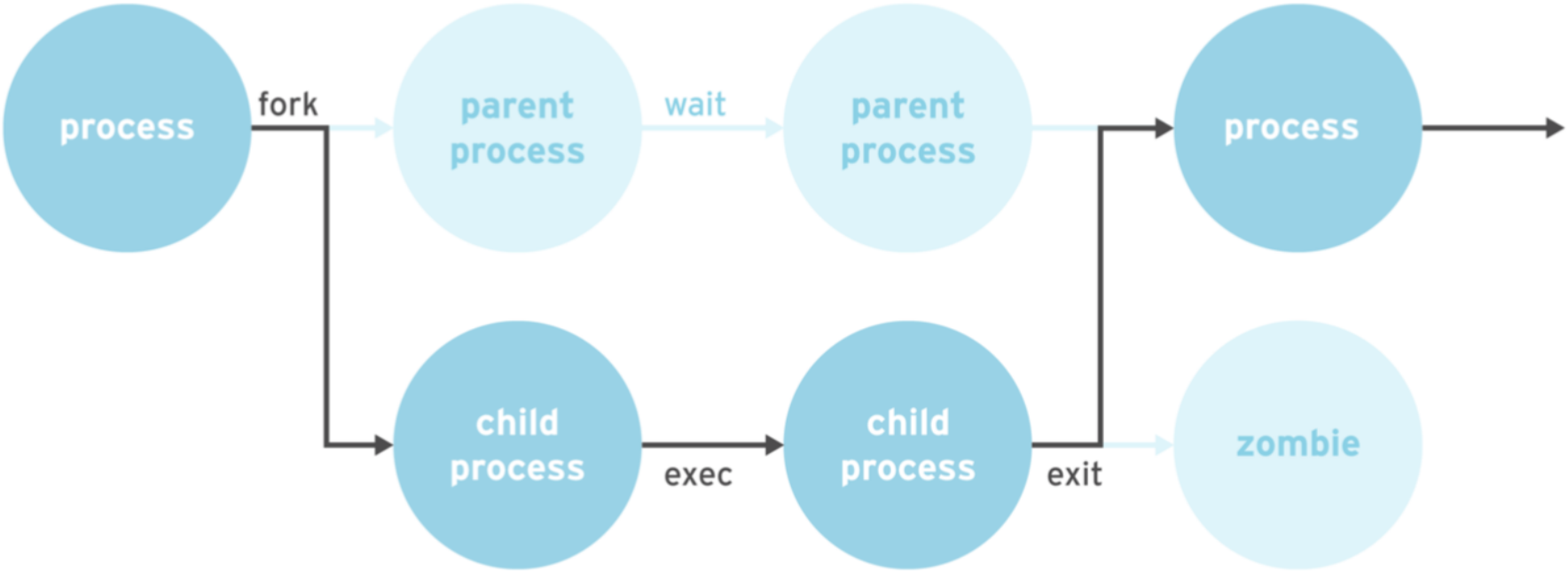
A process consists of:

- an address space of allocated memory,
- security properties including ownership credentials and privileges,
- one or more execution threads of program code, and
- the process state.

The environment of a process includes:

- local and global variables,
- a current scheduling context, and
- allocated system resources, such as file descriptors and network ports.

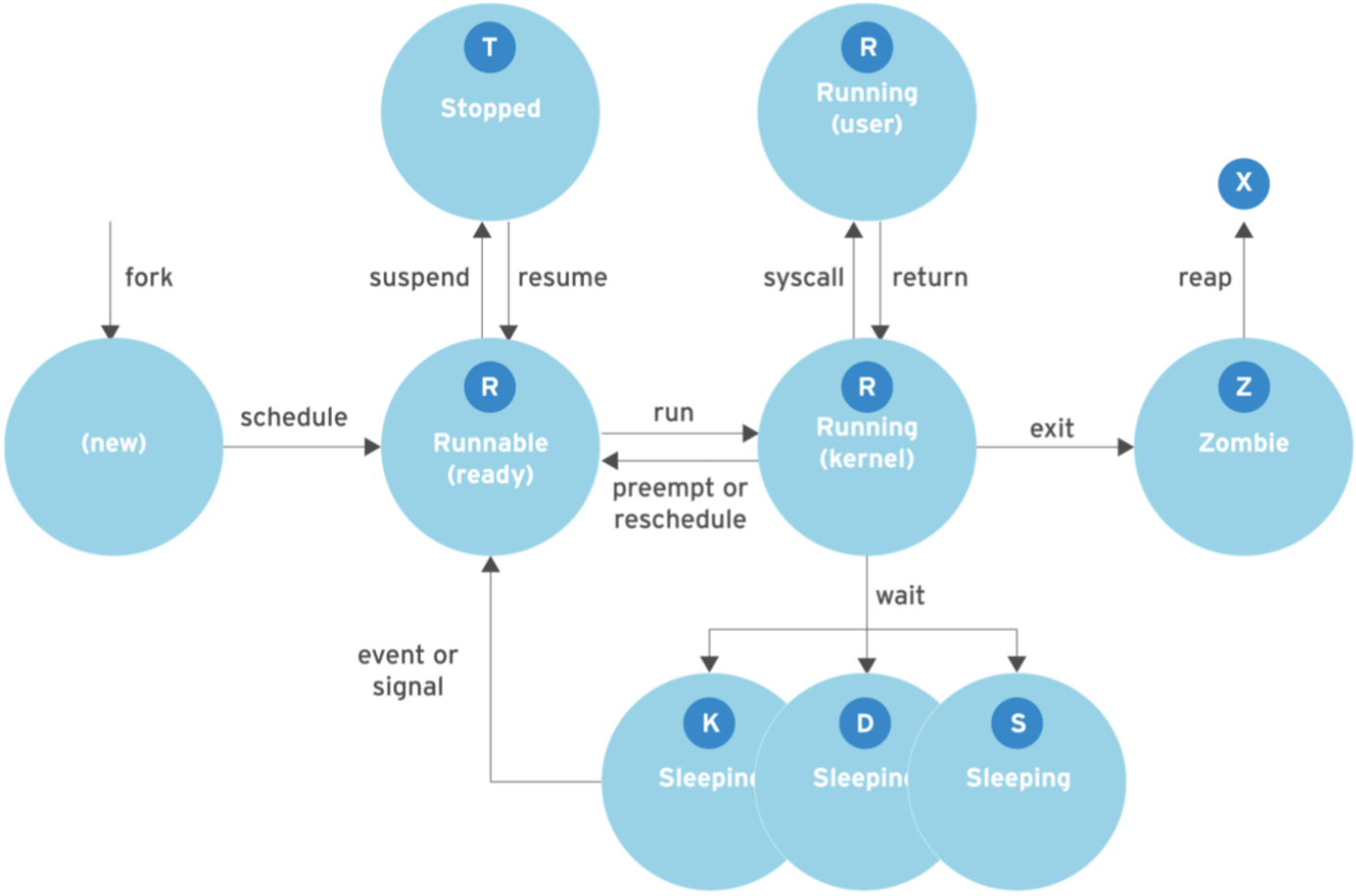
An existing (**parent)** process duplicates its own address space (**fork**) to create a new (**child**) process structure. Every new process is assigned a unique process ID (PID) for tracking and security. The PID and the parent's process ID (PPID) are elements of the new process environment. Any process may create a child process. All processes are descendants of the first system process, which is **systemd**(1).



Through the **fork** routine, a child process inherits security identities, previous and current file descriptors, port and resource privileges, environment variables, and program code. A child process may then **exec** its own program code. Normally, a parent process sleeps while the child process runs, setting a request (**wait**) to be signaled when the child completes. Upon exit, the child process has already closed or discarded its resources and environment; the remainder is referred to as a ***zombie***. The parent, signaled awake when the child exited, cleans the remaining structure, then continues with its own program code execution.

PROCESS STATE

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



NAME	FLAG	KERNEL-DEFINED STATE NAME AND DESCRIPTION
Running	**R**	TASK_RUNNING: The process is either executing on a CPU or waiting to run. Process can be executing user routines or kernel routines (system calls), or be queued and ready when in the Running (or Runnable) state.
Sleeping	**S**	TASK_INTERRUPTIBLE: The process is waiting for some condition: a hardware request, system resource access, or signal. When an event or signal satisfies the condition, the process returns to Running.
	D	TASK_UNINTERRUPTIBLE: This process is also Sleeping, but unlike S state, will not respond to delivered signals. Used only under specific conditions in which process interruption may cause an unpredictable device state.
	K	TASK_KILLABLE: Identical to the uninterruptible D state, but modified to allow the waiting task to respond to a signal to be killed (exited completely). Utilities frequently display Killable processes as D state.
Stopped	**T**	TASK_STOPPED: The process has been Stopped (suspended), usually by being signaled by a user or another process. The process can be continued (resumed) by another signal to return to Running.
	T	TASK_TRACED: A process that is being debugged is also temporarily Stopped and shares the same T state flag.
Zombie	**Z**	EXIT_ZOMBIE: A child process signals its parent as it exits. All resources except for the

		process identity (PID) are released.
	X	EXIT_DEAD: When the parent cleans up (reaps) the remaining child process structure, the process is now released completely. This state will never be observed in process-listing utilities.

LISTING PROCESS

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

- the user identification (UID) which determines process privileges,
- the unique process identification (PID),
- the CPU and real time already expended,
- how much memory the process has allocated in various locations,
- the current process state.

A common display listing (options **aux**) displays all processes, with columns in which users will be interested, and includes processes without a controlling terminal. A long listing (options **lax**) provides more technical detail, but may display faster by avoiding the username lookup. The similar UNIX syntax uses the options **-ef** to display all processes.

```
[root@desktop ~]# ps aux | head -5
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.2 129468  4920 ?        Ss   set13   0:25 /usr/lib[..]
root          2  0.0  0.0      0     0 ?        S    set13   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        S    set13   0:02 [ksoftirqd/0]
root          5  0.0  0.0      0     0 ?        S<   set13   0:00 [kworker/0:0H]

[root@desktop ~]# ps lax | head -5
F   UID    PID   PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0      1      0  20   0 129468  4920 ep_pol Ss   ?          0:25
/usr/lib[..]
1     0      2      0  20   0      0     0 kthrea S    ?          0:00 [kthreadd]
1     0      3      2  20   0      0     0 smpboo S    ?          0:02
[ksoftirqd]
1     0      5      2   0 -20      0     0 worker S<   ?          0:00
[kworker/0:0H]

[root@desktop ~]# ps -ef | head -5
UID         PID   PPID  C STIME TTY          TIME CMD
```

```
root           1       0   0 set13 ?        00:00:25 /usr/lib/systemd/systemd
--s[..]
root           2       0   0 set13 ?        00:00:00 [kthreadd]
root           3       2   0 set13 ?        00:00:02 [ksoftirqd/0]
root           5       2   0 set13 ?        00:00:00 [kworker/0:0H]
```

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

Processes in brackets (usually at the top) are scheduled kernel threads.
• Zombies in a ps listing as exiting or defunct.
• ps displays once. Use top(1) for a repetitive update process display.
• ps can display in tree format to view parent/child relationships.
• The default output is unsorted. Display order matches that of the system process table, which reuses table rows as processes die and new ones are created. Output may appear chronological, but is not guaranteed unless explicit **-O** or **--sort** options are used.

CONTROLLING JOBS

JOBS AND SESSIONS

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

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

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

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

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

RUNNING JOBS IN THE BACKGROUND

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

```
[root@desktop ~]# sleep 10 &
[1] 122888
[root@desktop ~]# ps -ef | grep sleep
UID         PID   PPID  C STIME TTY          TIME CMD
root     122909 121912  0 19:08 pts/0    00:00:00 sleep 10
[1]+  Done                    sleep 10
```

The bash shell tracks jobs, per session, in a table displayed with the **jobs** command.

```
[root@desktop ~]# sleep 1000 &
[1] 123060
[root@desktop ~]# jobs
[1]+  Running                 sleep 1000 &
```

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

```
[root@desktop ~]# fg %1
sleep 1000
```

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

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

```
[root@desktop ~]# fg %1
sleep 1000
^Z
```

```
[1]+  Stopped                 sleep 1000
```

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

The ps j command will display information relating to jobs.The **PGID** is the **PID** of the process group leader, normally the first process in the job's pipeline. The **SID** is the **PID** of the session leader, which for a job is normally the interactive shell that is running on its controlling terminal. Since the example sleep command is currently suspended, its process state is **T**.

```
[root@desktop ~]# ps j
  PPID    PID   PGID    SID TTY       TPGID STAT   UID   TIME COMMAND
  6974   8700   8700   8700 tty1       8700 Ss+      0   0:00 -bash
121906 121912 121912 121912 pts/0    123350 Ss       0   0:00 -bash
121912 123060 123060 121912 pts/0    123350 T        0   0:00 sleep 1000
121912 123350 123350 121912 pts/0    123350 R+       0   0:00 ps j
```

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

```
[root@desktop ~]# bg %1
[1]+ sleep 1000 &
[root@desktop ~]# jobs
[1]+  Running                 sleep 1000 &
```

If the user tries exiting, the jobs are killed.

```
[student@desktop ~]$ sleep 100 &
[1] 123516
[student@desktop ~]$ exit
logout
[root@desktop ~]# su - student
[student@desktop ~]$ jobs
```

LAB - BACKGROUND AND FOREGROUND PROCESSES

OUTCOMES

Practice suspending and restarting user processes.

1. Open two terminal windows, side by side, to be referred to as left and right.
2. In the left windows start a process that continuously appends the word "rock" and a space to the file ~/outfile at one-second intervals. The complete command set must be contained in parentheses for job control to interpret the set as a single job.

```
[student@desktop ~]$ (while true; do echo -n "rock " >> ~/outfile; sleep
1; done)
```

3. In the right windows use tail to confirm that the process is writing to the file

```
[student@desktop ~]$ tailf outfile
rock rock rock rock rock rock rock rock rock rock rock rock rock rock rock
rock
```

4. In the left window, suspend the running process. The shell returns the jobID in square brackets. In the right window, confirm that the process output has stopped.

```
[student@desktop ~]
Ctrl+z
```

5. In the left window, view the jobs list. The + denotes the current job. Restart the job in the background. In the right window, confirm that the process output is again active.

```
[student@desktop ~]$ jobs
[1]+  Stopped ( while true; do echo -n "rock " >> ~/outfile; sleep 1;
done )
[student@desktop ~]$ fg
( while true; do
    echo -n "rock " >> ~/outfile; sleep 1;
done )
Ctrl+C
[student@desktop ~]$ jobs
```

6. In the left window, start two more processes to append to the same output file. Replace "rock" with "paper," and then with "scissors." To properly background the process, the complete command set must be contained in parentheses and ended with an ampersand.

```
[student@desktop ~]$ (while true; do echo -n "paper " >> ~/outfile; sleep
1; done) &
[1] 126177
[student@desktop ~]$ (while true; do echo -n "scissors " >> ~/outfile;
sleep 1; done) &
[2] 126193
[student@desktop ~]$ jobs
[1]-  Running                 ( while true; do
    echo -n "paper " >> ~/outfile; sleep 1;
done ) &
[2]+  Running                 ( while true; do
    echo -n "scissors " >> ~/outfile; sleep 1;
done ) &
```

7. Using only commands previously learned, suspend the "rock" process. In the left window, foreground the job, using the job ID determined from the jobs listing, then suspend it using Ctrl+z. Confirm that the "rock" process is "Stopped". In the right window, confirm that "rock" output is no longer active.

```
[student@desktop ~]$ jobs
[student@desktop ~]$ fg %number
```

```
[student@desktop ~]$ Ctrl+Z
```

8. End the "paper" process. In the left window, foreground the job, then terminate it using Ctrl+c. Confirm that the "paper" process has disappeared. In the right window, confirm that "paper" output is no longer active.

```
[student@desktop ~]$ jobs
[student@desktop ~]$ fg %number
[student@desktop ~]$ Ctrl+C
```

9. In the left window, view the remaining jobs using ps. The suspended job has state T. The other background job is sleeping (S), since ps is "on cpu" (R) while displaying.

```
[student@desktop ~]$ ps j
  PPID    PID   PGID    SID TTY       TPGID STAT   UID   TIME COMMAND
123524 123525 123525 121912 pts/0    128131 S     1001   0:00 -bash
123525 126177 126177 121912 pts/0    128131 T     1001   0:00 -bash
126177 127357 126177 121912 pts/0    128131 T     1001   0:00 sleep 1
123525 127394 127394 121912 pts/0    128131 S     1001   0:00 -bash
128083 128115 128115 128083 pts/1    128115 S+    1001   0:00 tailf
outfile
127394 128130 127394 121912 pts/0    128131 S     1001   0:00 sleep 1
123525 128131 128131 121912 pts/0    128131 R+    1001   0:00 ps j
```

10. Stop the remaining two jobs. In the left window, foreground either job. Terminate it using Ctrl+c. Repeat with the remaining job. The "Stopped" job temporarily restarts when foregrounded. Confirm that no jobs remain and that output has stopped.

```
[student@desktop ~]$ jobs
[student@desktop ~]$ fg %number
Ctrl+C
[student@desktop ~]$ fg %number
Ctrl+C
[student@desktop ~]$ jobs
```

11. In the right window, stop the tail command. Close extra terminal windows.

```
[student@desktop ~]$
Ctrl+C
```

KILLING PROCESSES

PROCESS CONTROL USING SIGNALS

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

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

SIGNAL NUMBER	SHORT NAME	DEFINITION	PURPOSE
1	**HUP**	Hangup	Used to report termination of the controlling process of a terminal. Also used to request process reinitialization (configuration reload) without termination.
2	**INT**	Keyboard	Causes program termination. Can be blocked or handled. Sent

		Interrupt	by pressing **INTR** key combination (**Ctrl+c**).
3	**QUIT**	Keyboard quit	Similar to **SIGINT**, but also produces a process dump at termination. Sent by pressing **QUIT** key combination (**Ctrl+D)**.
9	**KILL**	kill, unblockable	Causes abrupt program termination. Cannot be blocked, ignored, or handled; always fatal.
15	**TERM**	Terminate	Causes program termination. Unlike **SIGKILL**, can be blocked, ignored, or handled. The polite way to ask a program to terminate; allows self-cleanup.
18	**CONT**	Continue	Sent to a process to resume if stopped. Cannot be blocked. Even if handled, always resumes the process.
19	**STOP**	Stop, unblockable	Suspends the process. Cannot be blocked or handled.
20	**TSTP**	Keyboard stop	Unlike SIGSTOP, can be blocked, ignored, or handled. Sent by pressing SUSP key combination (**Ctrl+z**).

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

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

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

Commands for sending signals by explicit request

Users signal their current foreground process by pressing a keyboard control sequence to suspend (**Ctrl+z**), kill (**Ctrl+c**), or core dump (**Ctrl+**) the process. To signal a background process or processes in a different session requires a signal-sending command.
Signals can be specified either by name (e.g., **-HUP** or **-SIGHUP**) or by number (e.g., **-1**).
Users may kill their own processes, but root privilege is required to kill processes owned by others.

- The kill command sends a signal to a process by ID. Despite its name, the kill command can be used for sending any signal, not just those for terminating programs.

```
[student@desktop ~]$ kill -l
 1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL    5) SIGTRAP
 6) SIGABRT   7) SIGBUS    8) SIGFPE    9) SIGKILL  10) SIGUSR1
11) SIGSEGV  12) SIGUSR2  13) SIGPIPE  14) SIGALRM  15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD  18) SIGCONT  19) SIGSTOP  20) SIGTSTP
21) SIGTTIN  22) SIGTTOU  23) SIGURG   24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF  28) SIGWINCH 29) SIGIO    30) SIGPWR
31) SIGSYS   34) SIGRTMIN 35) SIGRTMIN+1   36) SIGRTMIN+2   37) SIGRTMIN+3
```

```
38) SIGRTMIN+4   39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
[..]
```

Use **killall** to send a signal to one or more processes matching selection criteria, such as a command name, processes owned by a specific user, or all system-wide processes.

```
[student@desktop ~]$
killall command_pattern
[student@desktop ~]$
killall -signal command_pattern
[student@desktop ~]$
killall -signal -u username command_pattern
```

The **pkill** command,like **killall**, can signal multiple processes. **pkill** uses advanced selection criteria, which can include combinations of:

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

```
[student@desktop ~]$
pkill command_pattern
[student@desktop ~]$
pkill -signal command_pattern
[root@desktop ~]#
pkill -G GID command_pattern
[root@desktop ~]#
pkill -P PPID command_pattern
[root@desktop ~]#
pkill -t terminal_name -U UID command_pattern
```

LOGGING USERS OUT ADMINISTRATIVELY

The **w** command views users currently logged into the system and their cumulative activities. Use the **TTY** and **FROM** columns to determine the user's location.

All users have a controlling terminal, listed as **pts/N** while working in a graphical environment window (pseudo-terminal) or **ttyN** on a system console, alternate console, or other directly connected terminal device. Remote users display their connecting system name in the **FROM** column when using the **-f** option.

```
[student@desktop ~]$ w -f
 22:44:44 up 1 day, 12:57,  3 users,  load average: 0,16, 0,14, 0,14
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
root     tty1      24ago19 30:28   0.39s  0.39s -bash
root     pts/0     18:57   25:48   0.16s  0.10s -bash
student  pts/1     22:15    4.00s  0.04s  0.01s w -f
```

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

Users may be forced off a system for security violations, resource overallocation, or other administrative need. Users are expected to quit unnecessary applications, close unused command shells, and exit login sessions when requested.

When situations occur in which users cannot be contacted or have unresponsive sessions, runaway resource consumption, or improper system access, their sessions may need to be administratively terminated using signals.

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

```
[root@desktop ~]$ pgrep -l -u student
123525 bash
128082 sshd
128083 bash
129892 sleep
[root@desktop ~]# pkill -SIGKILL -u student
[root@desktop ~]# pgrep -l -u student
```

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

```
[root@desktop ~]# pgrep -l -u student
130661 sshd
130662 bash
130842 sleep
```

```
130844 sleep
130847 sleep
[root@desktop ~]# w -h -u student
student  pts/1    gateway          23:00    7.00s  0.02s  0.02s -bash
[root@desktop ~]# pkill -t pts/1
[root@desktop ~]# pgrep -l -u student
130661 sshd
130662 bash
```

LAB - KILLING PROCESSES

OUTCOMES

Experience with observing the results of starting and stopping multiple shell processes.

