UTILI

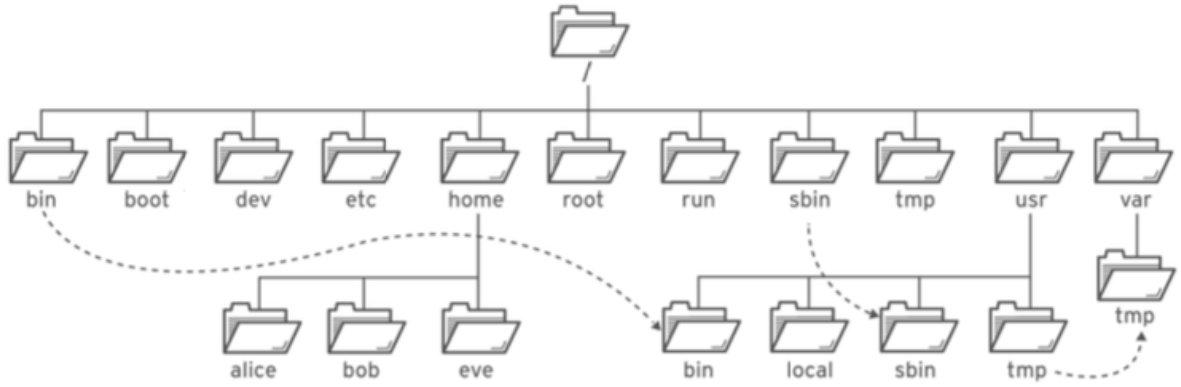
Pw: cl9op245Xdf#art
Ip adress: 10.0.2.15/24
Host name: class

Start: *ssh -l root 127.0.0.1 -p 2222*

- Cat [file1] [file2]: concatena più file
- Cd - :torna al path precedente
- Cd : torna al path user
- Useradd <nameuser>: aggiungi un nuovo utente al sistema
- Su – student: switch to student user from root

VIM COMMANDS:
- “:q!” per uscire dall’editor
- “:wq” esce e salva il file
- Premi “i” per inerire nuovo testo
- Premi “x per cancellare dei caratteri”
- Premi “a” per appendere del testo a una riga
- “dw” per eliminare una parola
- “d$” cancella tutto quello che c’è a destra del cursore fino a fine riga
- “dd” cancella l’intera riga
- “u” per annullare il comando precedente
- “U” per annullare le modifiche su una riga
-

BASICS CONCEPTS



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.

COMMAND-LINE FILE MANAGEMENT

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

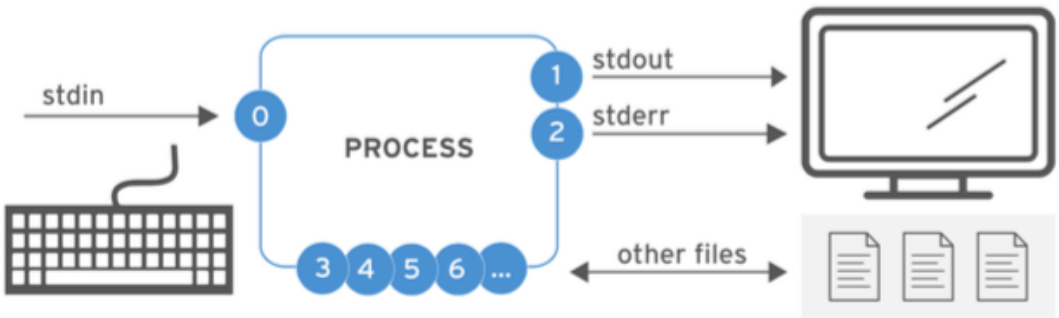
FILE GLOBBING: path name expansion

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.

STANDARD INPUT/OUTPUT/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.

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	

PIPELINES

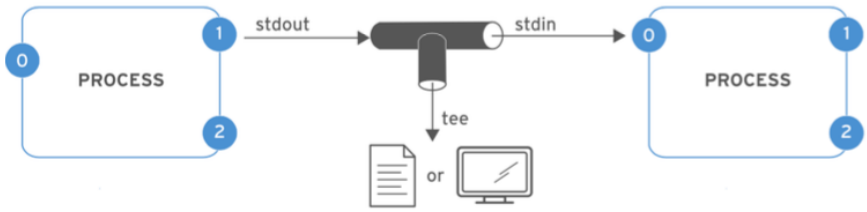
A pipeline is a sequence of one or more commands separated by |, the pipe character. A **pipe** connects the standard output of the first command to the standard input of the next command.