1. Open two terminal windows, side by side, to be referred to as left and right.
2. In the left window, start three processes that append text to an output file at one-second intervals. To properly background each process, the complete command set must be contained in parentheses and ended with an ampersand.

```
[student@desktop ~]$
(while true; do echo -n "game " >> ~/outfile; sleep 1; done) &
[student@desktop ~]$
(while true; do echo -n "set " >> ~/outfile; sleep 1; done) &
[student@desktop ~]$
(while true; do echo -n "match " >> ~/outfile; sleep 1; done) &
```

3. In the right window, use tail to confirm that all three processes are appending to the file. In the left window, view jobs to see all three processes "Running".

```
[student@desktop ~]$ tail -f ~/outfile
[student@desktop ~]$ jobs
[1]-  Running                 ( while true; do
```

```
    echo -n "game " >> ~/outfile; sleep 1;
done ) &
[2]-  Running                 ( while true; do
    echo -n "set " >> ~/outfile; sleep 1;
done ) &
[3]+  Running                 ( while true; do
    echo -n "match " >> ~/outfile; sleep 1;
done ) &
```

4. Suspend the "game" process using signals. Confirm that the "game" process is "Stopped". In the right window, confirm that "game" output is no longer active.

```
[student@desktop ~]$
kill -SIGSTOP %number
[student@desktop ~]$
jobs
```

5. Terminate the "set" process using signals. Confirm that the "set" process has disappeared. In the right window, confirm that "set" output is no longer active.

```
[student@desktop ~]$
kill -SIGTERM %number
[student@desktop ~]$ jobs
```

6. Resume the "game" process using signals. Confirm that the "game" process is "Running". In the right window, confirm that "game" output is again active.

```
[student@desktop ~]$
kill -SIGCONT %number
[student@desktop ~]$ jobs
```

7. Terminate the remaining two jobs. Confirm that no jobs remain and that output has stopped. From the left window, terminate the right window's tail command.
   Close extra terminal windows.

```
[student@desktop ~]$
kill -SIGTERM %number
[student@desktop ~]$ kill -SIGTERM %number
[student@desktop ~]$ jobs
[student@desktop ~]$
pkill -SIGTERM tail
[student@desktop ~]$
```

MONITORING PROCESS ACTIVITY

LOAD AVERAGE

The Linux kernel calculates a load average metric as an exponential moving average of the load number, a cumulative CPU count of active system resource requests.

- Active requests are counted from per-CPU queues for running thread sand threads waiting for I/ O, as the kernel tracks process resource activity and corresponding process state changes.
- Load number is a calculation routine run every five seconds by default, which accumulates and averages the active requests into a single number for all CPUs.
- Load average is the load number calculation routine result. Collectively, it refers to the three displayed values of system activity data averaged for the last 1, 5, and 15 minutes.

Understanding the Linux load average calculation

The load average represents the perceived system load over a time period. Linux implements the load average calculation as a representation of expected service wait times, not only for CPU but also for disk and network I/O.

Linux counts not only processes, but threads individually, as separate tasks. CPU request queues for running threads (nr_running) and threads waiting for I/O resources (nr_iowait) reasonably correspond to process states **R** (Running) and **D** (Uninterruptable Sleeping). Waiting for I/O includes tasks sleeping for expected disk and network responses.

The load number is a global counter calculation, which is sum-totaled for all CPUs. Since tasks returning from sleep may reschedule to different CPUs, accurate per-CPU counts are difficult, but an accurate cumulative count is assured. Displayed load averages represent all CPUs.

Linux counts each physical CPU core and microprocessor hyperthread as separate execution units, logically represented and referred to as individual CPUs. Each CPU has independent request queues. [1]

Interpreting displayed load average values

The three values represent the weighted values over the last 1, 5, and 15 minutes. A quick glance can indicate whether system load appears to be increasing or decreasing. Calculate the approximate per-CPU load value to determine whether the system is experiencing significant waiting.

top, **uptime**, **w**, display load average values.

```
[student@desktop ~]$ uptime
 21:17:21 up 1 day, 13:54,  2 users,  load average: 0,22, 0,23, 0,29
```

Divide the displayed load average values by the number of logical CPUs in the system. A value below 1 indicates satisfactory resource utilization and minimal wait times. A value above 1 indicates resource saturation and some amount of service waiting times.

An idle CPU queue has a load number of 0. Each ready and waiting thread adds a count of 1. With a total queue count of 1, the resource (CPU, disk, or network) is in use, but no requests spend time waiting. Additional requests increment the count, but since many requests can be processed within the time period, resource utilization increases, but not wait times.

Processes sleeping for I/O due to a busy disk or network resource are included in the count and increase the load average. While not an indication of CPU utilization, the queue count still indicates that users and programs are waiting for resource services.
Until resource saturation, a load average will remain below 1, since tasks will seldom be found waiting in queue. Load average only increases when resource saturation causes requests to remain queued and counted by the load calculation routine. When resource utilization approaches 100%, each additional request starts experiencing service wait time.

REAL-TIME PROCESS MONITORING

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

Default output columns are recognizable from other resource tools:

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

```
[student@desktop ~]$ top
top - 21:30:43 up 1 day, 14:07,  2 users,  load average: 0,04, 0,11, 0,19
Tasks: 165 total,   2 running, 163 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5,3 us,  5,3 sy,  0,0 ni, 89,5 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem :  2028088 total,    66252 free,  1664072 used,   297764 buff/cache
KiB Swap:  1470460 total,  1381884 free,    88576 used.    36136 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  8590 cassand+  20   0 3143772   1,5g 157872 S 15,9 79,3 377:26.21 java
     1 root      20   0  129468   4968   2304 S  0,0  0,2   0:27.33 systemd
     2 root      20   0       0      0      0 S  0,0  0,0   0:00.04 kthreadd
     3 root      20   0       0      0      0 S  0,0  0,0   0:03.25 ksoftirqd/0
     5 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 kworker/0:0H
     7 root      rt   0       0      0      0 S  0,0  0,0   0:00.00 migration/0
     8 root      20   0       0      0      0 S  0,0  0,0   0:00.00 rcu_bh
     9 root      20   0       0      0      0 S  0,0  0,0   0:09.84 rcu_sched
```

```
 10 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 lru-add-drain
 11 root      rt   0       0      0      0 S  0,0  0,0   0:02.78 watchdog/0
```

IDENTIFYING AUTOMATICALLY STARTED SYSTEM PROCESSES

INTRODUCTION TO systemd

System startup and server processes are managed by the systemd System and Service Manager. This program provides a method for activating system resources, server daemons, and other processes, both at boot time and on a running system.

Daemons are processes that wait or run in the background performing various tasks. Generally, daemons start automatically at boot time and continue to run until shutdown or until they are manually stopped. By convention, the names of many daemon programs end in the letter "d".

To listen for connections, a daemon uses a socket. This is the primary communication channel with local or remote clients. Sockets may be created by daemons or may be separated from the daemon and be created by another process, such as systemd. The socket is passed to the daemon when a connection is established by the client.

A service often refers to one or more daemons, but starting or stopping a service may instead make a one-time change to the state of the system, which does not involve leaving a daemon process running afterward (called oneshot).

A bit of history

For many years, process ID 1 of Linux and UNIX systems has been the init process. This process was responsible for activating other services on the system and is the origin of the term "init system." Frequently used daemons were started on systems at boot time with System V and LSB init scripts. These are shell scripts, and may vary from one distribution to another. Less frequently used daemons were started on demand by another service, such as **initd** or **xinetd**, which listens for client connections. These systems have several limitations, which are addressed with systemd.

In most of the Linux operating systems before Ubuntu 15.04, Debian 8, CentOS 7, Fedora 15, process ID 1 was systemd, the new init system. A few of the new features provided by systemd include:

- Parallelization capabilities, which increase the boot speed of a system.
- On-demand starting of daemons without requiring a separate service.
- Automatic service dependency management, which can prevent long timeouts, such as by not starting a network service when the network is not available.
- A method of tracking related processes to get her by using Linux control groups.

systemctl and systemd units

The systemctl command is used to manage different types of systemd objects, called units. A list of available unit type scan be displayed with **systemctl -l** help.

Some common unit types are listed below:

- Service units have a .service extension and represent system services. This type of unit is used to start frequently accessed daemons, such as a web server.
- Socket units have a .socket extension and represent inter-process communication(IPC) sockets. Control of the socket will be passed to a daemon or newly started service when a client connection is made. Socket units are used to delay the start of a service at boot time and to start less frequently used services on demand. These are similar in principle to services which use the xinetd superserver to start on demand.
- Path units have a .path extension and a reused to delay the activation of a service until a specific file system change occurs. This is commonly used for services which use spool directories, such as a printing system.

Service states

The status of a service can be viewed with systemctl status name.type. If the unit type is not provided, systemctl will show the status of a service unit, if one exists.

```
[root@desktop ~]# systemctl status sshd.service
sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
   Active: active (running) since Thu 2014-02-27 11:51:39 EST; 7h ago
 Main PID: 1073 (sshd)
   CGroup: /system.slice/sshd.service
           └─1073 /usr/sbin/sshd -D
Feb 27 11:51:39 server0.example.com systemd[1]: Started OpenSSH server daemon.
Feb 27 11:51:39 server0.example.com sshd[1073]: Could not load host key: /et...y
Feb 27 11:51:39 server0.example.com sshd[1073]: Server listening on 0.0.0.0 ....
Feb 27 11:51:39 server0.example.com sshd[1073]: Server listening on :: port 22.
Feb 27 11:53:21 server0.example.com sshd[1270]: error: Could not load host k...y
Feb 27 11:53:22 server0.example.com sshd[1270]: Accepted password for root f...2
Hint: Some lines were ellipsized, use -l to show in full.
```

Several keywords indicating the state of the service can be found in the status output:

KEYWORD:	DESCRIPTION:
loaded	Unit configuration file has been processed.
active (running)	Running with one or more continuing processes.
active (exited)	Successfully completed a one-time configuration.
active (waiting)	Running but waiting for an event.
inactive	Not running.
enabled	Will be started at boot time.
disabled	Will not be started at boot time.
static	Can not be enabled, but may be started by an enabled unit automatically.

LISTING UNIT FILES WITH **systemctl**

In this example, please follow along with the next steps while your instructor demonstrates obtaining status information of services.

Query the state of all units to verify a system startup.

```
[root@desktop ~]#
systemctl
```

Query the state of only the service units.

```
[root@desktop ~]#
systemctl --type=service
```

Investigate any units which are in a failed or maintenance state. Optionally, add the-loption to show the full output.

```
[root@desktop ~]#
systemctl status sshd.service -l
```

The status argument may also be used to determine if a particular unit is active and show if the unit is enabled to start at boot time. Alternate commands can also easily show the active and enabled states:

```
[root@desktop ~]#
systemctl is-active sshd
```

```
[root@desktop ~]#
systemctl is-enabled sshd
```

List the active state of all loaded units. Optionally, limit the type of unit. The --all option will add inactive units.

```
[root@desktop ~]#
systemctl list-units --type=service
[root@desktop ~]#
systemctl list-units --type=service --all
```

View the enabled and disabled settings for all units. Optionally,limit the type of unit.

```
[root@desktop ~]#
systemctl list-unit-files --type=service
```

View only failed services.

```
[root@desktop ~]#
systemctl --failed --type=service
```

LAB - IDENTIFY THE STATUS OF systemd UNITS

List all service units on the system.

```
[student@server ~]$
systemctl list-units --type=service
```

List all socket units, active and inactive, on the system.

```
[student@server ~]$
systemctl list-units --type=socket --all
```

Explore the status of the chronyd service. This service is used for network time synchronization (NTP).

1. Display the status of the **chronyd** service. Note the process ID of any active daemons.

   ```
   [student@server ~]$
   systemctl status sshd
   ```

2. Confirm that the listed daemons are running.

   ```
   [student@server ~]$
   ps -p PID
   ```

Explore the status of the sshd service. This service is used for secure encrypted communication between systems.

1. Determine if the sshd service is enabled to start at system boot.

```
[student@server ~]$
systemctl is-enabled sshd
```

2. Determine if the sshd service is active without displaying all of the status information.

```
[student@server ~]$
systemctl is-active sshd
```

3. Display the status of the sshd service.

```
[student@server ~]$
systemctl status sshd
```

4. List the enabled or disabled state so fall service units.

```
[student@server ~]$
systemctl list-unit-files --type=service
```

CONTROLLING SYSTEM SERVICES

STARTING AND STOPPING SYSTEM DAEMONS ON A RUNNING SYSTEM

Changes to a configuration file or other updates to a service may require that the service be restarted. A service that is no longer used may be stopped before removing the software. A service that is not frequently used may be manually started by an administrator only when it is needed.
In this example, please follow along with the next steps while your instructor demonstrates managing services on a running system.

View the status of a service.

```
[root@desktop ~]#
systemctl status sshd.service
```

Verify that the process is running.

```
[root@desktop ~]#
ps -up PID
```

Stop the service and verify the status.

```
[root@desktop ~]#
```

```
systemctl stop sshd.service
[root@desktop ~]#
systemctl status sshd.service
```

Start the service and view the status. The process ID has changed.

```
[root@desktop ~]#
systemctl start sshd.service
[root@desktop ~]#
systemctl status sshd.service
```

Stop, then start, the service in a single command.

```
[root@desktop ~]#
systemctl restart sshd.service
[root@desktop ~]#
systemctl status sshd.service
```

Issue instructions for a service to read and reload its configuration file without a complete stop and start. The process ID will not change.

```
[root@desktop ~]#
systemctl reload sshd.service
[root@desktop ~]#
systemctl status sshd.service
```

Unit dependencies

Services may be started as dependencies of other services. If a socket unit is enabled and the service unit with the same name is not, the service will automatically be started when a request is made on the network socket. Services may also be triggered by path units when a file system condition is met. For example, a file placed into the print spool directory will cause the cups print service to be started if it is not running.

```
[root@desktop ~]# systemctl stop cups.service
Warning: Stopping cups, but it can still be activated by:
  cups.path
  cups.socket
```

To completely stop printing services on a system, stop all three units. Disabling the service will disable the dependencies.

The **systemctl list-dependencies** UNIT command can be used to print out a tree of what other units must be started if the specified unit is started. Depending on the exact dependency, the other unit may need to be running before or after the specified unit starts. The --reverse option to this command will show what units need to have the specified unit started in order to run.

Masking services

At times, a system may have conflicting services installed. For example, there are multiple methods to manage networks (network and NetworkManager) and firewalls (iptables and firewalld). To prevent an administrator from accidentally starting a service, that service may be masked. Masking will create a link in the configuration directories so that if the service is started, nothing will happen.

```
[root@desktop ~]# systemctl mask network
ln -s '/dev/null' '/etc/systemd/system/network.service'
[root@desktop ~]# systemctl unmask network
rm '/etc/systemd/system/network.service'
```

ENABLING SYSTEM DAEMONS TO START OR STOP AT BOOT

Starting a service on a running system does not guarantee that the service will be started when the system reboots. Similarly, stopping a service on a running system will not keep it from starting again when the system reboots. Services are started at boot time when links are created in
the appropriate systemd configuration directories. These links are created and removed with systemctl commands.
In this example, please follow along with the next steps while your instructor demonstrates enabling and disabling services.

View the status of a service.

```
[root@desktop ~]#
systemctl status sshd.service
```

Disable the service and verify the status. Note that disabling a service does not stop the service.

```
[root@desktop ~]#
systemctl disable sshd.service
[root@desktop ~]#
systemctl status sshd.service
```

Enable the service and verify the status.

```
[root@desktop ~]#
systemctl enable sshd.service
[root@desktop ~]#
systemctl is-enabled sshd.service
```

SUMMARY OF systemctl COMMANDS

Services can be started and stopped on a running system and enabled or disabled for automatic start at boot time.

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 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 umask 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 which are required and wanted by the specified unit.	systemctl list-dependencies UNIT

LAB - USING systemctl TO MANAGE SERVICES

Observe the results of systemctl restart and systemctl reload commands.
Display the status of the sshd service. Note the process ID of the daemon.
Restart the sshd service and view the status. The process ID of the daemon has changed.
Reload the sshd service and view the status. The process ID of the daemon has not changed and connections have not been interrupted.
Verify that the chronyd service is running.
Stop the chronyd service and view the status.
Determine if the chronyd service is enabled to start at system boot.

REGULAR EXPRESSIONS FUNDAMENTALS

WRITING REGULAR EXPRESSIONS

Regular expressions is a pattern-matching language used for enabling applications to sift through data looking for specific content. In addition to **vim**, **grep**, and **less** using regular expressions, programming languages such as **Perl**, **Python**, and **C** all use regular expressions when using pattern-matching criteria.

Regular expressions are a language of their own, which means they have their own syntax and rules. This section will take a look at the syntax used in creating regular expressions, as well as showing some examples of using regular expressions.

A simple regular expression

The simplest regular expression is an exact match. An exact match is when the characters in the regular expression match the type and order in the data that is being searched.

Suppose that a user was looking through the following file of data looking for all occurrences of the pattern cat:

```
cat
dog
concatenate
dogma
category
educated
boondoggle
vindication
chilidog
```

cat is an exact match of a **c**, followed by an **a**, followed by a **t**. Using cat as the regular expression while searching the previous file gives the following matches:

cat

con**cat**enate

category

edu**cat**ed

vindi**cat**ion

Using line anchors

The previous section used an exact match regular expression on a file of data. Note that the regular expression would match the data no matter where on the line it occurred: beginning, end, or

middle of the word or line. One way that can be used to control the location of where the regular expression looks for a match is a line anchor.

Use a **^**, a beginning of line anchor, or **$**, an end of line anchor. Using the file from earlier:

```
cat
dog
concatenate
dogma
```

```
category
educated
boondoggle
vindication
chilidog
```

To have the regular expression match cat, but only if it occurs at the beginning of the line in the file, use **^cat**. Applying the regular expression **^cat** to the data would yield the following matches:

cat
category

If users only wanted to locate lines in the file that ended with **dog**, use that exact expression and an end of line anchor to create the regular expression **dog$**. Applying **dog$** to the file would find two matches:

dog
chili**dog**

If users wanted to make sure that the pattern was the only thing on a line, use both the beginning and end of line anchors. **^cat$** would locate only one line in the file, one with a beginning of a line, a **c**, followed by an **a**, followed with a **t**, and ending with an end of line.

Another type of anchor is the word boundary. **\<** and **\>** can be used to respectively match the beginning and end of a word.

Wildcards and multipliers

Regular expressions use a **.** as the unrestricted wildcard character. A regular expression of **c.t** will look for data containing a **c**, followed by any one character, followed by a **t**. Examples of data that would match this regular expression's pattern are cat, cot, and cut, but also c5t and cQt.

Another type of wildcard used in regular expressions is a set of acceptable characters at a specific character position. When using an unrestricted wildcard, users could not predict the character that would match the wildcard; however, if users wanted to only match the words cat, cot, and cut, but not odd items like c5t or cQt, replace the unrestricted wildcard with one where acceptable characters are specified. If the regular expression was changed to **c[aou]t**, it would be specifying that the regular expression should match patterns that start with a **c**, are followed by an **a** or an **o** or a **u**, followed by a **t**.

Multipliers are a mechanism used often with wildcards. Multipliers apply to the previous character in the regular expression. One of the more common multipliers used is ***.** A *, when used in a regular expression, modifies the previous character to mean zero to infinitely many of that character. If a regular expression of **c.*t** was used, it would match **ct**, **cat**, **coat**, **culvert**, etc.; any data that started with a **c**, then zero to infinitely many characters, ending with a **t**.

Another type of multiplier would indicate the number of previous characters desired in the pattern. An example of using an explicit multiplier would be **c.\{2\}t**. Using this regular expression, users are looking for data that begins with a c, followed by exactly any **two characters**, ending with a **t**.

MATCHING TEXT WITH **grep**

USING grep

grep is a command provided as part of the distribution which utilizes regular expressions to isolate matching data.

grep usage

The basic usage of grep is to provide a regular expression and a file on which the regular expression should be matched.

```
[student@server ~]$ grep 'cat$' /usr/share/dict/words
```

grep can be used in conjunction with other commands using a |.

```
[root@desktop ~]# ps aux | grep '^student'
```

grep options

grep has many useful options for adjusting how it uses the provided regular expression with data.

OPTION	FUNCTION
-i	Use the regular expression provided; however, do not enforce case sensitivity (run case-insensitive).
-v	Only display lines that DO NOT contain matches to the regular expression.
-r	Apply the search for data matching the regular expression recursively to a group of files or directories.
-A <NUMBER>	Display <NUMBER> of lines after the regular expression match.
-B <NUMBER>	Display <NUMBER> of lines before the regular expression match.
-e	With multiple -e options used, multiple regular expressions can be supplied and will be used with a logical or.

grep examples

For the next few examples, use the following file contents, stored in a file named dogs-n-cats.

```
[student@server ~]$ cat dogs-n-cats
# This file contains words with cats and dogs
Cat
dog
concatenate
dogma
category
educated
boondoggle
vindication
Chilidog
```

Regular expressions are case-sensitive by default; using the **-i** option with grep will cause it to treat the regular expression without case sensitivity.

```
[root@desktop ~]# grep -i 'cat' dogs-n-cats
Cat
concatenate
category
educated
vindication
```

Sometimes, users know what they are not looking for, instead of what they are looking for. In those cases, using **-v** is quite handy. In the following example, all lines, case insensitive, that do not contain the regular expression 'cat' will display.

```
[root@desktop ~]# grep -i -v 'cat' dogs-n-cats dog
dogma
boondoggle
Chilidog
```

Another practical example of using **-v** is needing to look at a file, but not wanting to be distracted with content in comments. In the following example, the regular expression will match all lines that begin with a **#** or **;** (typical characters that indicate the line will be interpreted as a comment).

```
[student@server ~]$ grep -v '^[#;]' <FILENAME>
```

There are times where users need to look for lines that contain information so different that users cannot create just one regular expression to find all the data. grep provides the -e option for these situations. In the following example, users will locate all occurrences of either 'cat' or 'dog'.

```
[student@server ~]$ grep -e 'cat' -e 'dog' dogs-n-cats
# This file contains words with cats and dogs
dog
concatenate
dogma
category
educated
boondoggle
vindication
Chilidog
```

LAB - USING grep

WITH LOGS

Files: **/var/log/messages**
User: **root**

1) Elevate your privileges to gain a root login using su - (or sudo su -)

```
[student@desktop ~]$ su -
```

2) Check the current time so we know not only the starting time, but the ending time of the messages we are looking for.
The following commands assume a start time provided by the lab grep script of Oct 4 11:36.

```
[root@desktop ~]# date
Fri Oct  4 11:36:33 CEST 2019

[root@desktop ~]# grep '^Oct  4 11:[0-9][0-9]' /var/log/messages
Oct  4 11:02:44 desktop systemd: Time has been changed
Oct  4 11:02:44 desktop kernel: usb 2-2.1: USB disconnect, device number 13
Oct  4 11:02:44 desktop systemd: Stopping Load/Save RF Kill Switch Status of
rfkill9...
```

```
Oct  4 11:02:44 desktop systemd: Unit bluetooth.target is not needed anymore.
Stopping.
Oct  4 11:02:44 desktop systemd: Stopped target Bluetooth.
Oct  4 11:02:44 desktop systemd: Stopped Load/Save RF Kill Switch Status of
rfkill9.[...]
```

3) Create an entry log in /var/log/message

```
[root@desktop ~]# logger "ERROR: this is an error example log"
```

4) Modify your regular expression to locate the ERROR message.

```
[root@desktop ~]# grep '^Oct  4 11:[0-9][0-9].*ERROR' /var/log/messages
Oct  4 11:36:33 desktop root: ERROR: this is an error example log
```

LAB - Regex

https://regexone.com/lesson/introduction_abcs

BASH SHELL SCRIPTING BASICS

BASH SCRIPTING BASICS

Many simple day-to-day system administration tasks can be accomplished by the numerous Linux command-line tools available to administrators. However, tasks with greater complexity often require the chaining together of multiple commands. In these situations, Linux command-line tools can be combined with the offerings of the Bash shell to create powerful shell scripts to solve real-world problems.

In its simplest form, a Bash shell script is simply an executable file composed of a list of commands. However, when well-written, a shell script can itself become a powerful command-line tool when executed on its own, and can even be further leveraged by other scripts.

Proficiency in shell scripting is essential to the success of Linux system administrators in all operational environments. Working knowledge of shell scripting is especially crucial in enterprise environments, where its use can translate to improved efficiency and accuracy of routine task completion.

Choosing a programming language

While Bash shell scripting can be used to accomplish many tasks, it may not be the right tool for all scenarios. Administrators have a wide variety of programming languages at their disposal, such as C, C++, Perl, Python,

Ruby and other programming languages. Each programming language has its strengths and weaknesses, and as such, none of the programming languages are the right tool for every situation.

Bash shell scripts are a good choice for tasks which can be accomplished mainly by calling other command-line utilities. If the task involves heavy data processing and manipulation, other languages such as Perl or Python will be better suited for the job. While Bash supports arithmetic operations, they are limited to simple integer arithmetic. For more complex arithmetic operations, C or C++ should be considered. If a solution requires the use of arrays, Bash is probably not the best tool. Bash has supported one-dimensional arrays for some time, and the latest version even supports associative arrays. However, Perl or Python have much better array functionality, with the ability to accommodate multidimensional arrays.

As administrators become proficient at shell scripting, they will gain more knowledge regarding its capabilities and limitations. This experience combined with exposure to other programming languages over time, will provide administrators with a better understanding of the advantages and disadvantages of each, and which problems each one is best suited for.

CREATING AND EXECUTING BASH SHELL SCRIPTS

A Bash shell script can be created by opening a new empty file in a text editor. While any text editor can be used, advanced editors, such as vim or emacs, understand Bash shell syntax and can provide color-coded highlighting. This highlighting can be a tremendous help for spotting syntactical errors, such as unpaired quotes, unclosed brackets, and other common blunders.

The command interpreter

The first line of a Bash shell script begins with '#!', also commonly referred to as a sharp-bang, or the abbreviated version, sha-bang. This two-byte notation is technically referred to as the magic pattern. It indicates that the file is an executable shell script. The path name that follows is the command interpreter, the program that should be used to execute the script. Since Bash shell scripts are to be interpreted by the Bash shell, they begin with the following first line.

```
#!/bin/bash
...
```

Executing a Bash shell script

After a Bash shell script is written, its file permissions and ownership need to be modified so that it is executable. Execute permission is modified with the **chmod** command, possibly in conjunction with the **chown** command to change the file ownership of the script accordingly. Execute permission should only be granted to the users that the script is intended for.

Once a Bash shell script is executable, it can be invoked by entering its name on the command line. If only the base name of the script file is entered, Bash will search through the directories specified in the shell's **PATH** environmental variable, looking for the first instance of an executable file matching that name. Administrators should avoid script names that match other executable files, and should also ensure that the **PATH** variable is correctly configured on their system so that the script will be the first match found by the shell. The **which** command, followed by the file-name of the executable script displays in which directory the script resides that is executed when the script name is invoked as a command resides.

```
[student@server ~]$ which hello

~/bin/hello

[student@server ~]$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/student/.local/bin:/home/
student/bin
```

DISPLAYING OUTPUT

The **echo** command can be used to display arbitrary text by passing the text as an argument to the command. By default, the text is directed to standard out (STDOUT), but can also be directed to standard error (STDERR) using output redirection. In the following simple Bash script, the **echo** command displays the message "Hello, world" to STDOUT.

```
[student@server ~]$ cat ~/bin/hello
#!/bin/bash
echo "Hello, world"
[student@server ~]$ hello
Hello, world
```

While seemingly simple in its function, the **echo** command is widely used in shell scripts due to its usefulness for various purposes. It is commonly used to display informational or error messages during the script execution.

These messages can be a helpful indicator of the progress of a script and can be directed either to standard out, standard error, or be redirected to a log file for archiving. When displaying error messages, it is good practice to direct them to STDERR to make it easier to differentiate error messages from normal status messages.

```
[student@server ~]$ cat ~/bin/hello
#!/bin/bash
echo "Hello, world"
echo "ERROR: Houston, we have a problem." >&2

[student@server ~]$ hello 2> hello.log
Hello, world
[student@server ~]$ cat hello.log
ERROR: Houston, we have a problem.
```

The **echo** command can also be very helpful when trying to debug a problematic shell script. The addition of **echo** statements to the portion of the script that is not behaving as expected can help clarify the commands being executed, as well as the values of variables being invoked.

Quoting special characters

A number of characters or words have special meanings to the Bash shell in specific context. There are situations when the literal values, rather than the special meanings, of these characters or words are desired. For example, the **#** character is interpreted by Bash as the beginning of a **comment** and is therefore ignored, along with everything following it on the same line. If this special meaning is not desired, then Bash needs to be informed that the # character is to be treated as a literal value. The meanings of special characters or words can be disabled through the use of the escape character, ****, single quotes, **‘’**, or double quotes, **“”**.

The escape character, ****, removes the special meaning of the single character immediately following it. For example, to display the literal string **#** test with the **echo** command, the **#** character must not be interpreted by Bash with special meaning. The escape character can be placed in front of the **#** character to disable its special meaning.

```
[student@server ~]$ echo # not a comment


[student@server ~]$ echo \# not a comment
# not a comment
```

The escape character, \, only removes the special meaning of a single character. When more than one character in a text string needs to be escaped, users can either use the escape character multiple times or employ single quotes, **‘ ‘**. Single quotes preserve the literal meaning of all characters they enclose. The following example demonstrates how single quotes can be used when multiple characters need to be escaped.

```
[student@server ~]$ echo # not a comment #
[student@server ~]$ echo \# not a comment #
# not a comment
[student@server ~]$ echo \# not a comment \#
# not a comment #
[student@server ~]$ echo '# not a comment #'
# not a comment #
```

While single quotes preserve the literal value of all characters they enclose, double quotes differ in that they do not preserve the literal value of the dollar sign, **$**, the back-ticks, ``, and the backslash, ****. When enclosed with double quotes, the dollar sign and back-ticks preserve their special meaning, and the special meaning of the backslash character is only retained when it precedes a dollar sign, back-tick, double quote, backslash, or newline.

```
[student@server ~]$ echo '$HOME'
$HOME
[student@server ~]$ echo '`pwd`'
`pwd`
[student@server ~]$ echo '"Hello, world"'
"Hello, world"
[student@server ~]$ echo "$HOME"
/home/student
[student@server ~]$ echo "`pwd`"
/home/student
[student@server ~]$ echo ""Hello, world""
Hello, world
[student@server ~]$ echo "\$HOME"
$HOME
[student@server ~]$ echo "\`pwd\`"
```

```
`pwd`
[student@server ~]$ echo "\"Hello, world\""
"Hello, world"
```

USING VARIABLES

As the complexity of a shell script increases, it is often helpful to make use of variables. A variable serves as a container, within which a shell script can store data in memory. Variables make it easy to access and modify the stored data during a script's execution.

Assigning values to variables

Data is assigned as a value to a variable via the following syntax:

```
VARIABLENAME=value
```

While variable names are typically uppercase letters, they can be made up of numbers, letters (uppercase and lowercase), and the underscore character, '_'. However, a variable name cannot start with a number. The equal

sign, =, is used to assign values to variables and must not be separated from the variable name or the value by spaces. The following are some examples of valid variable declarations.

```
COUNT=40

first_name=John

file1=/tmp/abc

_ID=RH123
```

Two common types of data stored in variables are integer values and string values. It is good practice to quote string values when assigning them to variables, since the space character is interpreted by Bash as a word separator when not enclosed within single or double quotes. Whether single or double quotes should be used to enclose variable values depends on how characters with special meanings to Bash should be treated.

```
full_name='John Doe'

full_name="$FIRST $LAST"

price='$1'
```

Expanding variable values

The value of a variable can be recalled through a process known as variable expansion by preceding the variable name with a dollar sign, **$**. For example, the value of the **VARIABLENAME** variable can be referenced with **$VARIABLENAME**. The **$VARIABLENAME** syntax is the simplified version of the brace-quoted form of variable expansion, **${VARIABLENAME}**. While the simplified form is usually acceptable, there are situations where the brace-quoted form must be used to remove ambiguity and avoid unexpected results.

In the following example, without the use of brace quotes, Bash will interpret **$FIRST_$LAST** as the variable **$FIRST_** followed by the variable **$LAST**, rather than the variables **$FIRST** and **$LAST** separated by the '_' character. Therefore, brace quoting must be used for variable expansion to function properly in this scenario.

```
[student@server ~]$ FIRST_=Jane

[student@server ~]$ FIRST=John

[student@server ~]$ LAST=Doe


[student@server ~]$ echo $FIRST_$LAST JaneDoe

[student@server ~]$ echo ${FIRST}_$LAST John_Doe
```

USING BASH SHELL EXPANSION FEATURES

Aside from variable expansion, the Bash shell offers several other types of shell expansion features. Of these, command substitution and arithmetic expansion can be useful in Bash shell scripting, and are commonly used.

Command substitution

Command substitution replaces the invocation of a command with the output from its execution. This feature allows the output of a command to be used in a new context, such as the argument to another command, the value for a variable, and the list for a loop construct.

Command substitution can be invoked with the old form of enclosing the command in back- ticks, such as **`<COMMAND>`**. However, the preferred method is to use the newer **$()** syntax, **$(<COMMAND>)**.

```
[student@server ~]$ echo "Current time: `date`"

Current time is Thu Jun 5 16:24:24 EDT 2014.

[student@server ~]$ echo "Current time: $(date)"

Current time is Thu Jun 5 16:24:30 EDT 2014.
```

The newer syntax is preferred since it allows for nesting of command substitutions. In the following nested command substitution example, the output of the find command is used as arguments for the tar command, which then has its output stored into the variable TAROUTPUT.

```
[root@desktop ~]# TAROUTPUT=$(tar cvf /tmp/incremental_backup.tar $(find /etc -type f -mtime -1))

[root@desktop ~]# echo $TAROUTPUT

/etc/group /etc/gshadow /etc/shadow- /etc/passwd /etc/shadow /etc/passwd- /etc/tuned/active_profile /etc/rht /etc/group- /etc/gshadow- /etc/resolv.conf
```

Arithmetic expansion

Bash's arithmetic expansion can be used to perform simple integer arithmetic operations, and uses the syntax **$((<EXPRESSION>))**. When enclosed within **$(())**, arithmetic expressions are evaluated by Bash and then replaced with their results. Bash performs variable expansion and command substitution on the enclosed expression before its evaluation. Like command line substitution, nesting of arithmetic substitutions is allowed.

```
[student@server ~]$ echo $((1+1))

2

[student@server ~]$ echo $((2*2))

4

[student@server ~]$ COUNT=1; echo $(($(($COUNT+1))*2))

4
```

Space characters are allowed in the expression used within an arithmetic expansion. The use of space characters can improve readability in complicated expressions or when variables are included.

```
[student@server ~]$ SEC_PER_MIN=60
[student@server ~]$ MIN_PER_HR=60
[student@server ~]$ HR_PER_DAY=24
[student@server ~]$ SEC_PER_DAY=$(( $SEC_PER_MIN * $MIN_PER_HR * $HR_PER_DAY ))
[student@server ~]$ echo "There are $SEC_PER_DAY seconds in a day."
There are 86400 seconds in a day.
```

The following are some of the commonly used operators in arithmetic expressions, along with their meanings.

OPERATOR	MEANING
<VARIABLE>++	variable post-increment
<VARIABLE>--	variable post-decrement
++<VARIABLE>	variable pre-increment
--<VARIABLE>	variable pre-decrement
-	unary minus
+	unary plus
****	exponentiation
*****	multiplication
/	division
%	remainder
+	addition
-	substraction

When multiple operators exist in an expression, Bash will evaluate certain operators in order according to their precedence. For example, multiplication and division operators have a higher precedence than addition and subtraction. Single parentheses can be used to group sub- expressions if the evaluation order desired differs from the default precedence.

```
[student@server ~]$ echo $(( 1 + 1 * 2 ))
3
[student@server ~]$ echo $(( (1 + 1) * 2 ))
4
```

The following table lists the order of precedence for commonly used arithmetic operators from highest to lowest. Operators that have equal precedence are listed together.

OPERATOR	MEANING
<VARIABLE>++, <VARIABLE>--	variable post-increment and post-decrement
++<VARIABLE>, --<VARIABLE>	variable pre-increment and pre-decrement

-,+	unary minus and plus
**	exponentiation
-	unary minus
+	unary plus
**	exponentiation
*, /, %	multiplication, division, remainder
+, -	addition, subtraction

ITERATING WITH THE FOR LOOP

System administrators often encounter repetitive tasks in their day-to-day activities. Repetitive tasks can take the form of executing an action multiple times on a target, such as checking a process every minute for 10 minutes to see if it has completed. Task repetition can also take the form of executing an action a single time across multiple targets, such as performing a database backup of each database on a system. The for loop is one of the multiple shell looping constructs offered by Bash, and can be used for task iterations.

Using the for loop

Bash's for-loop construct uses the following syntax. The loop processes the items provided in **<LIST>** in order one by one and exits after processing the last item on the list. Each item in the list is temporarily stored as the

value of **<VARIABLE>**, while the for loop executes the block of commands contained in its construct. The naming of the variable is arbitrary. Typically, the variable value is referenced by commands in the command block.

```
for <VARIABLE> in <LIST>; do
     <COMMAND>
...
     <COMMAND> referencing <VARIABLE>
done
```

The list of items provided to a for loop can be supplied in several ways. It can be a list of items entered directly by the user, or be generated from different types of shell expansion, such as variable expansion, brace expansion, file name expansion, and command substitution. Some examples that demonstrate the different ways lists can be provided to for loops follow.

```
[student@server ~]$ for HOST in host1 host2 host3; do echo $HOST; done
host1
```

```
host2
host3

[student@server ~]$ for HOST in host{1,2,3}; do echo $HOST; done
host1
host2
host3

[student@server ~]$ for HOST in host{1..3}; do echo $HOST; done
host1
host2
host3

[student@server ~]$ for FILE in file*; do ls $FILE; done
filea
fileb
```

```
filec


[student@server ~]$ for FILE in file{a..c}; do ls $FILE; done
filea
fileb
filec

[student@server ~]$ for PACKAGE in $(rpm -qa | grep kernel); do echo "$PACKAGE was
installed on $(date -d @$(rpm -q --qf "%{INSTALLTIME}\n" $PACKAGE))"; done
abrt-addon-kerneloops-2.1.11-12.el7.x86_64 was installed on Tue Apr 22 00:09:07
 EDT 2014
kernel-3.10.0-121.el7.x86_64 was installed on Thu Apr 10 15:27:52 EDT 2014
kernel-tools-3.10.0-121.el7.x86_64 was installed on Thu Apr 10 15:28:01 EDT 2014
kernel-tools-libs-3.10.0-121.el7.x86_64 was installed on Thu Apr 10 15:26:22 EDT
2014
```

```
[student@server ~]$ for EVEN in $(seq 2 2 8); do echo "$EVEN"; done; echo "Who do
 we appreciate?"
2
4
6
8
Who do we appreciate?
```

TROUBLESHOOTING SHELL SCRIPT BUGS

Inevitably, administrators who write, use, or maintain shell scripts will encounter bugs with a script. Bugs are typically due to typographical errors, syntactical errors, or poor script logic.

A good way to deal with shell scripting bugs is to make a concerted effort to prevent them from occurring in the first place during the authoring of the script. As previously mentioned, using a text editor with Bash syntactical highlighting can help make mistakes more obvious when writing scripts. Another easy way to avoid introducing bugs into scripts is by adhering to good practices during the creation of the script.

Good styling practices

Use comments to help clarify to readers the purpose and logic of the script. The top of every script should include comments providing an overview of the script's purpose, intended actions, and general logic. Also use comments throughout the script to clarify the key portions, and especially

sections that may cause confusion. Comments will not only aid other users in the reading and debugging of the script, but will also often help the author recall the workings of the script once some time has passed.

Structure the contents of the script to improve readability. As long as the syntax is correct, the command interpreter will flawlessly execute the commands within a script with absolutely no regard for their structure or formatting. Here are some good practices to follow:

• Break up long commands in to multiple lines of smaller code chunks.Shorter pieces of code are much easier for readers to digest and comprehend.

• Line up the beginning and ending of multiline statements to make it easier to see that control structures begin and end, and whether they are being closed properly.

• Indent lines with multiline statements to represent the hierarchy of code logic and the flow of control structures.

• Use line spacing to separate command blocks to clarify when one code section end sand another begins.

• Use consistent formatting through the entirety of a script.

When utilized, these simple practices can make it significantly easier to spot mistakes during authoring, as well as improve the readability of the script for future readers. The following example demonstrates how the incorporation of comments and spacing can greatly improve script readability.

```
#!/bin/bash
for PACKAGE in $(rpm -qa | grep kernel); do echo "$PACKAGE was installed on
$(date
 -d @$(rpm -q --qf "%{INSTALLTIME}\n" $PACKAGE))"; done
```

```
#!/bin/bash
#
# This script provides information regarding when kernel-related packages
# are installed on a system by querying information from the RPM database.
#
# Variables
```

```
PACKAGETYPE=kernel
PACKAGES=$(rpm -qa | grep $PACKAGETYPE)
# Loop through packages
for PACKAGE in $PACKAGES; do
     # Determine package install date and time
     INSTALLEPOCH=$(rpm -q --qf "%{INSTALLTIME}\n" $PACKAGE)
   # RPM reports time in epoch, so need to convert
     # it to date and time format with date command
     INSTALLDATETIME=$(date -d @$INSTALLEPOCH)
     # Print message
     echo "$PACKAGE was installed on $INSTALLDATETIME"
done
```

Do not make assumptions regarding the outcome of actions taken by a script. This is especially true of inputs to the script, such as command-line arguments, input from users, command substitutions, variable expansions, and file name expansions. Rather than making assumptions about the integrity of these inputs, make the worthwhile effort to employ the use of proper quoting and sanity checking.

The same caution should be utilized when acting upon entities external to the script. This includes interacting with files, and calling external commands. Make use of Bash's vast number of file and directory tests when interacting with files and directories. Perform error checking on the exit status of commands rather than counting on their success and blindly continuing along with the script when an unexpected error occurs.

The extra steps taken to rule out assumptions will increase the script's robustness, and keep it from being easily derailed and then inflicting unintended and unnecessary damage to a system. A couple of seemingly harmless lines of code, such as the ones that follow, make very risky assumptions about command execution outcome and file name expansion. If the directory change fails, either due to directory permissions or the directory being nonexistent, the subsequent file removal will be performed on a list of unknown files in an unintended directory.

```
cd $TMPDIR

rm *
```

Lastly, while well-intentioned administrators may employ good practices when authoring their scripts, not all will always agree on what constitutes good practices. Administrators should do themselves and others a favor and always apply their practices consistently through the entirety of their scripts. They should also be considerate and understanding of individual differences when it comes to programming styles and formatting in scripts authored by others. When modifying others' scripts, administrators should follow the existing structure,

formatting, and practices used by the original author, rather than imposing their own style on a portion of the script and destroying the script's consistency, and thereby ruining its readability and future maintainability.

Debug and verbose modes

If despite best efforts, bugs are introduced into a script, administrators will find Bash's debug mode extremely useful. To activate the debug mode on a script, add the -x option to the command interpreter in the first line of the script.

```
#!/bin/bash -x
```

Another way to run a script in debug mode is to execute the script as an argument to Bash with the **-x** option.

```
[student@server bin]$ bash -x <SCRIPTNAME>
```

Bash's debug mode will print out commands executed by the script prior to their execution. The results of all shell expansion performed will be displayed in the printout. The following example shows the extra output that is displayed when debug mode is activated.

```
[student@server bin]$ cat filesize
#!/bin/bash
DIR=/home/student/tmp
for FILE in $DIR/*; do
 echo "File $FILE is $(stat --printf='%s' $FILE) bytes."
done
[student@server bin]$ ./filesize

File /home/student/tmp/filea is 133 bytes.
File /home/student/tmp/fileb is 266 bytes.
```

```
File /home/student/tmp/filec is 399 bytes.
[student@server bin]$ bash -x ./filesize
+ DIR=/home/student/tmp
+ for FILE in '$DIR/*'
++ stat --printf=%s /home/student/tmp/filea
+ echo 'File /home/student/tmp/filea is 133 bytes.' File /home/student/tmp/filea
is 133 bytes.
+ for FILE in '$DIR/*'
++ stat --printf=%s /home/student/tmp/fileb
+ echo 'File /home/student/tmp/fileb is 266 bytes.'
File /home/student/tmp/fileb is 266 bytes.
+ for FILE in '$DIR/*'
++ stat --printf=%s /home/student/tmp/filec
+ echo 'File /home/student/tmp/filec is 399 bytes.'
File /home/student/tmp/filec is 399 bytes.
```

While Bash's debug mode provides helpful information, the voluminous output may actually become more hindrance than help for troubleshooting, especially as the lengths of scripts increase. Fortunately, the debug mode can be enabled partially on just a portion of a script, rather than on its entirety. This feature is especially useful when debugging a long script and the source of the problem has been narrowed to a portion of the script.