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.

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.

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
```

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.

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).

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.
Senza "-" la working directory rimane inalterata; tuttavia vengono modificate le informazioni di user e group: viene cambiato l'account

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.

MANAGING LOCAL USER ACCOUNTS

Some defaults, such as the range of valid **UID** numbers and default password aging rules, are read from the **/etc/login.defs** file

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

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.

MANAGING LOCAL GROUP ACCOUNTS

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

Salvati in /etc/group

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
```

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.**

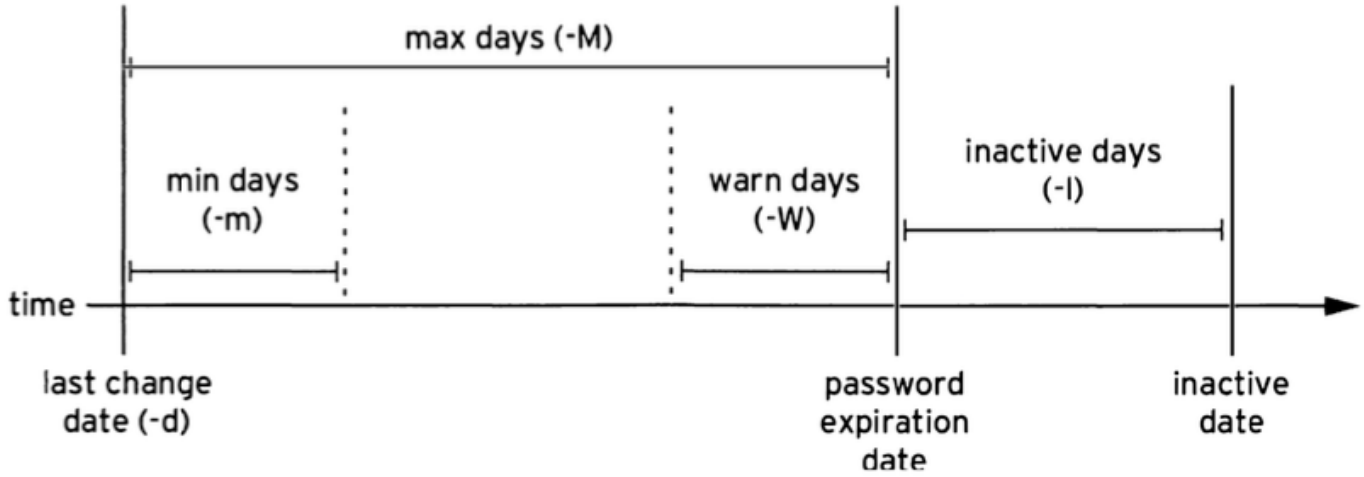
```
[root@desktop ~]# usermod -g ateam romeo
```

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

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

MANAGING USER PASSWORDS

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.

chage -E $(date -d "+365 days" '+%Y-%m-%d') gbianchi

LINUX FILE SYSTEM PERMISSIONS

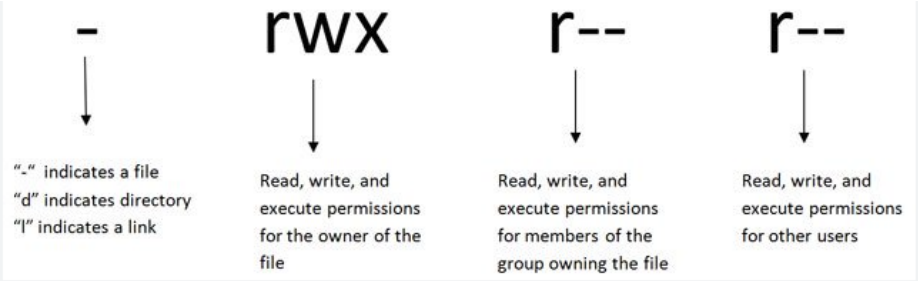
Access to files by users are controlled by file permissions.