Debugging can be turned on at a specific point in a script by inserting the command **set -x** and turned off by inserting the command **set +x**. The following demonstration shows the previous example script with debugging enabled just for the command line enclosed in the for loop.

```
[student@server bin]$ cat filesize
#!/bin/bash
DIR=/home/student/tmp
for FILE in $DIR/*; do
        set -x
        echo "File $FILE is $(stat --printf='%s' $FILE) bytes."
set +x done
[student@server bin]$ ./filesize
++ stat --printf=%s /home/student/tmp/filea
```

```
+ echo 'File /home/student/tmp/filea is 133 bytes.' File /home/student/tmp/filea
is 133 bytes.
+ set +x
++ stat --printf=%s /home/student/tmp/fileb
+ echo 'File /home/student/tmp/fileb is 266 bytes.' File /home/student/tmp/fileb
is 266 bytes.
+ set +x
++ stat --printf=%s /home/student/tmp/filec
+ echo 'File /home/student/tmp/filec is 399 bytes.' File /home/student/tmp/filec
is 399 bytes.
+ set +x
```

In addition to debug mode, Bash also offers a verbose mode, which can be invoked with the **-v** option. In verbose mode, Bash will print each command to standard out prior to its execution.

```
[student@desktop ~]$ cat filesize
#!/bin/bash
```

```
DIR=/home/student/tmp
for FILE in $DIR/*; do
 echo "File $FILE is $(stat --printf='%s' $FILE) bytes."
done

[student@desktop ~]$ ./filesize
File /home/student/tmp/file1 is 5 bytes.
File /home/student/tmp/file2 is 10 bytes.
File /home/student/tmp/file3 is 15 bytes.
File /home/student/tmp/file4 is 15 bytes.
File /home/student/tmp/file5 is 20 bytes.
[student@desktop ~]$ bash -v ./filesize
#!/bin/bash
DIR=/home/student/tmp
for FILE in $DIR/*; do
 echo "File $FILE is $(stat --printf='%s' $FILE) bytes."
```

```
done
File /home/student/tmp/file1 is 5 bytes.
File /home/student/tmp/file2 is 10 bytes.
File /home/student/tmp/file3 is 15 bytes.
File /home/student/tmp/file4 is 15 bytes.
File /home/student/tmp/file5 is 20 bytes.
```

Like the debug feature, the verbose feature can also be turned on and off at specific points in a script by inserting the **set -v** and **set +v** lines, respectively.

LAB - WRITING BASH SCRIPTS

Your company provides hosting service to customers, and you have been tasked with writing a Bash shell script called **/usr/local/sbin/mkaccounts** to automate the process of creating accounts for new customers. At the end of each day, a colon-separated data file called **/tmp/support/newusers** is created and contains information on new customers that have signed up. The script will read through this data file and create a user account for each new customer.

The sales department has requested that the script also generate a report breaking down the new customers by support tier, so sales staff can stay informed regarding trends of support tier purchases. For each support tier, they would like the report to detail the total number of new customers and the percentage of the day's new customers that chose the tier type.

Create user file

```
[root@desktop ~]# mkdir /tmp/support/
[root@desktop ~]# cat > /tmp/support/newusers  <<HERE
Betsey:Werts:60:1
```

```
Henriette:Balla:30:1
Julieann:Hopps:30:2
Conrad:Menz:60:1
Annabell:Cho:90:1
Allyn:Kenley:60:1
Ceola:Jacquez:60:1
Rebecca:Fabry:60:1
Philip:Bamber:90:1
HERE
```

Create your script. Set the file name, /tmp/support/newusers, as the value of the variable NEWUSERSFILE.

Create the new script with a text editor.

```
[root@desktop ~]# vim /usr/local/sbin/mkaccounts
```

Specify the interpreter program for the script.

```
#!/bin/bash
```

Set the NEWUSERSFILE variable.

```
# Variables

NEWUSERSFILE=/tmp/support/newusers
```

Loop through the entries in the $NEWUSERSFILE datafile and extract the values of the first, second, and last fields to obtain the first name, last name, and support tier level of each new customer. Create an account for each new customer such that the account name is the lowercase combination of their first initial and last name (i.e., jdoe) and the comment of the account is their full name (i.e., John Doe).

Read through the entries in the file by initiating a for loop in combination with command substitution.

```
 # Loop

for ENTRY in $(cat $NEWUSERSFILE); do
```

Extract and save the values of the first, second, and last fields.

```
# Extract first, last, and tier fields

FIRSTNAME=$(echo $ENTRY | cut -d: -f1)
```

```
LASTNAME=$(echo $ENTRY | cut -d: -f2)
TIER=$(echo $ENTRY | cut -d: -f4)
```

Compose the account name for the new customer.

```
# Make account name
FIRSTINITIAL=$(echo $FIRSTNAME | cut -c 1 | tr 'A-Z' 'a-z')
LOWERLASTNAME=$(echo $LASTNAME | tr 'A-Z' 'a-z')

ACCTNAME=$FIRSTINITIAL$LOWERLASTNAME
```

Create the account with the useradd command.

```
# Create account
useradd $ACCTNAME -c "$FIRSTNAME $LASTNAME"
```

Close the for loop.

```
done
```

Create a report summarizing the number of new customers for each support tier and what percentage of the total new user accounts the support type comprises. The report should print as follows:

```
"Tier 1","8","88%"
"Tier 2","1","11%"
"Tier 3","0","0%"
```

Determine the total number of new customers and store the value in a variable.

```
TOTAL=$(wc -l < $NEWUSERSFILE)
```

Determine the total number of new customers for each support tier and store the totals into variables.

```
TIER1COUNT=$(grep -c :1$ $NEWUSERSFILE)
TIER2COUNT=$(grep -c :2$ $NEWUSERSFILE)
TIER3COUNT=$(grep -c :3$ $NEWUSERSFILE)
```

Calculate the percentages for each support tier.

```
TIER1PCT=$(( $TIER1COUNT * 100 / $TOTAL ))
TIER2PCT=$(( $TIER2COUNT * 100 / $TOTAL ))
TIER3PCT=$(( $TIER3COUNT * 100 / $TOTAL ))
```

Print the report.

```
# Print report
echo "\"Tier 1\",\"$TIER1COUNT\",\"$TIER1PCT%\""
echo "\"Tier 2\",\"$TIER2COUNT\",\"$TIER2PCT%\""
echo "\"Tier 3\",\"$TIER3COUNT\",\"$TIER3PCT%\""
```

Save and execute the script.

Make the script executable.

```
[root@desktop ~]# chmod u+x /usr/local/sbin/mkaccounts
```

Execute the script.

```
[root@desktopX ~]# /usr/local/sbin/mkaccounts
"Tier 1","8","88%"
"Tier 2","1","11%"
"Tier 3","0","0%"
```

Check the /etc/passwd file to check if the new users was correctly created

Create another script `/usr/local/sbin/rmaccounts` to delete account created

Check the /etc/passwd file to check if the new users was correctly deleted

ENHANCING BASH SHELL SCRIPTS WITH CONDITIONALS AND CONTROL STRUCTURES

USING BASH SPECIAL VARIABLES

While user-defined variables provide a means for script authors to create containers to store values used by a script, Bash also provides some predefined variables, which can be useful when writing shell scripts. One type of predefined variable is positional parameters.

Positional parameters

Positional parameters are variables which store the values of command-line arguments to a script. The variables are named numerically. The variable **0** refers to the script name itself. Following that, the variable **1** is predefined with the first argument to the script as its value, the variable **2** contains the second argument, and so on. The values can be referenced with the syntax **$1**, **$2**, etc.

Bash provides special variables to refer to positional parameters: **$*** and **$@**. Both of these variables refer to all arguments in a script, but with a slight difference. When **$*** is used, all of the arguments are seen as a single word. However, when **$@** is used, each argument is seen as a separate word. This is demonstrated in the following example.

```
[student@server bin]$ cat showargs
#!/bin/bash
for ARG in "$*"; do
 echo $ARG
done

[student@server bin]$ ./showargs "argument 1" 2 "argument 3"
argument 1 2 argument 3
```

```
[student@server bin]$ cat showargs

#!/bin/bash
for ARG in "$@"; do
 echo $ARG
done

[student@server bin]$ ./showargs "argument 1" 2 "argument 3"
argument 1
2
argument 3
```

Another value which may be useful when working with positional parameters is **$#**, which represents the number of command-line arguments passed to a script. This value can be used to verify whether any arguments, or the correct number of arguments, are passed to a script.

```
[student@server bin]$ cat countargs

#!/bin/bash
echo "There are $# arguments."

[student@server bin]$ ./countargs There are 0 arguments.
[student@server bin]$ ./countargs "argument 1" 2 "argument 3"
There are 3 arguments.
```

EVALUATING EXIT CODES

Every command returns an exit status, also commonly referred to as return status or exit code. A successful command exits with an exit status of **0**. Unsuccessful commands exit with a nonzero exit status. Upon completion, a command's exit status is passed to the parent process and stored in the **?** variable. Therefore, the exit status of an executed command can be retrieved by displaying the value of **$?**. The following examples demonstrate the execution and exit status retrieval of several common commands.

```
[student@server bin]$
ls /etc/hosts
/etc/hosts
[student@server bin]$ echo $?
0
[student@server bin]$ ls /etc/nofile
ls: cannot access /etc/nofile: No such file or directory
[student@server bin]$ echo $?
2
```

```
[student@server bin]$ grep localhost /etc/hosts
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4 ::1
localhost localhost.localdomain localhost6 localhost6.localdomain6
[student@server bin]$ echo $?
0
[student@server bin]$ grep random /etc/hosts
[student@server bin]$ echo $?
1
```

Using exit codes within a script

Once executed, a script will exit when it has processed all of its contents. However, there may be times when it is desirable to **exit** a script midway through, such as when an error condition is encountered. This can be accomplished with the use of the **exit** command within a script. When a script encounters the exit command, it will exit immediately and skip the processing of the remainder of the script.

The **exit** command can be executed with an optional integer argument between **0** and **255**, which represents an exit code. An exit code value of **0** represents no error. All other nonzero values indicate an error exit code. Script authors can use different nonzero values to differentiate between different types of errors encountered. This exit

code is passed back to the parent process, which stores it in the **?** variable and can be accessed with **$?** as demonstrated in the following examples.

```
[student@server bin]$ cat hello


#!/bin/bash
echo "Hello, world"
exit 0
[student@server bin]$ ./hello
Hello, world
[student@server bin]$ echo $?
0
```

```
[student@server bin]$ cat hello


#!/bin/bash
echo "Hello, world"
exit 1
[student@server bin]$ ./hello
Hello, world
[student@server bin]$ echo $?
1
```

If the exit command is called without an argument, then the script will exit and pass on to the parent process the exit status of the last command executed.

TESTING SCRIPT INPUTS

To ensure that scripts are not easily derailed by unexpected conditions, it is good practice for script authors to not make assumptions regarding inputs, such as command-line arguments, user inputs, command substitutions, variable expansions, file name expansions, etc. Integrity checking can be performed by using Bash's test feature. Tests can be performed using Bash's test command syntax,**[<TESTEXPRESSION>]**. They can also be performed using Bash's newer extended test command syntax, **[[<TESTEXPRESSION>]]**, which has been available since Bash version 2.02.

Like all commands, the test command produces an exit code upon completion, which is stored as the value **$?**. To see the conclusion of a test, simply display the value of **$?** immediately following the execution of the test command. Once again, an exit status value of **0** indicates the test succeeded, while nonzero values indicate the test failed.

Performing comparison tests

Comparison test expressions make use of binary comparison operators. These operators expect two objects, one on each side of the operator, and evaluate the two for equality and inequality. Bash

uses a different set of operators for string and numeric comparisons, and uses the following syntax

format:

```
[ <ITEM1> <BINARY COMPARISON OPERATOR> <ITEM2> ]
```

Bash's numeric comparison is limited to integer comparison. The following list of binary comparison operators is used in Bash for integer comparison.

OPERATOR	MEANING	EXAMPLE
-eq	is equal to	["$a" -eq "$b"]
-ne	is not equal to	["$a" -ne "$b"]
-gt	is greater than	["$a" -gt "$b"]
-ge	is greater than or equal to	["$a" -ge "$b"]
-lt	is less then	["$a" -lt "$b"]

-le	is less than or equal to	["$a" -le "$b"]

The following examples demonstrate the use of Bash's numeric comparison operators.

```
[student@server ~]$ [ 1 -eq 1 ]; echo $?
0
[student@server ~]$ [ 1 -ne 1 ]; echo $?
1
[student@server ~]$ [ 8 -gt 2 ]; echo $?
0
[student@server ~]$ [ 2 -ge 2 ]; echo $?
0
[student@server ~]$ [ 2 -lt 2 ]; echo $?
1
[student@server ~]$ [ 1 -lt 2 ]; echo $?
0
```

Bash's string comparison uses the following binary operators.

OPERATOR	MEANING	EXAMPLE
=	is equal to	["$a" = "$b"]
==	is equal to	["$a" == "$b"]
!=	is not equal to	["$a" != "$b"]

The following examples demonstrate the use of Bash's string comparison operators.

```
[student@server ~]$ [ abc = abc ]; echo $?
0
[student@server ~]$ [ abc == def ]; echo $?
1
[student@server ~]$ [ abc != def ]; echo $?
0
```

Bash also has a few unary operators available for string evaluation. Unary operators evaluate just one item using the following format.

```
[ <UNARY OPERATOR> <ITEM> ]
```

The following table shows Bash's unary operators for string evaluation.

OPERATOR	MEANING	EXAMPLE
-z	string is zero length (null)	[-z "$a"]
-n	string is not null	[-n "$a"]

```
[student@desktop ~]$ STRING=''; [ -z "$STRING" ]; echo $?
0
[student@desktop ~]$ STRING='abc'; [ -n "$STRING" ]; echo $?
0
```

Testing files and directories

Bash's string and binary operators allow users to implement the good practice of not assuming the integrity of inputs to a shell script. The same caution should be utilized when scripts interact with external entities, such as files and directories. Bash offers a large number of test operators for this purpose, as listed in the following table.

OPERATOR	MEANING	EXAMPLE
-b	file exists and is block special	[-b <FILE>]
-c	file exists and is character special	[-c <FILE>]
-d	file exists and is a directory	[-d <DIRECTORY>]
-e	file exists	[-e <FILE>]
-f	file is a regular file	[-f <FILE>]
-L	file exists and is a symbolic link	[-L <FILE>]
-r	file exists and read permission is granted	[-r <FILE>]
-s	file exists and has a size greater than zero	[-s <FILE>]
-w	file exists and write permission is granted	[-w <FILE>]

-x	file exists and execute (or search) permission is granted	[-x <FILE>]

Bash also offers a few binary comparison operators for performing file comparison. These operators are defined in the following table.

OPERATOR	MEANING	EXAMPLE
-ef	FILE1 has the same device and inode number as FILE2	[<FILE1> -ef <FILE2>]
-nt	FILE1 has newer modification date than FILE2	[<FILE1> -nt <FILE2>]
-ot	FILE1 has older modification date than FILE2	[<FILE1> -ot <FILE2>]

Logical AND, OR operators

Situations may arise where it may be useful to test more than one condition. Bash's logical **AND** operator, **&&**, allows users to perform a compound condition test to see if both of two conditions are true. On the other hand, Bash's logical OR operator, **||**, allows users to test whether one of two conditions are true. The following examples demonstrate the use of Bash's logical **AND** and **OR** operators.

```
[student@desktop ~]$ [ 2 -gt 1 ] && [ 1 -gt 0 ]; echo $?
0
[student@desktop ~]$ [ 2 -gt 1 ] && [ 1 -gt 2 ]; echo $?
1
[student@desktop ~]$ [ 2 -gt 1 ] || [ 1 -gt 2 ]; echo $?
0
[student@desktop ~]$ [ 0 -gt 1 ] || [ 1 -gt 2 ]; echo $?
1
```

USING CONDITIONAL STRUCTURES

Simple shell scripts represent a collection of commands which are executed from beginning to end. Conditional structures allow users to incorporate decision making into shell scripts, so that certain portions of the script are executed only when certain conditions are met.

If/then statement

The simplest of the conditional structures in Bash is the **if/then** construct, which has the following syntax.

```
if <CONDITION>; then
      <STATEMENT>
...
      <STATEMENT>
fi
```

With this construct, if a given condition is met, one or more actions are taken. If the given condition is not met, then no action is taken. The numeric, string, and file tests previously demonstrated are frequently utilized for testing the conditions in **if/then** statements. The following code section demonstrates the use of an **if/then** statement to start the psacct service if it is not active.

```
systemctl is-active sshd > /dev/null 2>&1
if  [ $? -ne 0 ]; then
  systemctl start sshd
fi
```

If/then/else statement

The **if/then** conditional structure can be further expanded so that different sets of actions can be taken depending on whether a condition is met. This is accomplished with the **if/then/else** conditional construct.

```
if <CONDITION>; then
      <STATEMENT>
...
      <STATEMENT>
    else
      <STATEMENT>
```

```
      ...
      <STATEMENT>
fi
```

The following code section demonstrates the use of an **if/then/else** statement to start the sshd service if it is not active and to stop it if it is active.

```
systemctl is-active sshd > /dev/null 2>&1
if  [ $? -ne 0 ]; then
  systemctl start sshd
else
  systemctl stop sshd
fi
```

If/then/elif/then/else statement

Lastly, the **if/then/else** conditional structure can be further expanded to test more than one condition, executing a different set of actions when a condition is met. The construct for this is shown in the following example. In this conditional structure, Bash will test the conditions in the order presented. Upon finding a condition that is true, Bash will execute the actions associated with the condition and then skip the remainder of the conditional structure. If none of the conditions are true, then Bash will execute the actions enumerated in the **else** clause.

```
if <CONDITION>; then
      <STATEMENT>
...
      <STATEMENT>
    elif <CONDITION>; then
      <STATEMENT>
      ...
      <STATEMENT>
    else
```

```
  <STATEMENT>
  ...
  <STATEMENT>
fi
```

The following code section demonstrates the use of an **if/then/elif/then/else** statement to run the **mysql** client if the mariadb service is active, run the **psql** client if the postgresql service is active, or run the **sqlite3** client if both the mariadb and postgresql services are not active.

```
systemctl is-active sshd > /dev/null 2>&1
SSHD_ACTIVE=$?
systemctl is-active network > /dev/null 2>&1
NETWORK_ACTIVE=$?
if  [ "$SSHD_ACTIVE" -eq 0 ]; then
  echo "sshd active"
elif  [ "$NETWORK_ACTIVE" -eq 0 ]; then
```

```
  echo "network is active"
else
  echo "no connection active"
fi
```

Case statement

Users can add as many **elif** clauses as they want into an **if/then/elif/then/else** statement to test as many conditions as they need. However, as more are added, the statement and its logic becomes increasingly harder to read and comprehend. For these more complex situations, Bash offers another conditional structure known as case statements. The **case** statement utilizes the following syntax:

```
case <VALUE> in
   <PATTERN1>)
       <STATEMENT>
       ...
       <STATEMENT>
       ;;
   <PATTERN2>)
    <STATEMENT>
       ...
       <STATEMENT>
```

```
        ;;
esac
```

The **case** statement attempts to match **<VALUE>** to each **<PATTERN>** in order, one by one. When a pattern matches, the code segment associated with that pattern is executed, with the **;;** syntax indicating the end of the block. All other patterns remaining in the **case** statement are then skipped and the **case** statement is exited. As many pattern/statement blocks as needed can be added.

To mimic the behavior of an **else** clause in an **if/then/elif/then/else** construct, simply use ***** as the final pattern in the **case** statement. Since this expression matches anything, it has the effect of executing a set of commands if none of the other patterns are matched.

The **case** statements are widely used in init scripts. The following code section is an example of how they are commonly used to take different actions, depending on the argument passed to the script.

```
case "$1" in
    start)
        echo “start”
        ;;
    stop)
        echo ‘rm -f $lockfile’
        echo “stop”
        ;;
    restart)
        echo “restart”
        ;;
    reload)
        echo “reload”
        ;;
    status)
        echo “status”
```

```
        ;;
    *)
       echo "Usage: $0 (start|stop|restart|reload|status)"
       ;;
esac
```

If the actions to be taken are the same for more than one pattern in a case statement, the patterns can be combined to share the same action block, as demonstrated in the following example. The pipe character, **|**, is used to separate the multiple patterns.

```
case "$1" in
   ...
   reload|restart)
       restart
;; ...
esac
```

LAB - ENHANCING BASH SHELL SCRIPTS WITH CONDITIONALS AND CONTROL STRUCTURES

Your company provides web hosting service to customers, and you have been tasked with writing a Bash shell script called **/usr/local/sbin/mkvhost** to automate the many steps involved in setting up an Apache name-based virtual host for your customers. The script will be used for virtual host creation on all servers going forward, so it needs to also be able to accommodate the one-time tasks that are executed the first time a new server is configured for name-based virtual hosting.

The script will take two arguments. The first argument will be the fully qualified domain name of the new virtual host. The second argument will be a number between 1 and 3, which represents the support tier that the customer purchased. The support tier determines the support email address, which will be set with the Apache **ServerAdmin** directive for the virtual host.

The script will create a configuration file under **/etc/httpd/conf.vhosts.d** with the name **<VIRTUALHOSTNAME>.conf** for each virtual host. It will also create a document root directory for the virtual host at **/srv/<VIRTUALHOSTNAME>/www**. Prior to creating the virtual host configuration file and document root directory, the script will check to make sure they do not already exist to ensure there will not be a conflict.

- Install the httpd package with yum.

```
[root@desktop ~]# yum install -y httpd
```

- Enable and start httpd.

```
[root@desktop ~]# systemctl enable httpd
 ln -s '/usr/lib/systemd/system/httpd.service' '/etc/systemd/system/multi-user.target.wants/httpd.service'
[root@desktop ~]# systemctl start httpd
```

Begin writing your script. Store the first and second argument of the script in the **VHOSTNAME** and **TIER** variables, respectively. Set the following variables:

VARIABLE	VALUE
HTTPDCONF	/etc/httpd/conf/httpd.conf
VHOSTCONFDIR	/etc/httpd/conf.vhosts.d
DEFVHOSTCONFFILE	$VHOSTCONFDIR/00-default- vhost.conf
VHOSTCONFFILE	$VHOSTCONFDIR/$VHOSTNAME.conf
WWWROOT	/srv
DEFVHOSTDOCROOT	$WWWROOT/default/www
VHOSTDOCROOT	$WWWROOT/$VHOSTNAME/www

Create the new script file with a text editor.

[root@desktop ~]# vim /usr/local/sbin/mkvhost

Specify the interpreter program for the script.

```
#!/bin/bash
```

Set the variables for the arguments.

```
# Variables

VHOSTNAME=$1

TIER=$2
```

Set the other variables.

```
HTTPDCONF=/etc/httpd/conf/httpd.conf

VHOSTCONFDIR=/etc/httpd/conf.vhosts.d

DEFVHOSTCONFFILE=$VHOSTCONFDIR/00-default-vhost.conf

VHOSTCONFFILE=$VHOSTCONFDIR/$VHOSTNAME.conf

WWWROOT=/srv

DEFVHOSTDOCROOT=$WWWROOT/default/www

VHOSTDOCROOT=$WWWROOT/$VHOSTNAME/www
```

Check the argument values in the **VHOSTNAME** and **TIER** variables. If either is blank, display the message "Usage: **mkvhost VHOSTNAME TIER**" and exit with a status of 1. If the arguments are passed correctly, then use a case statement to set a **VHOSTADMIN** variable to the proper support email address, based on the value of **$TIER**. The case statement will use the **$TIER** values of 1, 2, and 3 to set **VHOSTADMIN** to the corresponding support email address. If any other **$TIER** value is encountered, the case statement should display the message "Invalid tier specified." and exit with a status of 1.

TIER	$VHOSTADMIN
1	basic_support@example.com
2	business_support@example.com
3	enterprise_support@example.com

Create the **if/then/else/fi** statement. Use an OR conditional to check whether either of the arguments is blank and, if so, display the usage message and exit with a status of 1.

```
# Check arguments
```

```
if [ "$VHOSTNAME"  = '' ] || [ "$TIER" = '' ]; then
    echo "Usage: $0 VHOSTNAME TIER"
    exit 1
else
```

Create the case statement

```
    # Set support email address
    case $TIER in

    1)      VHOSTADMIN='basic_support@example.com'
            ;;
    2)      VHOSTADMIN='business_support@example.com'
            ;;
    3)      VHOSTADMIN='enterprise_support@example.com'
            ;;
    *)      echo "Invalid tier specified."
```

```
            exit 1
    ;;
    esac
```

Close the if statement.

```
fi
```

Check to see if the $VHOSTCONFDIR directory is non existent. If so, create the directory.

```
if [ ! -d $VHOSTCONFDIR ]; then
        mkdir -p $VHOSTCONFDIR
        restorecon -Rv $VHOSTCONFDIR
fi
```

IncludeOptional conf\.vhosts\.d/*\.conf

```
# Add include one time if missing
```

```
grep -q '^IncludeOptional conf\.vhosts\.d/\*\.conf$' $HTTPDCONF

if [ $? -ne 0 ]; then
        # Backup before modifying
        cp -a $HTTPDCONF $HTTPDCONF.orig
        echo "IncludeOptional conf.vhosts.d/*.conf" >> $HTTPDCONF
        mkdir $VHOSTCONFDIR

        if [ $? -ne 0 ]; then
                echo "ERROR: Failed adding include directive."
                exit 1
         fi
fi
```

Check to see if a default virtual host already exists and, if not, create it.

Verify if the default virtual host configuration file already exists and, if not, create and populate it with the following statement:

```
# Check for default virtual host

if [ ! -f $DEFVHOSTCONFFILE ]; then
    cat <<DEFCONFEOF > $DEFVHOSTCONFFILE
    <VirtualHost _default_:80>
      DocumentRoot $DEFVHOSTDOCROOT
      CustomLog "logs/default-vhost.log" combined
    </VirtualHost>

    <Directory $DEFVHOSTDOCROOT>
      Require all granted
    </Directory>
```

```
    DEFCONFEOF
fi
```

Verify if the default virtual host document root directory already exists and, if not, create it.

```
if [ ! -d $DEFVHOSTDOCROOT ]; then
        mkdir -p $DEFVHOSTDOCROOT
        restorecon -Rv $WWWROOT
fi
```

Check to see if the virtual host's configuration file already exists and, if so, display the error message "ERROR: $VHOSTCONFFILE already exists." and exit with a status of 1. Check to see if the virtual host's document root directory already exists and, if so, display the error message "ERROR: $VHOSTDOCROOT already exists." and exit with a status of 1. If no errors are encountered with the previous two checks, continue with the creation of

the virtual host configuration file, $VHOSTCONFFILE, and document root directory, $VHOSTDOCROOT. Populate the virtual host configuration file with the following here statement:

```
# Check for virtual host conflict
if [ -f $VHOSTCONFFILE ]; then
        echo "ERROR: $VHOSTCONFFILE already exists."
        exit 1
elif [ -d $VHOSTDOCROOT ]; then
        echo "ERROR: $VHOSTDOCROOT already exists."
exit 1
else
    cat <<CONFEOF > $VHOSTCONFFILE
    <Directory $VHOSTDOCROOT>
      Require all granted
      AllowOverride None
    </Directory>
    <VirtualHost *:80>
```

```
  DocumentRoot $VHOSTDOCROOT
  ServerName $VHOSTNAME
  ServerAdmin $VHOSTADMIN
  ErrorLog "logs/${VHOSTNAME}_error_log"
  CustomLog "logs/${VHOSTNAME}_access_log" common
</VirtualHost>
CONFEOF
mkdir -p $VHOSTDOCROOT
restorecon -Rv $WWWROOT
cat <<INDEXHTMLEOF > $VHOSTCONFFILE/index.html
LAB - ENHANCING BASH SHELL SCRIPTS WITH CONDITIONALS AND CONTROL STRUCTURES
INDEXHTMLEOF
chown -R apache $WWWROOT
```

add a sample index.html file

```
fi
```

Verify the syntax of the configuration file with the command apachectl configtest. If there are no errors, then reload httpd.service. Otherwise, display the error message "ERROR: Configuration error." and exit with a status of 1.

```
# Check config and reload
apachectl configtest &> /dev/null
if [ $? -eq 0 ]; then
        systemctl reload httpd &> /dev/null
else
        echo "ERROR: Config error."
        exit 1
fi
```

Save and execute the script.

Make the script executable.

```
[root@desktop ~]# chmod u+x /usr/local/sbin/mkvhost
```

Execute the script.

```
[root@desktop ~]# /usr/local/sbin/mkvhost www.example.com 3
```

Test:

```
[root@desktop ~]# cat "127.0.0.1   www.example.com" >> /etc/hosts
[root@desktop ~]# curl www.example.com
```

LAB - ENHANCING BASH SHELL SCRIPTS WITH CONDITIONALS AND CONTROL STRUCTURES

LAB - BASH CONDITIONALS AND CONTROL STRUCTURES

Your company provides hosting service to customers, and currently uses a Bash shell script called **/usr/local/sbin/mkaccounts** to automate the daily task of creating accounts for new customers by processing a colon-separated data file located at **/tmp/support/newusers**.

You have been tasked with extending the script to add the following new features:

• The script should accept a command-line argument so that it can either be run in verbose mode or generate a usage message.

• When in verbose mode, the script should generate a message to indicate the creation of each account.

• Prior to the creation of an account, the script should check existing accounts and report if a conflict or duplication occurs.

1. Begin by extending the script located at **/usr/local/sbin/mkaccounts** by storing the first argument to the **OPTION** variable.

Back up the script prior to making edits.

```
[root@desktop ~]# cp -a /usr/local/sbin/mkaccounts
/usr/local/sbin/mkaccounts.orig
```

After the existing line declaring the **NEWUSERSFILE** variable, add a new variable, **OPTION**, to store the first argument

```
OPTION=$1
```

2. After the variable declaration at the top of the script, add a case statement that expects three patterns for the value of the OPTION variable and takes the following corresponding actions.

PATTERN	ACTION
Blank value	No action.
-v	Set variable VERBOSE to value of 'y'.
-h	Display usage message "Usage: mkaccounts [-h|-v]" and exit.
All other patterns	Display usage message "Usage: mkaccounts [-h|-v]" and exit with status 1.

```
case $OPTION in
    '')
         ;;
     -v)
         VERBOSE=y
         ;;
     -h)
         echo "Usage: $0 [-h|-v]"
         echo
         exit
         ;;
    *)
```

```
        echo "Usage: $0 [-h|-v]"
        echo
        exit 1
        ;;
esac
```

3. With in the existing for loop, prior to creating the new user account,check whether an account of the same name already exists. If an account already exists, set the value of the ACCTEXIST variable to 'y'. Store the value of the GECOS field of the existing account in the variable ACCTEXISTNAME.

```
# Test for dups and conflicts
   ACCTEXIST=''
   ACCTEXISTNAME=''

   id $ACCTNAME &> /dev/null

   if [ $? -eq 0 ]; then
           ACCTEXIST=y
           ACCTEXISTNAME="$(grep ^$ACCTNAME: /etc/passwd | cut -f5 -d:)"
fi
```

4. With in the existing for loop, uset he **ACCTEXIST** and **ACCTEXISTNAME** variables to perform the following evaluations and corresponding actions.

CONDIRION	ACTION
Account exists and the full name of the existing account matches that of a new customer.	Display message "Skipping <ACCOUNTNAME>. Duplicate found."
Account exists and the full name of the existing account does not match that of a new customer.	Display message "Skipping <ACCOUNTNAME>. Conflict found."
Account does not already exist.	Create the account. If the value of the variable VERBOSE is set to 'y', display message "Added <ACCOUNTNAME>."

```
# Test for dups and conflicts
            if [ "$ACCTEXIST" = 'y' ] && [ "$ACCTEXISTNAME" = "$FIRSTNAME
$LASTNAME" ]; then
                  echo "Skipping $ACCTNAME. Duplicate found."
```

```
            elif [ "$ACCTEXIST" = 'y' ]; then
                  echo "Skipping $ACCTNAME. Conflict found."
            else
                  useradd $ACCTNAME -c "$FIRSTNAME $LASTNAME"
                  if [ "$VERBOSE" =  'y' ]; then
                        echo "Added $ACCTNAME."
    fi
fi
```

5. Save and execute the script. Verify that the script performed as intended with each of the expected options. You may need to run /usr/local/sbin/rmaccounts to clear accounts that might have been created before rerunning your script.

Make the script executable.

```
[root@desktop ~]# chmod u+x /usr/local/sbin/mkaccounts
```

Execute the script with an invalid argument.

```
[root@desktop ~]# /usr/local/sbin/mkaccounts test
Usage: ./mkaccounts [-h|-v]

[root@desktop sbin]# echo $?
1
```

Execute the script with the -h argument.

```
[root@desktop ~]# /usr/local/sbin/mkaccounts -h
Usage: ./mkaccounts [-h|-v]

[root@desktop sbin]# echo $?
0
```

Execute the script with the -v argument.

```
[root@desktop ~]# mkaccounts -v
Added bwerts.
Added hballa.
[...]
Added rfabry.
Added pbamber.
"Tier 1","8","88%"
"Tier 2","1","11%"
"Tier 3","0","0%"
```

Execute the script again with the -v argument

```
[root@desktop ~]# mkaccounts -v
Skipping bwerts. Duplicate found.
Skipping hballa. Duplicate found.
```

```
[...]
Skipping rfabry. Duplicate found.
Skipping pbamber. Duplicate found.
"Tier 1","8","88%"
"Tier 2","1","11%"
"Tier 3","0","0%"
```

Execute script `rmaccounts` to remove users

```
[root@desktop ~]# rmaccounts
```

CHANGING THE SHELL ENVIRONMENT

ENVIRONMENT VARIABLES

The shell and scripts use variables to store data; some variables can be passed to sub-processes along with their content. These special variables are called environment variables.

Applications and sessions use these variables to determine their behavior. Some of them are likely familiar to administrators, such as **PATH**, **USER**, and **HOSTNAME**, among others. What makes variables environment variables is that they have been exported in the shell. A variable that is flagged as an export will be passed, with its value, to any sub-process spawned from the shell. Users can use the **env** command to view all environment variables that are defined in their shell.

Any variable defined in the shell can be an environment variable. The key to making a variable become an environment variable is flagging it for export using the **export** command.

In the following example, a variable, **MYVAR**, will be set. A sub-shell is spawned, and the **MYVAR** variable does not exist in the sub-shell.

```
[student@desktop ~]$ MYVAR="some value"
[student@desktop ~]$ echo $MYVAR
some value
[student@desktop ~]$ bash
[student@desktop ~]$ echo $MYVAR
[student@desktop ~]$ exit
```

In a similar example, the export command will be used to tag the MYVAR variable as an environment variable, which will be passed to a sub-shell.

```
[student@desktop ~]$ MYVAR="some value"
[student@desktop ~]$ export MYVAR
[student@desktop ~]$ echo $MYVAR
some value
[student@desktop ~]$ bash
[student@desktop ~]$ echo $MYVAR
```

```
some value
[student@desktop ~]$ exit
```

bash START-UP SCRIPTS

One place where environment variables are used is in initializing the **bash** environment upon user login. When a user logs in, several shell scripts are executed to initialize their environment, starting with the **/etc/profile**, followed by a profile in the user's home directory, typically **~/.bash_profile.**

Because these profiles have additional scripting in them, which calls other shell scripts, the **bash** login scripting will typically be the following:

```
/etc/profile
           \__ /etc/profile.d/*.sh
~/.bash_profile
              \__ ~/.bashrc
                        \__ /etc/bashrc
```

Generally, there are two types of login scripts, profiles and "RCs". Profiles are for setting and exporting of environment variables, as well as running commands that should only be run upon login. RCs, such as **/etc/bashrc**, are for running commands, setting aliases, defining functions, and other settings that cannot be exported to sub-shells. Usually, profiles are only executed in a login shell, whereas RCs are executed every time a shell is created, login or non-login.

The layout of the file call is such that a user can override the default settings provided by the system wide scripts. Many of the configuration files provided will contain a comment indicating where user-specific changes should be added.

USING alias

alias is a way administrators or users can define their own command to the system or override the use of existing system commands. Aliases are parsed and substituted prior to the shell checking **PATH**. **alias** can also be used to display all existing aliases defined in the shell.

```
[student@desktop ~]$ alias
alias egrep='egrep --color=auto'
```

```
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

All the default aliases defined in a user environment are in the previously shown output of **alias**. The definition of the **ll** alias means that when the user types **ll** as a command, the shell will expand the alias and execute **ls -l --color=auto**. In this way, a new command **ll** has been added to the shell. In another example, the alias for **grep**, the alias overrides the default invocation of an existing command on the system. When a user enters the grep command, the shell will expand the alias and substitute the **grep --color=auto** command. Due to the alias, all calls to **grep** are overridden to become calls to **grep** with the **--color** option passed automatically.

Use the **alias** command to set an alias. The defined alias will exist for the duration of the current shell only.

```
alias mycomm="<command to execute>"
```

```
[student@desktop ~]$ alias usercmd='echo "Hurrah!"; ls -l'
[student@desktop ~]$ usercmd
Hurrah!
total 0
-rw-rw-r--. 1 student student 0 Jun  9 13:21 file1
-rw-rw-r--. 1 student student 0 Jun  9 13:21 file2
-rw-rw-r--. 1 student student 0 Jun  9 13:21 file3
```

To make the alias persistent, the user would need to add the command to the bottom of their **~/.bashrc.**

```
[student@desktop ~]$ vi ~/.bashrc
...
# User specific aliases and functions alias
```

```
usercmd='echo "Hurrah!"; ls -l'
```

After the **alias** is added to the **~/.bashrc**, it will be available in every shell created. To remove an alias from the environment, use the **unalias** command.

USING FUNCTIONS

Defining Bash Functions

The syntax for declaring a bash function is simple. Functions may be declared in two different formats:
The first format starts with the function name, followed by parentheses. This is the preferred and more used format.

```
function_name () {
  commands
}
```

Single line version:

```
function_name () { commands; }
```

The second format starts with the reserved word function, followed by the function name.

```
function function_name {
  commands
}
```

Single line version:

```
function function_name { commands; }
```

Few points to be noted:

- The commands between the curly braces ({}) are called the body of the function. The curly braces must be separated from the body by spaces or newlines.
- Defining a function doesn't execute it. To invoke a bash function, simply use the function name. Commands between the curly braces are executed whenever the function is called in the shell script.
- The function definition must be placed before any calls to the function.
- When using single line "compacted" functions, a semicolon ; must follow the last command in the function.
- Always try to keep your function names descriptive.

To understand this better, take a look at the following example:

```
1    #!/bin/bash
2
3    hello_world () {
4        echo 'hello, world'
5    }
6
7    hello_world
```

In line 3, we are defining the function by giving it a name. The curly brace { marks the start of the function's body.
Line 4 is the function body. The function body can contain multiple commands, statements and variable declarations.
Line 5, the closing curly bracket }, defines the end of the hello_world function.
In line 7 we are executing the function. You can execute the function as many times as you need.
If you run the script, it will print hello, world.

Variables Scope

Global variables are variables that can be accessed from anywhere in the script regardless of the scope. In Bash, all variables by default are defined as global, even if declared inside the function.

Local variables can be declared within the function body with the **local** keyword and can be used only inside that function. You can have local variables with the same name in different functions.
To better illustrate how variables scope works in Bash, let's consider this example:

```
#!/bin/bash

var1='A'
var2='B'

my_function () {
  local var1='C'
  var2='D'
  echo "Inside function: var1: $var1, var2: $var2"
}

echo "Before executing function: var1: $var1, var2: $var2"

my_function

echo "After executing function: var1: $var1, var2: $var2"
```

The script starts by defining two global variables var1 and var2. Then there is an function that sets a local variable var1 and modifies the global variable var2.

If you run the script, you should see the following output:

```
Before executing function: var1: A, var2: B
Inside function: var1: C, var2: D
After executing function: var1: A, var2: D
```

Return Values

Unlike functions in “real” programming languages, Bash functions don’t allow you to return a value when called. When a bash function completes, its return value is the status of the last statement executed in the function, 0 for success and non-zero decimal number in the 1 - 255 range for failure.
The return status can be specified by using the return keyword, and it is assigned to the variable $?. The return statement terminates the function.

```
#!/bin/bash

my_function () {
  echo "some result"
  return 55
}

my_function
echo $?
```

To actually return an arbitrary value from a function, we need to use other methods. The simplest option is to assign the result of the function to a global variable:

```
#!/bin/bash

my_function () {
  func_result="some result"
}

my_function
echo $func_result
```

Another, better option to return a value from a function is to send the value to stdout using echo or printf like shown below:

```
#!/bin/bash

my_function () {
  local func_result="some result"
  echo "$func_result"
```

```
}

func_result="$(my_function)"
echo $func_result
```

Passing Arguments to Bash Functions

To pass any number of arguments to the bash function simply put them right after the function's name, separated by a space. It is a good practice to double-quote the arguments to avoid the misparsing of an argument with spaces in it.

- The passed parameters are **$1, $2, $3 … $n**, corresponding to the position of the parameter after the function's name.
- The **$0** variable is reserved for the function's name.
- The **$#** variable holds the number of positional parameters/arguments passed to the function.
- The **$*** and **$@** variables hold all positional parameters/arguments passed to the function.
    - When double-quoted, **"$*"** expands to a single string separated by space (the first character of IFS) - **"$1 $2 $n"**.
    - When double-quoted, **"$@"** expands to separate strings - **"$1" "$2" "$n"**.
    - When not double-quoted, **$*** and **$@** are the same.

```
#!/bin/bash
```

```
greeting () {
  echo "Hello $1"
}

greeting "Joe"
```

A Bash function is a block of reusable code designed to perform a particular operation. Once defined, the function can be called multiple times within a script.

An example of defining and using a function within a shell script follows, taken from **/etc/profile**. The **pathmunge** function takes two arguments; the first is a directory, the second (optional) is the word "after". Based on whether "after" is passed as $2, the directory will be added to the **PATH** environment variable at the front or end of the existing list of directories. Later, the function is invoked several times to build the **PATH** for root or regular users. Notice that for root, all the directories are prepended to **PATH**, where regular users have their **PATH** built by appending.