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.

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:



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)

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.

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 **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.

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.

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

DEFAULT 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.
Per modificare la umask di un utente vai nel file .bashrc e aggiungi sotto a “#user specific aliases and functions”: umask 0022

MONITORING AND MANAGING LINUX PROCESSES

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

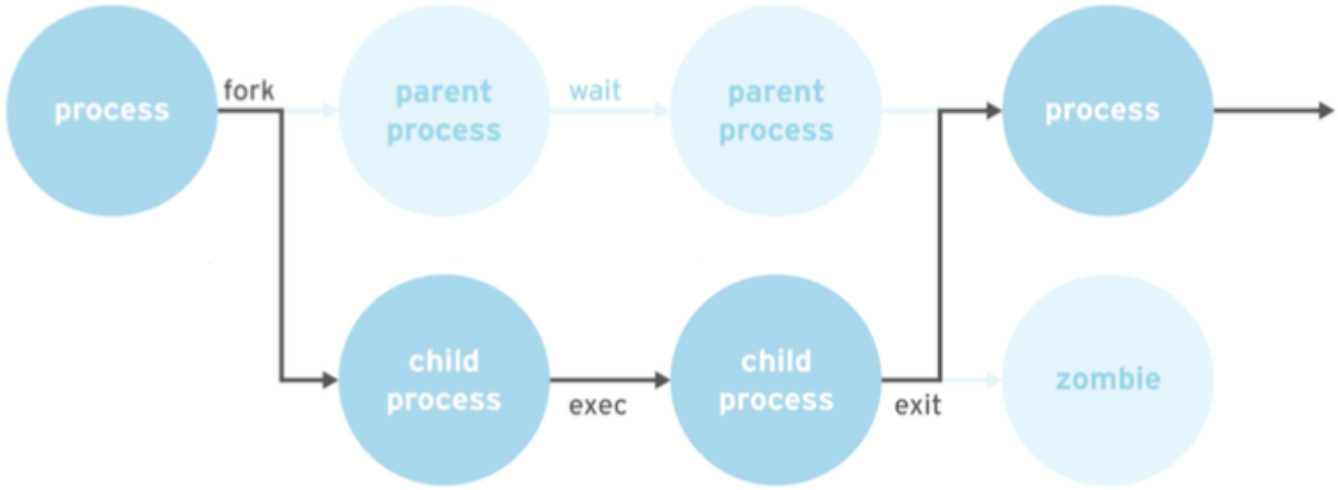
A process consists of:

- 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:

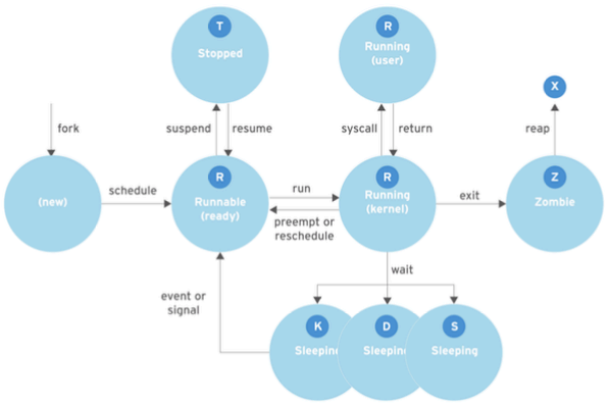
- local and global variables,
- a current scheduling context, and
- allocated system resources, such as file descriptors and network ports.

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.

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.

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

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
```

[1] is the job id

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
```

To send a foreground process to the background, first press the keyboard-generated suspend request (**Ctrl+z**) on the terminal. The job is immediately placed in the background and is suspended.

(Ctrl+c) to terminate a process.

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 &
```

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 Interrupt	Causes program termination. Can be blocked or handled. Sent 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.

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
```

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
```

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 -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
```

pkill per mandare segnali tra terminali diversi

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.
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.

- ***pgrep -l -u student***: restituisce per student i processi attivi e i loro process id
- use the command **jobs** to view all active processes in the terminal

MONITORING PROCESS ACTIVITY

- top, uptime, w

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.