```
pathmunge () {
```

```
  if [ "$2" = "after" ] ; then
    PATH=$PATH:$1
  else
    PATH=$1:$PATH
  fi
}

if [ "$EUID" = "0" ]; then
    pathmunge /sbin
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
else
    pathmunge /usr/local/sbin after
    pathmunge /usr/sbin after
    pathmunge /sbin after
fi
```

Functions can also be set in the **bash** shell environment. When set in the environment, they can be executed as commands on the command line, similar to aliases. Unlike aliases, they can take arguments, be much more sophisticated in their actions, and provide a return code. Functions can be defined in the current shell by typing them into the command line, but more realistically, they should be set in a user's **~/.bashrc** or the global **/etc/bashrc**.

FIREWALL

FIREWALLD OVERVIEW

firewalld is the default method in Fedora for managing host-level firewalls. Started from the **firewalld.service** systemd service, firewalld manages the Linux kernel netfilter subsystem using the **low-level iptables**, **ip6tables**, and **ebtables** commands.

firewalld separates all incoming traffic into **zones**, with each zone having its own set of rules. To check which zone to use for an incoming connection, firewalld uses this logic, where the first rule that matches wins:

1. If the source address of an incoming packet matches a source rule setup for a zone, that packet will be routed through that zone.

2. If the incoming interface for a packet matches a filter setup for a zone, that zone will be used.

3. Otherwise, the default zone is used. The default zone is not a separate zone; instead, it points to one of the other zones defined on the system.

Unless overridden by an administrator or a NetworkManager configuration, the default zone for any new network interface will be set to the public zone.

A number of predefined zones are shipped with firewalld, each with their own intended usage:

Default Configuration of firewalld Zones

ZONE NAME	DEFAULT CONFIGURATION
trusted	Allow all incoming traffic.
home	Reject incoming traffic unless related to outgoing traffic or matching the ssh, mdns, ipp-client, samba-client, or dhcpv6-client predefined services.
internal	Reject incoming traffic unless related to outgoing traffic or matching the ssh, mdns, ipp-client, samba-client, or dhcpv6-client predefined services (same as the home zone to start with).
work	Reject incoming traffic unless related to outgoing traffic or matching the ssh, ipp-client, or dhcpv6-client predefined services.
public	Reject incoming traffic unless related to outgoing traffic or matching the ssh or dhcpv6-client predefined services. The default zone for newly added network interfaces.
external	Reject incoming traffic unless related to outgoing traffic or matching the ssh predefined service. Outgoing IPv4 traffic forwarded through this zone is masqueraded to look like it originated from the IPv4 address of the outgoing network interface.

dmz	Reject incoming traffic unless related to outgoing traffic or matching the ssh predefined service.
block	Reject all incoming traffic unless related to outgoing traffic.
drop	Drop all incoming traffic unless related to outgoing traffic (do not even respond with ICMP errors).

MANAGING FIREWALLD

firewalld can be managed in three ways:

1. Using the command-line tool **firewall-cmd**.
2. Using the configuration files in **/etc/firewalld/**.

In most cases, editing the configuration files directly is not recommended, but it can be useful to copy configurations in this way when using configuration management tools.

Configure firewall settings with firewall-cmd

This section will focus on managing **firewalld** using the command-line tool **firewall-cmd**. **firewall-cmd** is installed as part of the main firewalld package. **firewall-cmd** can perform the same actions as **firewall-config**.

The following table lists a number of frequently used **firewall-cmd** commands, along with an explanation. Note that unless otherwise specified, almost all commands will work on the runtime configuration, unless the **--permanent** option is specified. Many of the commands listed take the **--zone=<ZONE>** option to determine which zone they affect. If **--zone** is omitted from those commands, the default zone is used.

While configuring a firewall, an administrator will normally apply all changes to the **--permanent** configuration, and then activate those changes with **firewall-cmd --reload**. While testing out new, and possibly dangerous, rules, an administrator can choose to work on the runtime configuration by omitting the **--permanent** option. In those cases, an extra option can be used to automatically remove a rule after a certain amount of time, preventing an administrator from accidentally locking out a system: **--timeout=<TIMEINSECONDS>**.

FIREWALL-CMD COMMANDS	EXPLANATION
--get-default-zone	Query the current default zone.
--set-default-zone=<ZONE>	Set the default zone. This changes both the runtime and the permanent configuration.
--get-zones	List all available zones.
--get-services	List all predefined services.
--get-active-zones	List all zones currently in use (have an interface or source tied to them), along with their interface and source information.
--add-source=<CIDR> [--zone=<ZONE>]	Route all traffic coming from the IP address or network/netmask <CIDR> to the specified zone. If no --zone= option is provided, the default zone will be used.
--remove-source=<CIDR> [--zone=<ZONE>]	Remove the rule routing all traffic coming from the IP address or network/netmask <CIDR> from the specified zone. If no --zone= option is provided, the default zone will be used.

--add-interface=<INTERFACE> [--zone=<ZONE>]	Route all traffic coming from <INTERFACE> to the specified zone. If no --zone= option is provided, the default zone will be used.
--change-interface=<INTERFACE> [--zone=<ZONE>]	Associate the interface with <ZONE> instead of its current zone. If no --zone= option is provided, the default zone will be used.
--list-all [--zone=<ZONE>]	List all configured interfaces, sources, services, and ports for <ZONE>. If no --zone= option is provided, the default zone will be used.
--list-all-zones	Retrieve all information for all zones (interfaces, sources, ports, services, etc.).
--add-service=<SERVICE>	Allow traffic to <SERVICE>. If no --zone= option is provided, the default zone will be used.
--add-port=<PORT/PROTOCOL>	Allow traffic to the <PORT/ PROTOCOL> port(s). If no --zone= option is provided, the default zone will be used.
--remove-service=<SERVICE>	Remove <SERVICE> from the allowed list for the zone. If no -- zone= option is provided, the default zone will be used.
--remove-port=<PORT/PROTOCOL>	Remove the <PORT/PROTOCOL> port(s) from the allowed list for the zone. If no --zone= option is provided, the default zone will be used.

--reload	Drop the runtime configuration and apply the persistent configuration.

firewall-cmd example

The following examples show the default zone being set to dmz, all traffic coming from the **192.168.0.0/24** network being assigned to the internal zone, and the network ports for mysql being opened on the internal zone.

```
[root@server ~]# firewall-cmd --set-default-zone=dmz
[root@server ~]# firewall-cmd --permanent --zone=internal
--add-source=192.168.0.0/24
[root@server ~]# firewall-cmd --permanent --zone=internal --add-service=mysql
[root@server ~]# firewall-cmd --reload
```

Firewalld configuration files

firewalld configuration files are stored in two places: **/etc/firewalld** and **/usr/lib/firewalld**. If a configuration file with the same name is stored in both locations, the version from **/etc/firewalld/** will be used. This allows administrators to override default zones and settings without fear of their changes being wiped out by a package update.

LAB - CONFIGURING A FIREWALL

As part of an ongoing project to track the designated caffe in a ted beverage fetcher of the day, you have been tasked with configuring a basic web server and firewall on your system.
Your setup must meet these requirements:
• The httpd and mod_ssl packages must be installed on system.
• The httpd.service must be enabled and started on system.
• Until your in-house web developers complete the actual application, the **web server** should serve a placeholder page with the text **COFFEE!**.
• firewalld must be enabled and started on system.
• The firewalld configuration on system must use the dmz zone zone for all unspecified connections.
• The work zone should have all the necessary ports for https opened, but unencrypted http traffic should be filtered.

Verify that firewalld is enabled and running on your system.

```
[student@server ~]$ systemctl status firewalld.service
```

Verify that the Loaded line ends in enabled and that the Active line specifies running. If this is not the case, enable and start the firewalld.service service using systemctl.

Install the httpd and mod_ssl packages

```
[student@server ~]$ yum install httpd mod_ssl firewalld
```

Enable and start the httpd.service service.

```
[root@desktop ~]$ systemctl enable httpd.service
[root@desktop ~]$ systemctl start httpd.service
```

Create the placeholder /var/www/html/index.html file with the contents COFFEE!

```
[root@desktop ~]$ vim /var/www/html/index.html
[root@desktop ~]$ cat /var/www/html/index.html
COFFEE!
```

Configure the firewalld daemon on your system to route all traffic through the dmz zone by default.

```
[root@desktop ~]$ firewall-cmd --set-default-zone=dmz
```

Inspect the configuration of the running firewall on your system.

```
[root@desktop ~]$ firewall-cmd --get-default-zone
dmz

[root@desktop ~]# firewall-cmd --get-active-zones
```

```
dmz
  interfaces: ens34

[root@desktop ~]# firewall-cmd --zone=dmz --list-all
dmz (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens34
  sources:
  services: ssh
  ports:
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

From your external system, open a shell and use curl to test access to https://SERVERIP:PORT. The connection should fail with a Connection refused error message.

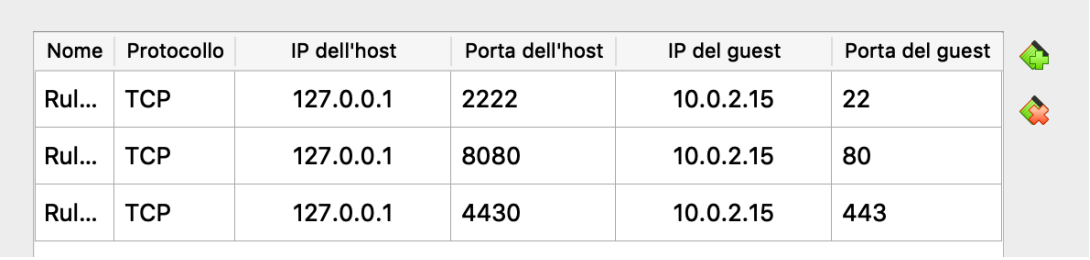
Since curl does not trust the placeholder self-signed certificate on server, you will have to use the -k option to skip certificate validation.

get the SERVERIP

```
[root@desktop ~]# ip a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
group default qlen 1000
    link/ether 08:00:27:1f:96:74 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 80942sec preferred_lft 80942sec
    inet6 fe80::a0d3:acb:f480:c771/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

VitrualBOX

add a new port forwarding from ip 127.0.0.1 on port 4430 to 10.0.2.15 on port 443



Nome	Protocollo	IP dell'host	Porta dell'host	IP del guest	Porta del guest
Rul...	TCP	127.0.0.1	2222	10.0.2.15	22
Rul...	TCP	127.0.0.1	8080	10.0.2.15	80
Rul...	TCP	127.0.0.1	4430	10.0.2.15	443

From your ssh client on your local system

```
curl -k https://127.0.0.1:4443
curl: (7) Failed to connect to 10.0.2.15 port 4443: Connection refused
```

Add https service to dmz zone

```
[root@desktop ~]# firewall-cmd --add-service=https
success
[root@desktop ~]# firewall-cmd --add-service=https --permanent
success
[root@desktop ~]# firewall-cmd --list-services --zone=dmz
ssh https
```

From your external system, open a shell and use curl to test access to https://SERVERIP:port. The connection now should show your content.

```
COFEE!!
```

Set the firewall zone to default

```
[root@desktop ~]# firewall-cmd --set-default-zone=public
success
```

MANAGING RICH RULES

RICH RULES CONCEPTS

Apart from the regular zones and services syntax that firewalld offers, administrators have two other options for adding firewall rules: direct rules and rich rules.

Direct rules

Configuring direct rules is not covered in this course, but documentation is available in the **firewall-cmd**(1) and **firewalld.direct**(5) man pages for those administrators who are already familiar with **{ip,ip6,eb}tables** syntax.

Rich rules

firewalld rich rules give administrators an expressive language in which to express custom firewall rules that are not covered by the basic **firewalld** syntax; for example, to only allow connections to a service from a single IP address, instead of all IP addresses routed through a zone.
Rich rules can be used to express basic allow/deny rules, but can also be used to configure logging, both to **syslog** and **auditd**, as well as port forwards, masquerading, and rate limiting.

The basic syntax of a rich rule can be expressed by the following block:

```
rule
  [source]
  [destination]
  service|port|protocol|icmp-block|masquerade|forward-port
  [log]
  [audit]
  [accept|reject|drop]
```

Almost every single element of a rule can take additional arguments in the form of option=value.

NOTE
For the full available syntax for rich rules, consult the **firewalld.richlanguage(5)** man page.

Rule ordering

Once multiple rules have been added to a zone (or the firewall in general), the ordering of rules can have a big effect on how the firewall behaves.
The basic ordering of rules inside a zone is the same for all zones:
1. Any port forwarding and masquerading rules set for that zone.

2. Any logging rules set for that zone.
3. Any deny rules set for that zone.
4. Any allow rules set for that zone.

In all cases, the first match will win. If a packet has not been matched by any rule in a zone, it will typically be denied, but zones might have a different default; for example, the trusted zone will accept any unmatched packet. Also, after matching a logging rule, a packet will continue to be processed as normal.
Direct rules are an exception. Most direct rules will be parsed before any other processing is done by firewalld, but the direct rule syntax allows an administrator to insert any rule they want anywhere in any zone.

Testing and debugging

To make testing and debugging easier, almost all rules can be added to the runtime configuration with a timeout. The moment the rule with a timeout is added to the firewall, the timer starts counting down for that rule. Once the timer for a rule has reached zero seconds, that rule is removed from the runtime configuration.
Using timeouts can be an incredibly useful tool while working on a remote firewalls, especially when testing more complicated rule sets. If a rule works, the administrator can add it again, but with the **--permanent** option (or at least without a timeout). If the rule does not work as intended, maybe even locking the administrator out of the system, it will be removed automatically, allowing the administrator to continue his or her work.
A timeout is added to a runtime rule by adding the option **--timeout=<TIMEINSECONDS>** to the end of the **firewall-cmd** that enables the rule.

WORKING WITH RICH RULES

firewall-cmd has four options for working with rich rules. All of these options can be used in combination with the regular **--permanent** or **--zone=<ZONE>** options.

OPTION	EXPLANATION
--add-rich-rule='<RULE>'	Add <RULE> to the specified zone, or the default zone if no zone is specified.
--remove-rich-rule='<RULE>'	Remove <RULE> to the specified zone, or the default zone if no zone is specified.
--query-rich-rule='<RULE>'	Query if <RULE> has been added to the specified zone, or the default zone if no zone is specified. Returns 0 if the rule is present, otherwise 1.
--list-rich-rules	Outputs all rich rules for the specified zone, or the default zone if no zone is specified.

Any configured rich rules are also shown in the output from **firewall-cmd --list-all** and **firewall-cmd --list-all-zones**.

Rich rules examples

Some examples of rich rules:

```
[root@server ~]# firewall-cmd --permanent --zone=classroom --add-rich-rule='rule family=ipv4 source address=192.168.0.11/32 reject'
```

Reject all traffic from the IP address **192.168.0.11** in the **classroom** zone.
When using **source** or **destination** with an **address** option, the **family=** option of **rule** must be set to either **ipv4** or **ipv6**.

```
[root@server ~]# firewall-cmd --add-rich-rule='rule family="ipv4" source address="192.168.0.11" port port=8080 protocol=tcp accept'
```

Allows port **8080** for a specific IP address **192.168.0.11** Note that this change is only made in the runtime configuration.

```
[root@server ~]# firewall-cmd --permanent --add-rich-rule='rule protocol
value=esp drop'
```

Drop all incoming IPsec **esp** protocol packets from anywhere in the default zone.

```
[root@serverX ~]# firewall-cmd --permanent --zone=vnc --add-rich-rule='rule
family=ipv4 source address=192.168.1.0/24 port port=7900-7905 protocol=tcp
accept'
```

Accept all TCP packets on ports **7900**, up to and including port **7905**, in the **vnc** zone for the **192.168.1.0/24** subnet.

LOGGING WITH RICH RULES

When debugging, or monitoring, a firewall, it can be useful to have a log of accepted or rejected connections. **firewalld** can accomplish this in two ways: by logging to **syslog**, or by sending messages to the kernel **audit** subsystem, managed by **auditd**.
In both cases, logging can be rate limited. Rate limiting ensures that system log files do not fill up with messages at a rate such that the system cannot keep up, or fills all its disk space.

The basic syntax for logging to **syslog** using rich rules is:

```
log [prefix="<PREFIX TEXT>"] [level=<LOGLEVEL>] [limit value="<RATE/DURATION>"]
```

Where **<LOGLEVEL>** is one of **emerg, alert, crit, error, warning, notice, info**, or **debug**.
<DURATION> can be one of **s** for seconds, **m** for minutes, **h** for hours, or **d** for days. For example, **limit value=3/m** will limit the log messages to a maximum of three per minute. The basic syntax for logging to the audit subsystem is:

```
audit [limit value="<RATE/DURATION>"]
```

Rate limiting is configured in the same way as for **syslog** logging.

Logging examples

Some examples of logging using rich rules:

```
[root@serverX ~]# firewall-cmd --permanent --zone=work --add-rich-rule='rule
service name="ssh" log prefix="ssh " level="notice" limit value="3/m" accept
```

Accept new connections to **ssh** from the **work** zone, log new connections to **syslog** at the **notice** level, and with a maximum of three message per minute.

```
[root@serverX ~]# firewall-cmd --add-rich-rule='rule family=ipv6 source
address="2001:db8::/64" service name="dns" audit limit value="1/h" reject'
--timeout=300
```

New IPv6 connections from the subnet **2001:db8::/64** in the default zone to DNS are rejected for the next five minutes, and rejected connections are logged to the **audit** system with a maximum of one message per hour.

LAB- WRITING CUSTOM RULES

Your company is running a trial that includes starting a web server on system. Since this could potentially generate many log entries, this logging should be limited to a maximum of three messages per second,and all log messages should be prefixed with the message "**NEW HTTP** ".
It has been decided that you, the IT Rock Star, will implement this using firewalld rich rules.

First install, start, and enable httpd. **(see LAB - CONFIGURING A FIREWALL)**

Configure a firewall rule in the **default** zone that allows traffic to http only from your client system. This traffic should be logged, but with a maximum of three new connections per second.

Permanently create the new firewall rule.

```
[root@desktop ~]# firewall-cmd --add-rich-rule='rule family="ipv4" service
name="http" log prefix="NEW HTTP " level="notice" limit value="3/s" accept'
```

Activate the changes to your firewall.

```
[root@desktop ~]# firewall-cmd --reload
```

On your system, use tail -f to view the additions to /var/log/messages in real time.

```
[root@desktop ~]# sudo tail -f /var/log/messages
```

From your client, use curl to connect to the httpd service running

```
localSystem ~> curl http://SERVERIP:SERVERPORT
COFEE!!
```

Inspect the output of your running tail command on your system. You should see a message for the new connection

Permanently delete the new firewall rule.

```
[root@desktop ~]# firewall-cmd --list-rich-rules
[root@desktop ~]# firewall-cmd --remove-rich-rule='rule…'
success
```

PORT FORWARDING

NETWORK ADDRESS TRANSLATION (NAT)

firewalld supports two types of Network Address Translation (NAT): masquerading and **port forwarding**. Both can be configured on a basic level with regular **firewall-cmd** rules, and more advanced forwarding configurations can be accomplished with rich rules. Both forms of NAT modify certain aspects of a packet, like the source or destination, before sending it on.

PORT FORWARDING

With port forwarding, traffic to a single port is forwarded either to a different port on the same machine, or to a port on a different machine. This mechanism is typically used to “hide” a server behind another machine, or to provide access to a service on an alternate port.

Assume that the machine with the IP address **10.0.0.100** behind the firewall is running a web server on port **8080/TCP,** and that the firewall is configured to forward traffic coming in on port **80/TCP** on its external interface to port **8080/TCP** on that machine.

1. A client from the Internet sends a packet to port **80/TCP** on the external interface of the firewall.
2. The firewall changes the destination address and port of this packet to **10.0.0.100** and **8080/TCP** and forwards it on. The source address and port remain unchanged.
3. The machine behind the firewall sends a response to this packet. Since this machine is being masqueraded (and the firewall is configured as the default gateway), this packet is sent to the original client, appearing to come from the external interface on the firewall.

Configuring port forwarding

To configure port forwarding with regular **firewall-cmd** commands, use the following syntax:

```
[root@serverX ~]# firewall-cmd --permanent --zone=<ZONE>
--add-forward-port=port=<PORTNUMBER>:proto=<PROTOCOL>[:toport=<PORTNUMBER>][:toad
dr=<IPADDR>]
```

Both the **toport=** and **toaddr=** parts are optional, but at least one of those two will need to be specified.

As an example, the following command will forward incoming connections on port **513/TCP** on the firewall to port **132/TCP** on the machine with the IP address **192.168.0.254** for clients from the public zone:

```
[root@serverX ~]# firewall-cmd --permanent --zone=public
--add-forward-port=port=513:proto=tcp:toport=132:toaddr=192.168.0.254
```

To gain more control over port forwarding rules, the following syntax can be used with rich rules:
forward-port port=<PORTNUM> protocol=tcp|udp [to-port=<PORTNUM>] [to- addr=<ADDRESS>]

An example that uses rich rules to forward traffic from **192.168.0.0/26** in the work zone to port **80/TCP** to port **8080/TCP** on the firewall machine itself:

```
[root@serverX ~]# firewall-cmd --permanent --zone=work --add-rich-rule='rule
family=ipv4 source address=192.168.0.0/26 forward-port port=80 protocol=tcp
to-port=8080'
```

LAB - FORWARDING A PORT

Your company is running a trial for a new bastion host. As part of this trial, you should be able to connect to the SSH daemon on your system on port 443/tcp. Since this is purely a trial, you do not wish to bind sshd to that port directly, but only to connect using port 443/tcp.

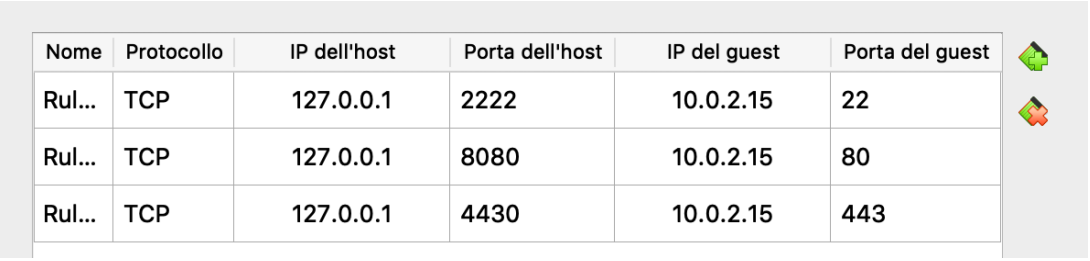
Configure the firewall on your system to forward port 443/tcp to 22/tcp

```
[root@desktop ~]# firewall-cmd --permanent --add-rich-rule 'rule family=ipv4
forward-port port=443 protocol=tcp to-port=22'
success
[root@desktop ~]# firewall-cmd --reload
success
[root@desktop ~]# firewall-cmd --list-rich-rules
rule family="ipv4" forward-port port="443" protocol="tcp" to-port="22"
```

Test if sshd is now available on port 443/tcp from your desktopX system.

```
ssh -l student SERVERIP -p 443
```

On **virtualbox** env add a new port forwarding from ip 127.0.0.1 on port 4430 to 10.0.2.15 on port 443



Nome	Protocollo	IP dell'host	Porta dell'host	IP del guest	Porta del guest
Rul...	TCP	127.0.0.1	2222	10.0.2.15	22
Rul...	TCP	127.0.0.1	8080	10.0.2.15	80
Rul...	TCP	127.0.0.1	4430	10.0.2.15	443

```
ssh -l student localhost/127.0.0.1 -p 4430
```

Clean the LAB:

```
firewall-cmd --remove-rich-rule='rule family="ipv4" forward-port port="443"
protocol="tcp" to-port="22"’ --permanent
firewall-cmd --reload
```

The yum package manager

Once a system is installed, additional software packages and updates are normally installed from a network package repository. The rpm command may be used to install, update, remove, and query RPM packages. However, it does not resolve dependencies automatically and all packages must be listed. Tools such as PackageKit and yum are front-end applications for rpm and can be used to install individual packages or package collections (sometimes called package groups) .
The yum command searches numerous repositories for packages and their dependencies so they may be installed together in an effort to alleviate dependency issues. The main configuration
file for yum is **/etc/yum.conf** with additional repository configuration files located in the **/etc/yum.repos.d** directory.
Repository configuration files include, at a minimum, a repo id (in square brackets), a name and the URL location of the package repository. The URL can point to a local directory (file) or remote network share (http, ftp, etc.). If the **URL** is pasted in a browser, the contents should display the RPM packages, possibly in one or more subdirectories, and a repodata directory with information about available packages.

The **yum** command is used to list repositories, packages, and package groups:

```
[root@fedora ~]# yum repolist
id repo                                                nome repo
fedora                                                 Fedora 36 - x86_64
fedora-cisco-openh264                                  Fedora 36 openh264 (From Cisco)
- x86_64
fedora-modular                                         Fedora Modular 36 - x86_64
updates                                                Fedora 36 - x86_64 - Updates
updates-modular                                        Fedora Modular 36 - x86_64 -
Updates
```

```
[root@fedora ~]# yum list installed
Pacchetti installati
ModemManager.x86_64                                    1.18.6-1.fc36
@anaconda
ModemManager-glib.x86_64                               1.18.6-1.fc36
@anaconda
NetworkManager.x86_64                                  1:1.36.4-1.fc36
@anaconda
```

```
NetworkManager-bluetooth.x86_64                                 1:1.36.4-1.fc36
@anaconda
NetworkManager-libnm.x86_64                                     1:1.36.4-1.fc36
@anaconda
[..]
```

Managing Software Updates with yum

yum is a powerful command-line tool that can be used to more flexibly manage (**install**, **update**, **remove**, and **query**) software packages. Official ContOS packages are normally downloaded from standard repository.

Finding software with yum

- **yum help** will display usage information.
- **yum list** displays installed and available packages.

```
[root@desktop ~]# yum list | head -20
```

```
Last metadata expiration check: 0:22:02 ago on Fri Oct 23 13:20:56 2020.
Installed Packages
NetworkManager.x86_64                                1:1.22.8-5.el8_2
@BaseOS
NetworkManager-libnm.x86_64                          1:1.22.8-5.el8_2
@BaseOS
NetworkManager-team.x86_64                           1:1.22.8-5.el8_2
@BaseOS
NetworkManager-tui.x86_64                            1:1.22.8-5.el8_2
@BaseOS
acl.x86_64                                           2.2.53-1.el8
@anaconda
adobe-mappings-cmap.noarch                           20171205-3.el8
@AppStream
adobe-mappings-cmap-deprecated.noarch                20171205-3.el8
@AppStream
adobe-mappings-pdf.noarch                            20180407-1.el8
@AppStream
apr.x86_64                                           1.6.3-9.el8
@AppStream
apr-util.x86_64                                      1.6.1-6.el8
@AppStream
```

```
apr-util-bdb.x86_64                                  1.6.1-6.el8
@AppStream
```

yum search **KEYWORD** lists packages by keywords found in the name and summary fields only.
To search for packages that have "web server" in their name, summary, and description fields, use search all:

```
[root@desktop ~]# yum search all "web server"
Failed to set locale, defaulting to C.UTF-8
Last metadata expiration check: 0:23:23 ago on Fri Oct 23 13:20:56 2020.
================================================== Summary & Description
Matched: web server ==================================================
pcp-pmda-weblog.x86_64 : Performance Co-Pilot (PCP) metrics from web server logs
nginx.x86_64 : A high performance web server and reverse proxy server
======================================================== Summary Matched:
web server ===========================================================
libcurl.x86_64 : A library for getting files from web servers
libcurl.i686 : A library for getting files from web servers
============================================================ Description
Matched: web server
============================================================
```

```
httpd.x86_64 : Apache HTTP Server
git-instaweb.x86_64 : Repository browser in gitweb
mod_security.x86_64 : Security module for the Apache HTTP Server
http-parser.i686 : HTTP request/response parser for C
[...]
```

yum info **PACKAGENAME** gives detailed information about a package, including the disk space needed for installation.

```
[root@desktop ~]# yum info httpd

Last metadata expiration check: 0:25:16 ago on Fri Oct 23 13:20:56 2020.
Installed Packages
Name         : httpd
Version      : 2.4.37
Release      : 21.module_el8.2.0+494+1df74eae
Architecture : x86_64
Size         : 4.9 M
Source       : httpd-2.4.37-21.module_el8.2.0+494+1df74eae.src.rpm
Repository   : @System
From repo    : AppStream
Summary      : Apache HTTP Server
```

```
URL          : https://httpd.apache.org/
License      : ASL 2.0
Description  : The Apache HTTP Server is a powerful, efficient, and extensible
             : web server.
```

yum provides PATHNAME displays packages that match the pathname specified (which often include wildcard characters).
To find packages that provide the **/var/www/html** directory, use:

```
[root@desktop ~]# yum provides vim
Failed to set locale, defaulting to C.UTF-8
Last metadata expiration check: 0:25:52 ago on Fri Oct 23 13:20:56 2020.
vim-enhanced-2:8.0.1763-13.el8.x86_64 : A version of the VIM editor which
includes recent enhancements
Repo        : @System
Matched from:
Provide    : vim = 8.0.1763-13.el8

vim-enhanced-2:8.0.1763-13.el8.x86_64 : A version of the VIM editor which
includes recent enhancements
```

```
Repo        : AppStream
Matched from:
Provide    : vim = 8.0.1763-13.el8
```

yum install **PACKAGENAME** obtains and installs a software package, including any dependencies.

yum update **PACKAGENAME** obtains and installs a newer version of the software package, including any dependencies. Generally the process tries to preserve configuration files in place, but in some cases, they may be renamed if the packager thinks the old one will not work after the update. With no PACKAGENAME specified, it will install **all** relevant updates.

```
[root@desktop ~]# yum update
Last metadata expiration check: 0:26:40 ago on Fri Oct 23 13:20:56 2020.
Dependencies resolved.
Nothing to do.
Complete!
```

Since a new kernel can only be tested by booting to that kernel, the package is specifically designed so that multiple versions may be installed at once. If the new kernel fails to boot. the old kernel is still available. Using yum update kernel will actually install the new kernel. The configuration files hold a list of packages to "always install" even if the administrator requests an update.

```
[root@desktop ~]# yum list kernel.x86_64
Last metadata expiration check: 0:28:08 ago on Fri Oct 23 13:20:56 2020.
Installed Packages
kernel.x86_64                4.18.0-147.el8                 @anaconda
kernel.x86_64                4.18.0-193.19.1.el8_2          @BaseOS
```

CONFIGURING APACHE HTTPD

INTRODUCTION TO APACHE HTTPD

Apache **HTTPD** is one of the most used web servers on the Internet. A web server is a daemon that speaks the **http(s)** protocol, a text-based protocol for sending and receiving objects over a network connection. The **http** protocol is sent over the wire in clear text, using port **80/TCP** by default (though other ports can be used). There is also a **TLS/SSL** encrypted version of the protocol called https that uses port **443/TCP** by default.
A basic http exchange has the client connecting to the server, and then requesting a resource using the **GET** command. Other commands like **HEAD** and **POST** exist, allowing clients to request just metadata for a resource, or send the server more information.

The following is an extract from a short http exchange:

```
GET /hello.html HTTP/1.1
Host: webapp0.example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20100101 Firefox/24.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
```

```
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cache-Control: max-age=0
```

The client starts by requesting a resource (the **GET** command), and then follows up with some extra headers, telling the server what types of encoding it can accept, what language it would prefer, etc. The request is ended with an empty line.

```
HTTP/1.1 200 OK
Date: Tue, 27 May 2014 09:57:40 GMT
Server: Apache/2.4.6 (Red Hat) OpenSSL/1.0.1e-fips mod_wsgi/3.4 Python/2.7.5
Content-Length: 12
Keep-Alive: timeout=5, max=82
Connection: Keep-Alive
Content-Type: text/plain; charset=UTF-8
Hello World!
```

The server then replies with a status code (**HTTP/1.1 200 OK**), followed by a list of headers.The **Content-Type** header is a mandatory one, telling the client what type of content is being sent. After the headers are done, the server sends an empty line, followed by the requested content. The length of this content must match the length indicated in the **Content-Length** header.

While the http protocol seems easy at first, implementing all of the protocol—along with security measures, support for clients not adhering fully to the standard, and support for dynamically generated pages—is not an easy task. That is why most application developers do not write their own web servers, but instead write their applications to be run behind a web server like Apache HTTPD.

About Apache HTTPD

Apache HTTPD, sometimes just called “Apache” or **httpd**, implements a fully configurable and extendable web server with full **http** support. The functionality of **httpd** can be extended with modules, small pieces of code that plug into the main web server framework and extend its functionality.
On Fedora Apache HTTPD is provided in the httpd package. The web-server package group will install not only the httpd package itself, but also the httpd-manual package. Once **httpd-manual** is installed, and the **httpd.service** service is started, the full Apache HTTPD manual is available on **http://localhost/manual**. This manual has a complete reference of all the configuration directives for **httpd**, along with examples. This makes it an invaluable resource while configuring **httpd**.
Fedora also ships an environment group called web-server-environment. This environment group pulls in the web-server group by default, but has a number of other groups, like backup tool and database clients, marked as optional.
A default dependency of the httpd package is the **httpd-tools** package. This package includes tools to manipulate password maps and databases, tools to resolve IP addresses in logfiles to hostnames, and a tool (**ab**) to benchmark and stress-test web servers.

BASIC APACHE HTTPD CONFIGURATION

After installing the web-server package group, or the httpd package, a default configuration is written to **/etc/httpd/conf/httpd.conf**.
This configuration serves out the contents of **/var/www/html** for requests coming in to any hostname over plain **http**.
The basic syntax of the **httpd.conf** is comprised of two parts:**Key Value** configuration directives, and HTML-like **<Blockname parameter>** blocks with other configuration directives embedded in them. Key/value pairs outside of a block affect the entire server configuration, while directives inside a block typically only apply to a part of the configuration indicated by the block, or when the requirement set by the block is met.

```
ServerRoot "/etc/httpd" 1
Listen 80 2
Include conf.modules.d/*.conf 3
User apache 4
Group apache 5
ServerAdmin root@localhost 6
<Directory /var/www/html> 7
    AllowOverride none
    Require all denied
```

```
</Directory>
DocumentRoot "/var/www/html" 8
<Directory "/var/www">
    AllowOverride None
    Require all granted
</Directory>
<Directory "/var/www/html">
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
<IfModule dir_module> 9
    DirectoryIndex index.html
</IfModule>
<Files ".ht*"> 10
    Require all denied
</Files>
ErrorLog "logs/error_log" 11
LogLevel warn
<IfModule log_config_module>
    LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\""
 combined
```

```
    LogFormat "%h %l %u %t \"%r\" %>s %b" common
    <IfModule logio_module>
      LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" %I
 %O" combinedio
</IfModule>
    CustomLog "logs/access_log" combined 12
</IfModule>
AddDefaultCharset UTF-8 13
IncludeOptional conf.d/*.conf 14
```

1 This directive specifies where **httpd** will look for any files referenced in the configuration files with a relative path name.
2 This directive tells **httpd** to start listening on port **80/TCP** on all interfaces. To only listen on select interfaces, the syntax "**Listen 1.2.3.4:80**" can be used for IPv4 or "**Listen [2001:db8::1]:80**" for IPv6.
Note: Multiple **listen** directives are allowed, but overlapping **listen** directives will result in a fatal error, preventing **httpd** from starting.
3 This directive includes other files, as if they were inserted into the configuration file in place of the **Include** statement. When multiple files are specified, they will be sorted by filename in alphanumeric order before being included. Filenames can either be absolute, or relative to **ServerRoot**, and include wildcards such as *****.
Note: Specifying a non existent file will result in a fatal error, preventing httpd from starting.

4 5 These two directives specify the user and group the **httpd** daemon should run as. **httpd** is always started as **root**, but once all actions that need root privileges have been performed— for example, binding to a port number under **1024**-privileges will be dropped and execution is continued as a non privileged user. This is a security measure.

6 Some error pages generated by **httpd** can include a link where users can report a problem. Setting this directive to a valid email address will make the webmaster easier to contact for users. Leaving this setting at the default of **root@localhost** is not recommended.

7 A **<Directory>** block sets configuration directives for the specified directory, and all descendent directories. Common directives inside a **<Directory>** block include the following:

- **AllowOverride None**:.**htaccess** files will not be consulted for per-directory configuration settings. Setting this to any other setting will have a performance penalty, as well as the possible security ramifications.
- **Require All Denied:httpd** will refuse to serve content out of this directory, returning a **HTTP/1.1 403 Forbidden** error when requested by a client.
- **Require All Granted**: Allow access to this directory. Setting this on a directory outside of the normal content tree can have security implications.
- **Options [[+|-]OPTIONS]...**: Turn on (or off) certain options for a directory. For example, the Indexes option will show a directory listing if a directory is requested and no **index.html** file exists in that directory.

8 This setting determines where **httpd** will search for requested files. It is important that the directory specified here is both readable by **httpd** (both regular permissions and SELinux), and that a corresponding **<Directory>** block has been declared to allow access.

9 This block only applies its contents if the specified extension module is loaded. In this case, the **dir_module** is loaded, so the **DirectoryIndex** directive can be used to specify what file should be used when a directory is requested.
10 A **<Files>** block works just as a **<Directory>** block, but here options for individual (wildcarded) files is used. In this case, the block prevents **httpd** from serving out any security-sensitive files like **.htaccess** and .**htpasswd**.
11 This specifies to what file **httpd** should log any errors it encounters. Since this is a relative pathname, it will be prepended with the **ServerRoot** directive. In a default configuration, **/etc/httpd/logs** is a symbolic link to **/var/log/httpd/**.
12 The **CustomLog** directive takes two parameters: a file to log to, and a log format defined with the **LogFormat** directive. Using these directives, administrators can log exactly the information they need or want. Most log parsing tools will assume that the default **combined** format is used.
13 This setting adds a **charset** part to the **Content-Type** header for **text/plain** and **text/html** resources. This can be disabled with **AddDefaultCharset Off**.
14 This works the same as a regular include, but if no files are found, no error is generated.

STARTING APACHE HTTPD

httpd can be started from the **httpd.service** systemd unit.

```
[root@desktop ~]# systemctl enable httpd.service
[root@desktop ~]# systemctl start httpd.service
```

Once httpd is started, status information can be requested with **systemctl status -l httpd.service**. If httpd has failed to start for any reason, this output will typically give a clear indication of why httpd failed to start.

Network security
firewalld has two predefined services for **httpd**. The **http** service opens port **80/TCP**, and the https service opens port **443/TCP**.

```
[root@serverX ~]# firewall-cmd --permanent --add-service=http --add-service=https
[root@serverX ~]# firewall-cmd --reload
```

LAB - CONFIGURING A WEB SERVER

You have been asked to configure a basic web server on your server machine. This web server should serve out the text **“Hello Class!”** when the URL http://SERVERIP/ is requested.
To aid your end users in submitting bug reports, the default error page should include a mailto: reference to the email address webmaster@example.com.
Since your organization is planning on further customizing the behavior of this web server, the full Apache httpd manual should be available under http://SERVERIP/manual/.

Begin by installing the httpd and httpd-manual packages.

```
[root@server ~]# sudo yum -y install httpd httpd-manual
```

Set the Server Admin directive for the main site configuration to point to webmaster@example.com.

Open /etc/httpd/conf/httpd.conf in a text editor with root privileges, and change the line that starts with ServerAdmin to the following:

```
ServerAdmin webmaster@example.com
```

Create the default content page.

- Create the **/var/www/html/index.html** file with a text editor as user root, and add the following content:

```
Hello Class!
```

Start and enable the httpd service.

```
[root@server ~]# systemctl start httpd.service
[root@server ~]# systemctl enable httpd.service
```

Open all the relevant ports for http on the firewall on server.

```
[root@server ~]# firewall-cmd --permanent --add-service=http
[root@server ~]# firewall-cmd --reload
```

Test if you can access the new static page, as well as the Apache httpd manual, from your desktop machine.
http://127.0.0.1:PORT (8080?)

CONFIGURING AND TROUBLESHOOTING VIRTUAL HOSTS

VIRTUAL HOSTS

Virtual hosts allow a single **httpd** server to serve content for multiple domains. Based on either the IP address of the server that was connected to, the hostname requested by the client in the http request, or a combination of both, **httpd** can use different configuration settings, including a different **DocumentRoot**.
Virtual hosts are typically used when it is not cost-effective to spin up multiple (virtual) machines to serve out many low-traffic sites; for example, in a shared hosting environment.

CONFIGURING VIRTUAL HOSTS

Virtual hosts are configured using **<VirtualHost>** blocks inside the main configuration. To ease administration, these virtual host blocks are typically not defined inside **/etc/httpd/conf/httpd.conf,** but rather in separate .conf files in **/etc/httpd/conf.d/.**
The following is an example file, **/etc/httpd/conf.d/site1.conf.**

```
<Directory /srv/site1/www> 1
  Require all granted
  AllowOverride None
</Directory>
<VirtualHost 192.168.0.1:80> 2
  DocumentRoot /srv/site1/www 3
  ServerName site1.example.com 4
  ServerAdmin webmaster@site1.example.com 5
  ErrorLog "logs/site1_error_log" 6
  CustomLog "logs/site1_access_log" combined 7
</VirtualHost>
```

1 This block provides access to the **DocumentRoot** defined further down.
2 This is the main tag of the block. The **192.168.0.1:80** part indicates to **httpd** that this block should be considered for all connections coming in on that **IP/port** combination.

3 Here the DocumentRoot is being set, but only for within this virtual host.
4 This setting is used to configure name-based virtual hosting. If multiple **<VirtualHost>** blocks are declared for the same **IP/port** combination, the block that matches ServerName with the hostname: header sent in the client **http** request will be used.
There can be exactly zero or one **ServerName** directives inside a single **<VirtualHost>** block. If a single virtual host needs to be used for more than one domain name, one or more **ServerAlias** statements can be used.
5 To help with sorting mail messages regarding the different websites, it is helpful to set unique **ServerAdmin** addresses for all virtual hosts.
6 The location for all error messages related to this virtual host.
7 The location for all access messages regarding this virtual host.

If a setting is not made explicitly for a virtual host, the same setting from the main configuration will be used.

Name-based vs. IP-based virtual hosting

By default, every virtual host is an IP-based virtual host, sorting traffic to the virtual hosts based on what IP address the client had connected to. If there are multiple virtual hosts declared for
a single IP/port combination, the **ServerName** and **ServerAlias** directives will be consulted, effectively enabling name-based virtual hosting.

Wildcards and priority

The IP address part of a <VirtualHost> directive can be replaced with one of two wildcards: **_default_** and *****.
Both have exactly the same meaning: “Match Anything”.
When a request comes in, **httpd** will first try to match against virtual hosts that have an explicit IP address set. If those matches fail, virtual hosts with a wildcard IP address are inspected. If there is still no match, the “main” server configuration is used.
If no exact match has been found for a **ServerName** or **ServerAlias** directive, and there are multiple virtual hosts defined for the IP/port combination the request came in on, the first virtual host that matches an IP/port is used, with first being seen as the order in which virtual hosts are defined in the configuration file.
When using multiple ***.conf** files, they will be included in alphanumeric sorting order. To create a catch-all (default) virtual host, the configuration file should be named something like **00-default.conf** to make sure that it is included before any others.

TROUBLESHOOTING VIRTUAL HOSTS

When troubleshooting virtual hosts, there are a number of approaches that can help.

- Configure a separate **DocumentRoot** for each virtual host,with identifying content.
- Configure separate log files, both for error logging and access logging, for each virtual host.
- Evaluate the order in which the virtual host definitions are parsed by **httpd**. Included files are read in alphanumeric sort order based on their filenames.

- Disable virtual hosts one by one to isolate the problem. Virtual host definitions can be commented out of the configuration file(s), and include files can be temporarily renamed to something that does not end in **.conf**.
- **journalctl UNIT=httpd.service** can isolate log messages from just the httpd.service service.

LAB - CONFIGURING A VIRTUAL HOST

Over the past few years, your company has been spinning up many web servers for new projects. Unfortunately, there was no structure or coordination between the various projects.

In an effort to clean up the mess, you have been asked to consolidate these various web servers into one, serving out the different domains using name-based virtual hosting.

For now, you will only have to set up a default virtual host that serves out a placeholder site from **/srv/default/www/**, and a virtual host for www.example.com that serves out content from **/srv/www.class-example.com/www**.

Create the content directories. The placeholder site should have an **index.html** file that reads:

```
Coming Soon!
```

The www.example.com site should have an **index.html** that reads:

```
www.class-example.com
```

Create new line in your **GUEST** **/etc/hosts Fedora** system

```
HOSTIP  www  www.class-example.com default.class-example.com default
```

“ip a s” to get HOSTIP

From you **HOST** machine

add a new line in /etc/host with content (linux,MacOs)

```
127.0.0.1 www  www.class-example.com default.class-example.com default
```

to add local dns entry in windows https://gist.github.com/zenorocha/18b10a14b2deb214dc4ce43a2d2e2992

Create the directories.

```
[root@server ~]# sudo mkdir -p /srv/{default,www.class-example.com}/www
```

Create the **index.html** files using a text editor. **/srv/default/www/index.html** gets the “Coming Soon!” text, and the **/srv/www.class-example.com/www/index.html** file should read “www”.

Disable selinux

```
[root@server ~]# setenforce 0
```

Disable selinux at boot

edit /etc/selinux/config with entry:

```
SELINUX=disabled
```

Create a new virtual host definition for the **_default_:80** virtual host. This virtual host should serve out content from **/srv/default/www/**, and log to **logs/default-vhost.log** using the combined format.
Create a new file called **/etc/httpd/conf.d/00-default-vhost.conf**. Give it the following content:

```
<VirtualHost _default_:80>
  DocumentRoot /srv/default/www
  CustomLog "logs/default-vhost.log" combined
</VirtualHost>
```

Since in a default configuration, http db locks access to all directories, you will need to open up the content directory for your default vhost. Add the following block to **/etc/httpd/conf.d/00-default-vhost.conf.**

```
<Directory /srv/default/www>
  Require all granted
</Directory>
```

Create a new virtual host definition for a **www.class-example.com** virtual host in **/etc/httpd/conf.d/01-www.class-example.com-vhost.conf.** This virtual host should respond to requests for both **www.class-example.com** and **www**, serve out content from **/srv/www.class-example.com/www**, and store logs in l**ogs/www.class-example.com.log.**
Create the file **/etc/httpd/conf.d/01-www.class-example.com-vhost.conf** with the following contents:

```
<VirtualHost *:80>
    ServerName www.class-example.com
```

```
    ServerAlias www
    DocumentRoot /srv/www.class-example.com/www
    CustomLog "logs/www.class-example.com.log" combined
</VirtualHost>
<Directory /srv/www.class-example.com/www>
    Require all granted
</Directory>
```

Start and enable the httpd service.