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.

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.

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.
Several keyword 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.

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
   ```

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
   ```

STARTING AND STOPPING SYSTEM DAEMONS

Opzioni per **systemclt [option] [file.type]**:

- *stop, start, restart*: cambiano il PID
- *reload*: il servizio legge e ricarica i suoi file di configurazione, senza cambiare PID

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'
```

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**

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 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.

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
```

BASH SCRIPTING

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.

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.

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
```

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.

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.
```

BASH SHELL EXPANSHION FEATURES

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.

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.
```

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.

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.

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

DEBUGGING

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.

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
```

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.
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.

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.

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
```

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

```
[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.

Comparion 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:

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"]

Bash's string comparison:

OPERATOR	MEANING	EXAMPLE
=	is equal to	["$a" = "$b"]
==	is equal to	["$a" == "$b"]
!=	is not equal to	["$a" != "$b"]

Bash also has a few **unary operators** available for string evaluation. Unary operators evaluate just one item using the following format.

```
[ <UNARY OPERATOR> <ITEM> ]
```

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

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>]

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>]

USING CONDITIONAL STRUCTURES

The simplest of the conditional structures in Bash is the **if/then** construct, which has the following syntax.

```
if <CONDITION>; then
      <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
```

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
```

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.

```
case <VALUE> in
   <PATTERN1>)
       <STATEMENT>
       ...
       <STATEMENT>
       ;;
   <PATTERN2>)
    <STATEMENT>
       ...
       <STATEMENT>
       ;;
esac
```

```
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.

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 ~]$ export MYVAR
[student@desktop ~]$ echo $MYVAR
some value
[student@desktop ~]$ bash
[student@desktop ~]$ echo $MYVAR
some value
[student@desktop ~]$ exit
```

BASH START-UP SCRIPTS AND ALIAS

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.**

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.

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'
```

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>"

To make the alias persistent, the user would need to add the command to the bottom of their **~/.bashrc.** 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

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.

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.

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.

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"
```

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 is the default method in Fedora for managing host-level firewalls.
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:

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.

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.

MANAGING RICH RULES

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.

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.

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.

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**.

Some examples:

```
[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.
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.

PORT FORWARDING

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.

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.

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'
```

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.

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 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.

```
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 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.

```
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!
```

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.
GUARDA SLIDE 338

STARTING APACHE HTTPD

Begin by installing the httpd and httpd-manual packages.

```
[root@server ~]# sudo yum -y install httpd httpd-manual
```

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
```

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.

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.

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.

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.

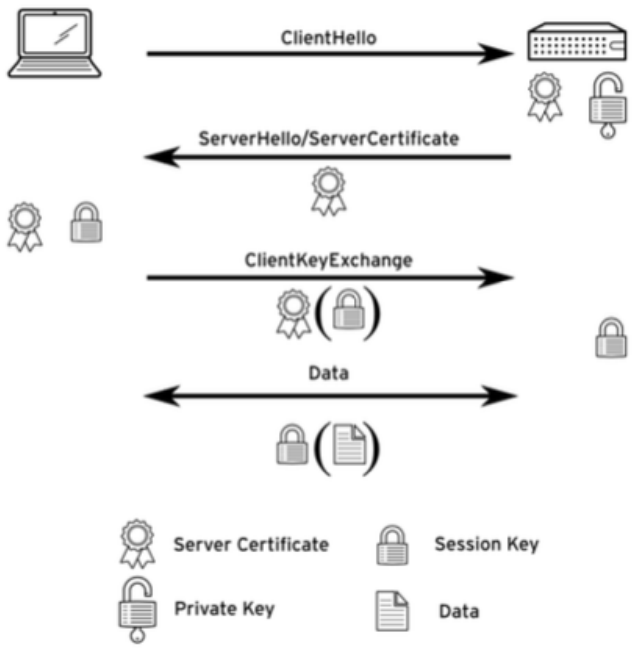
HTTPS: TRANSPORT LAYER SECURITY

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.



CONFIGURING TLF 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.

1. 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**.

2. 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.**

3. 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.
GUARDA SLIDE

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:

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.

DYNAMIC 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.

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.