```
[root@server ~]# systemctl start httpd.service
[root@server ~]# sudo systemctl enable httpd.service
```

Open up the firewall on server to allow traffic to the httpd service

```
[root@server ~]# firewall-cmd --permanent --add-service=http
[root@server ~]# firewall-cmd --reload
```

From your HOST system test the result on your local shell with curl

```
curl http://127.0.0.1:8080
Coming Soon!!!
curl http://default.class-example.com:8080
Coming Soon!!!
curl http://www.class-example.com:8080
```

```
www.class-example.com
curl http://www:8080
www.class-example.com
```

Try also on your local browser

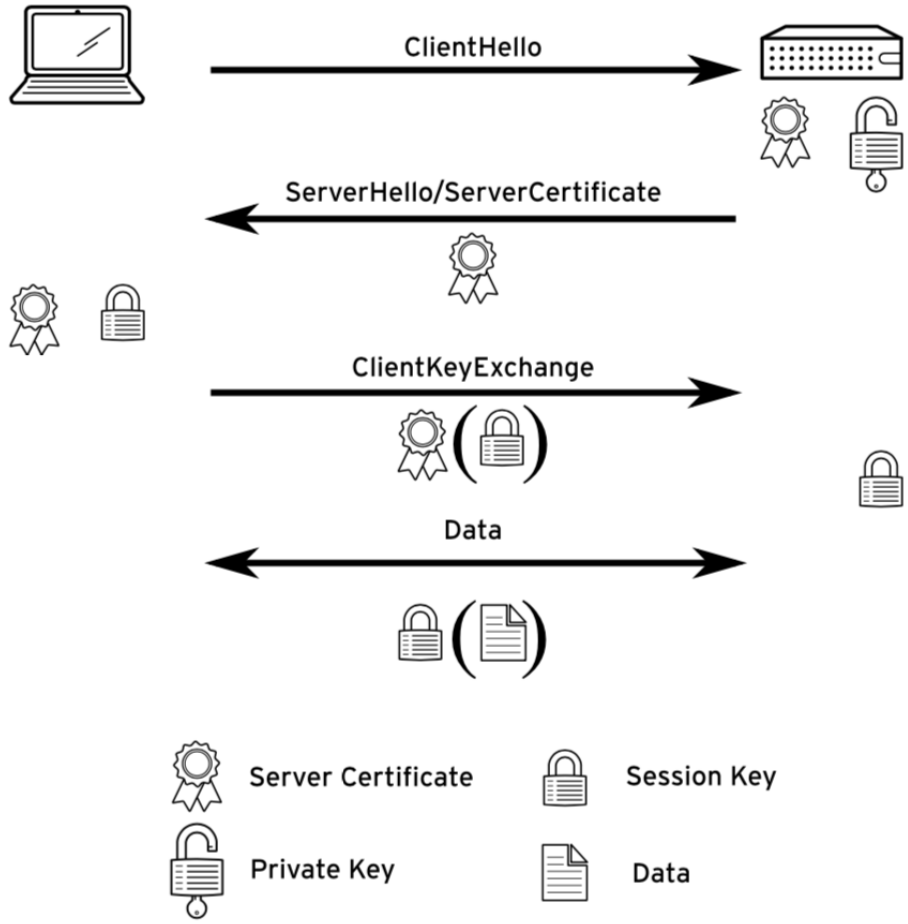
```
http://127.0.0.1:8080
http://default.class-example.com:8080
http://www.class-example.com:8080
http://www:8080
```

CONFIGURING HTTPS

TRANSPORT LAYER SECURITY (TLS)

Transport Layer Security (TLS) is a method for encrypting network communications. TLS is the successor to Secure Sockets Layer (SSL). TLS allows a client to verify the identity of the server and, optionally, allows the server to verify the identity of the client.
TLS is based around the concepts of certificates. A certificate has multiple parts: a public key, server identity, and a signature from a certificate authority. The corresponding private key is never made public. Any data encrypted with the private key can only be decrypted with the public key, and vice versa.
During the initial handshake, when setting up the encrypted connection, the client and server agree on a set of encryption ciphers supported by both the server and the client, and they exchange bits of random data. The client uses this random data to generate a session key, a key that will be used for much faster symmetric encryption, where the same key is used for both encryption and decryption. To make sure that this key is not compromised, it is sent to the server encrypted with the server's public key (part of the server certificate).

The following diagram shows a (simplified) version of a TLS handshake.



1 The client initiates a connection to the server with a ClientHello message. As part of this message, the client sends a 32-byte random number including a timestamp, and a list of encryption protocols and ciphers supported by the client.

2 The server responds with a **ServerHello** message, containing another 32-byte random number with a timestamp, and the encryption protocol and ciphers the client should use.

The server also sends the server certificate, which consists of a public key, general server identity information like the FQDN, and a signature from a trusted certificate authority (CA). This certificate can also include the public certificates for all certificate authorities that have signed the certificate, up to a root CA.

3 The client verifies the server certificate by checking if the supplied identity information matches, and by verifying all signatures, checking if they are made by a CA trusted by the client.
if the certificate verifies, the client creates a session key using the random numbers previously exchanged. The client then encrypts this session key using the public key from the server certificate, and sends it to the server using a **ClientKeyExchange** message.

4 The server decrypts the session key, and the client and server both start encrypting and decrypting all data sent over the connection using the session key.

CONFIGURING TLS CERTIFICATES

To configure a virtual host with TLS, multiple steps must be completed:

1. Obtain a (signed) certificate.
2. Install Apache HTTPD extension modules to support TLS.
3. Configure a virtual host to use TLS, using the certificates obtained earlier.

Obtaining a certificate

When obtaining a certificate, there are two options: creating a self-signed certificate (a certificate signed by itself, not an actual CA), or creating a certificate request and having a reputable CA sign that request so it becomes a certificate.
The **crypto-utils** package contains a utility called **genkey** that supports both methods. To create a certificate (signing request) with **genkey**, run the following command, where **<FQDN>** is the fully qualified domain name clients will use to connect to your server:

```
[root@server ~]# genkey <FQDN>
```

During the creation, **genkey** will ask for the desired key size (choose at least **2048** bits), if a signing request should be made (answering no will create a self-signed certificate), whether the private key should be protected with a passphrase, and general information about the identity of the server.
After the process has completed, a number of files will be generated:

/etc/pki/tls/private/<fqdn>.key: This is the private key. The private key should be kept at **0600** or **0400** permissions, and an SELinux context of **cert_t**. This key file should never be shared with the outside world.

/etc/pki/tls/certs/<fqdn>.0.csr: This file is only generated if you requested a signing request. This is the file that you send to your CA to get it signed. You never need to send the private key to your CA.

/etc/pki/tls/certs/<fqdn>.crt: This is the public certificate. This file is only generated when a self-signed certificate is requested. If a signing request was requested and sent to a CA, this is the file that will be returned from the CA. Permissions should be kept at **0644**, with an SELinux context of **cert_t**.

Install Apache HTTPD modules

Apache HTTPD needs an extension module to be installed to activate TLS support. On Fedora, you can install this module using the mod_ssl package.

This package will automatically enable **httpd** for a default virtual host listening on port **443/TCP**. This default virtual host is configured in the file **/etc/httpd/conf.d/ssl.conf.**

Configure a virtual host with TLS

Virtual hosts with TLS are configured in the same way as regular virtual hosts, with some additional parameters. It is possible to use name-based virtual hosting with TLS, but some older browsers are not compatible with this approach.
The following is a simplified version of **/etc/httpd/conf.d/ssl.conf**:

```
Listen 443 https 1
SSLPassPhraseDialog exec:/usr/libexec/httpd-ssl-pass-dialog 2
SSLSessionCache         shmcb:/run/httpd/sslcache(512000)
SSLSessionCacheTimeout  300
SSLRandomSeed startup file:/dev/urandom  256
SSLRandomSeed connect builtin
SSLCryptoDevice builtin
<VirtualHost _default_:443> 3
  ErrorLog logs/ssl_error_log
  TransferLog logs/ssl_access_log
  LogLevel warn
```

```
  SSLEngine on 4
  SSLProtocol all -SSLv2 5
  SSLCipherSuite HIGH:MEDIUM:!aNULL:!MD5 6
  SSLCertificateFile /etc/pki/tls/certs/localhost.crt 7
  SSLCertificateKeyFile /etc/pki/tls/private/localhost.key 8
  CustomLog logs/ssl_request_log \
          "%t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\" %b"
</VirtualHost>
```

1 This directive instructs **https** to listen on port **443/TCP**. The second argument (**https**) is optional, since **https** is the default protocol for port **443/TCP**.
2 If the private key is encrypted with a passphrase, **httpd** needs a method of requesting a passphrase from a user at the console at startup. This directive specifies what program to execute to retrieve that passphrase.
3 This is the virtual host definition for a catch-all virtual host on port **443/TCP**.
4 This is the directive that actually turns on **TLS** for this virtual host.
5 This directive specifies the list of protocols that **httpd** is willing to speak with clients. For security, the older, unsafe SSLv3 protocol should also be disabled:

```
SSLProtocol all -SSLv2 -SSLv3
```

6 This directive lists what encryption ciphers httpd is willing to use when communicating with clients. The selection of ciphers can have big impacts on both performance and security.

7 This directive instructs **httpd** where it can read the certificate for this virtual host.
8 This directive instructs **httpd** where it can read the private key for this virtual host. **httpd** reads all private keys before privileges are dropped, so file permissions on the private key can remain locked down.

If a certificate signed by an CA is used, and the certificate itself does not have copies of all the CA certificates used in signing, up to a root CA, embedded in it, the server will also need to provide a certificate chain, a copy of all CA certificates used in the signing process concatenated together. The **SSLCertificateChainFile** directive is used to identify such a file.
When defining a new TLS-encrypted virtual host, it is not needed to copy the entire contents of **ssl.conf**. Only a **<VirtualHost>** block with the **SSLEngine On** directive, and configuration for certificates, is strictly needed. The following is an example of a name-based TLS virtual host:

```
<VirtualHost *:443>
  ServerName demo.class-example.com
  SSLEngine on
  SSLCertificateFile /etc/pki/tls/certs/demo.class-example.com.crt
  SSLCertificateKeyFile /etc/pki/tls/private/demo.class-example.com.key
  SSLCertificateChainFile /etc/pki/tls/certs/example-ca.crt
</VirtualHost>
```

This example misses some important directives such as **DocumentRoot**; these will be inherited from the main configuration.

WARNING
Not defining what protocols and ciphers can be used will result in **httpd** using default options for these. **httpd** defaults are not considered secure, and it is highly recommended to restrict both to a more secure subset.

CONFIGURING FORWARD SECRECY

As mentioned earlier, the client and the server select the encryption cipher to be used to secure the TLS connection based on a negotiation during the initial handshake. Both the client and the server must find a cipher that both sides of the communication support.
If a weaker encryption cipher has been used, and the private key of the server has been compromised—for example, after a server break-in or due to a bug in the cryptography code—an attacker could possibly decrypt a recorded session.
One way to protect against these types of attacks is to use ciphers that ensure forward secrecy. Sessions encrypted using ciphers with this characteristic can not be decrypted if the private key is compromised at some later date. Forward secrecy can be established by carefully tuning the allowed ciphers in the **SSLCipherSuite** directive, to preferentially pick ciphers that support forward secrecy which both the server and client can use.
The following is an example that, at the date of publication, was considered a very good set of ciphers to allow. This list prioritizes ciphers that perform the initial session key exchange using elliptic curve Diffie-Hellman (EECDH) algorithms which support forward secrecy before falling back to less secure algorithms. Using Diffie-Hellman, the actual session key is never transmitted, but rather calculated by both sides.

```
SSLCipherSuite "EECDH+ECDSA+AESGCM EECDH+aRSA+AESGCM EECDH+ECDSA+SHA384 EECDH
+ECDSA+SHA256 EECDH+aRSA+SHA384 EECDH+aRSA+SHA256 EECDH EDH+aRSA DES-CBC3-SHA !
RC4 !aNULL !eNULL !LOW !DES !MD5 !EXP !PSK !SRP !DSS"
SSLHonorCipherOrder On
```

The example also disables RC4, due to its increasing vulnerability. This has the somewhat negative side effect of removing server-side BEAST (CVE-2011-3389) mitigation for very old web clients that only support TLSv1.0 and earlier. DES-CBC3-SHA is used in place of RC4 as a "last resort" cipher for support of old Internet Explorer 8 / Microsoft Windows XP clients. (In addition, to protect against other issues, the insecure SSLv3 and SSLv2 protocols should also be disabled on the web server, as previously discussed in this section.)
The **SSLHonorCipherOrder** On directive instructs httpd to preferentially select ciphers based on the order of the SSLCipherSuite list, regardless of the order preferred by the client.

IMPORTANT
Security research is an always ongoing arms race. It is recommended that administrators re-evaluate their selected ciphers on a regular basis.

CONFIGURING HTTP STRICT TRANSPORT SECURITY (HSTS)

A common misconfiguration, and one that will result in warnings in most modern browsers, is having a web page that is served out over https include resources served out over clear-text http.
To protect against this type of misconfiguration, add the following line inside a <VirtualHost> block that has TLS enabled:

```
Header always set Strict-Transport-Security "max-age=15768000"
```

Sending this extra header informs clients that they are not allowed to fetch any resources for this page that are not served using TLS.

Another possible issue comes from clients connecting over **http** to a resource they should have been using **https** for.

Simply not serving any content over **http** would alleviate this issue, but a more subtle approach is to automatically redirect clients connecting over **http** to the same resource using **https**.
To set up these redirects, configure a **http** virtual host for the same **ServerName** and **ServerAlias** as the TLS protected virtual host (a catch-all virtual host can be used), and add the following lines inside the **<VirtualHost *:80>** block:

```
RewriteEngine on
RewriteRule ^(/.*)$ https://%{HTTP_HOST}$1 [redirect=301]
```

The **RewriteEngine on** directive turns on the URL rewrite module for this virtualhost, and the **RewriteRule** matches any resource **(^(/.*)$)** and redirects it using a http **Moved Permanently** message **([redirect=301])** to the same resource served out over https. The **%{HTTP_HOST}** variable uses the hostname that was requested by the client, while the **$1** part is a back-reference to whatever was matched between the first set of parentheses in the regular expression.

LAB - CONFIGURING A TLS-ENABLED VIRTUAL HOST

The website will need to be protected with **TLS**.
You have been asked to configure a web server on your server machine to host this site. This web server will need to host two virtual hosts: **https://www.class-example.com** and **https://webapp.class-example.com**. The non encrypted version of these two sites should send browsers an automatic redirect to the encrypted version.
The certificates can be created with **openssl** command.

Install both the httpd and mod_ssl packages.

```
[root@server ~]$ yum install httpd mod_ssl
```

Create the content directories

```
[root@server ~]$ mkdir -p /srv/{www,webapp}/www
```

Generate certificate using **openssl** tool

- Generate private key and certificate signing request

```
[root@server ~]$ openssl genrsa -passout pass:x -out
class-example.com.pass.key 2048
[root@server ~]$ openssl genrsa -passout pass:x -out
webapp.class-example.com.pass.key 2048
[root@server ~]$ openssl rsa -passin pass:x -in class-example.com.pass.key
-out  class-example.com.key
[root@server ~]$ openssl rsa -passin pass:x -in
webapp.class-example.com.pass.key -out webapp.class-example.com.key
[root@server ~]$ rm *class-example.com.pass.key
[root@server ~]$ openssl req -new -key class-example.com.key -out
class-example.com.csr
-----------------------------------------------------------------------
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
```

Country Name (2 letter code) [XX]:it
State or Province Name (full name) []:italy
Locality Name (eg, city) [Default City]:parma
Organization Name (eg, company) [Default Company Ltd]:Canestracci oil Spa
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:example.com
Email Address []:info@webapp.class-example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: <<<<<<<<<<<< leave empty
An optional company name []:

[root@server ~]$ openssl req -new -key webapp.class-example.com.key -out webapp.class-example.com.csr
---
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.

```
-----
Country Name (2 letter code) [XX]:it
State or Province Name (full name) []:italy
Locality Name (eg, city) [Default City]:parma
Organization Name (eg, company) [Default Company Ltd]:Canestracci oil Spa
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname)
[]:webapp.class-example.com
Email Address []:info@webapp.class-example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: <<<<<<<<<<<< leave empty
An optional company name []:
```

- Generate SSL certificate
  The self-signed SSL certificate is generated from the server.key private key and server.csr files.

```
[root@server ~]$ openssl x509 -req -sha256 -days 365 -in
webapp.class-example.com.csr -signkey webapp.class-example.com.key -out
webapp.class-example.com.crt
```

```
[root@server ~]$ openssl x509 -req -sha256 -days 365 -in
class-example.com.csr -signkey class-example.com.key -out
class-example.com.crt

[root@server ~]# mv *class-example.com.key /etc/pki/tls/private/
[root@server ~]# mv *class-example.com.csr /etc/pki/tls/certs/
[root@server ~]# mv *class-example.com.crt /etc/pki/tls/certs/
[root@server ~]# chmod 0600 /etc/pki/tls/private/*class-example.com.key
[root@server ~]# chmod 0600 /etc/pki/tls/certs/*class-example.com.crt
[root@server ~]# chmod 0600 /etc/pki/tls/certs/*class-example.com.csr
```

Install VIM

```
[root@server ~]$ yum install -y vim
```

In both content directories, create an index.html file with distinct content.

```
[root@server ~]$ vim /srv/www/www/index.html
[root@server ~]$ vim /srv/webapp/www/index.html
```

Add new line in /etc/hosts file

```
[root@server ~]$ SERVERIP www.class-example.com www.webapp.class-example.com
class-example.com webapp.class-example.com (“ip a s” to get the system ip)
```

Configure the TLS name-based virtual host for your **www.class-example.com** domain in a new file called **/etc/httpd/conf.d/www.conf**. You can use the existing **/etc/httpd/conf.d/ssl.conf** as a template, but if you do, do not forget to strip out all the content outside of the <VirtualHost> block.

Do not forget to add an automatic redirect from the non-TLS-based http site to the TLS-encrypted https site.

Create **/etc/httpd/conf.d/www.conf** with the following content:

```
<VirtualHost *:443>
    ServerName www.class-example.com
```

```
    SSLEngine On
    SSLCertificateFile /etc/pki/tls/certs/class-example.com.crt
    SSLCertificateKeyFile /etc/pki/tls/private/class-example.com.key
    DocumentRoot /srv/www/www
</VirtualHost>
```

Add a <Directory> block for **/srv/www/www** to **/etc/httpd/conf.d/www.conf** like the following:

```
<Directory /srv/www/www>
    Require all granted
</Directory>
```

To accomplish the automatic redirect from http to https, add the following block to **/etc/httpd/conf.d/www.conf**:

```
<VirtualHost *:80>
ServerName www.class-example.com
RewriteEngine on
RewriteRule ^(/.*)$ https://%{HTTP_HOST}$1 [redirect=301]
</VirtualHost>
```

Configure the **webapp.class-example.com** virtual host by copying the configuration for your **www.class-example.com** virtual host, and changing every occurrence of **www** to **webapp**.

```
[root@server ~]$ vim /etc/httpd/conf.d/webapp.conf
```

Replace every occurrence of **www** with **webapp** in the new configuration file.

```
<VirtualHost *:443>
ServerName webapp.class-example.com
SSLEngine On
SSLCertificateFile /etc/pki/tls/certs/webapp.class-example.com.crt
SSLCertificateKeyFile /etc/pki/tls/private/webapp.class-example.com.key
DocumentRoot /srv/webapp/www
</VirtualHost>

<Directory /srv/webapp/www>
Require all granted
</Directory>

<VirtualHost *:80>
ServerName webapp.class-example.com
RewriteEngine on
RewriteRule ^(/.*)$ https://%{HTTP_HOST}$1 [redirect=301]
</VirtualHost>
```

Start and enable the httpd.service, and open the relevant firewall ports.

```
[root@server ~]$ systemctl start httpd.service
[root@server ~]$ systemctl enable httpd.service
```

Open both the http and https ports on the firewall.

```
[root@server ~]$ sudo firewall-cmd --permanent --add-service=http --add-
service=https
[root@server ~]$ sudo firewall-cmd --reload
```

From you **HOST** machine

add a new line in /etc/host with content (linux,MacOs)

```
SERVERIP www.class-example.com class-example.com www.webapp.class-example.com
webapp.class-example.com (SERVERIP IS THE IP OF THE @SERVER VM)
```

how to add local dns entry in windows https://gist.github.com/zenorocha/18b10a14b2deb214dc4ce43a2d2e2992

```
From ssh client
curl -k http://webapp.class-example.com:8080
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
```

```
<h1>Moved Permanently</h1>
<p>The document has moved <a
href="https://webapp.class-example.com/">here</a>.</p>
</body></html>
From ssh client
curl  -k https://webapp.class-example.com:4443
```
WEBAPP!!!!

```
curl -k http://www.class-example.com:8080
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://www.class-example.com/">here</a>.</p>
</body></html>
From ssh client
curl -k https://www.class-example.com:4443
```
WWW!!!!

Try also on your local browser accepting self signed certificate warning

tDYNAMIC CONTENT

Most modern websites do not consist of purely static content. Most content served out is actually generated dynamically, on demand. Integrating dynamic content with Apache **HTTPD** can be done in numerous ways. This section describes a number of the most common ways, but more ways exist.

SERVING DYNAMIC PHP CONTENT

A popular method of providing dynamic content is using the PHP scripting language. While PHP scripts can be served using old-fashioned CGI, both performance and security can be improved by having **httpd** run a PHP interpreter internally.
By installing the php package, a special **mod_php** module is added to **httpd**. The default configuration for this module adds the following lines to the main **httpd** configuration:

```
<FilesMatch \.php$>
  SetHandler application/x-httpd-php
<FilesMatch>
DirectoryIndex index.php
```

The **<FilesMatch>** block instructs **httpd** to use **mod_php** for any file with a name ending in **.php**, and the **DirectoryIndex** directive adds **index.php** to the list of files that will be sought when a directory is requested.

LAB - CONFIGURING A WEB APPLICATION

Create **/etc/httpd/conf.d/testphp.conf** with the following content:

```
<VirtualHost *:80>
ServerName www.php.class-example.com
DocumentRoot /srv/www/php
</VirtualHost>
```

Add a <Directory> block for **/srv/www/php** to **/etc/httpd/conf.d/testphp.conf** like the following:

```
<Directory /srv/www/php>
Require all granted
</Directory>
```

Add a new line in /etc/hosts

```
IPSERVER www.php.class-example.com
```

Install mod_php module

```
[root@server ~]# yum install -y php
```

Create the content directories

```
[root@server ~]$ mkdir -p /srv/www/php
```

Create **/srv/www/php/index.php** file with content

```
<?php echo 'Hello World'; ?>
```

Start and enable the httpd.service, and open the relevant firewall ports.

```
[root@server ~]$ systemctl start httpd.service
[root@server ~]$ systemctl enable httpd.service
```

Open both the http and https ports on the firewall.

```
[root@server ~]$ firewall-cmd --permanent --add-service=http
[root@server ~]$ firewall-cmd --reload
```

From you **HOST** machine

add a new line in /etc/host with content (linux,MacOs)

```
127.0.0.1 www.php.class-example.com
```

how to add local dns entry in windows https://gist.github.com/zenorocha/18b10a14b2deb214dc4ce43a2d2e2992

```
From ssh client
curl http://www.php.class-example.com:8080
Hello World
```

Try also on your local browser