



Red Hat System Administration I

Dr. Eng. Gamal Selim

Lecture 04

Agenda

- **Chapter 7: Controlling Access To Files With Linux File System Permissions**
- **Chapter 8: Monitoring And Managing Linux Processes**

Chapter 7: Controlling Access To Files With Linux File System Permissions

CONTROLLING ACCESS TO FILES WITH LINUX FILE SYSTEM PERMISSIONS

GOAL

To set Linux file system permissions on files and interpret the security effects of different permission settings.

OBJECTIVES

- Explain how the Linux file permissions model works.
- Change the permissions and ownership of files using command-line tools.
- Configure a directory in which newly created files are automatically writable by members of the group which owns the directory, using special permissions and default umask settings.

SECTIONS

- Linux File System Permissions (and Practice)
- Managing File System Permissions from the Command Line (and Practice)
- Managing Default Permissions and File Access (and Practice)

LINUX FILE SYSTEM PERMISSIONS

OBJECTIVES

After completing this section, students should be able to explain how the Linux file permissions model works.

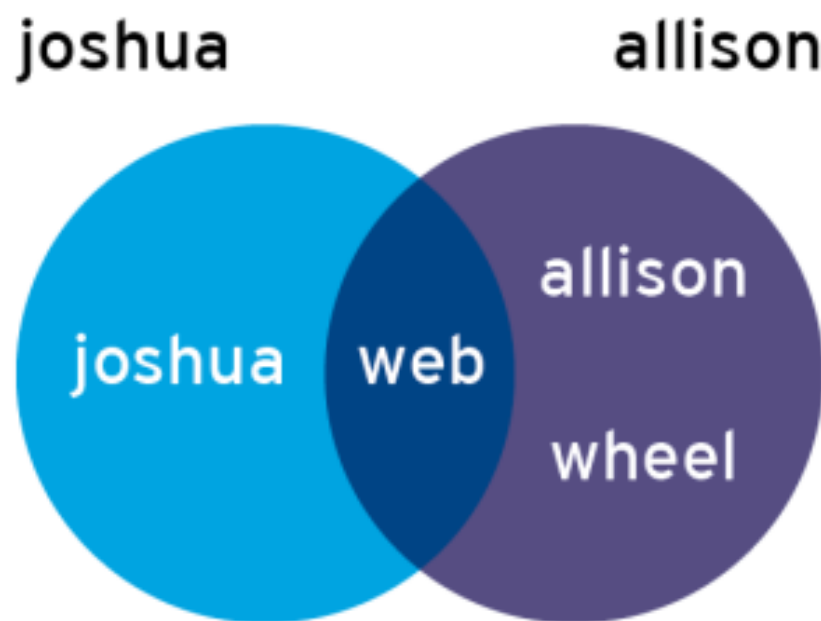
LINUX FILE SYSTEM PERMISSIONS

Access to files by users are controlled by *file permissions*. The Linux file permissions system is simple but flexible, which makes it easy to understand and apply, yet able to handle most normal permission cases easily.

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 graphic that follows, 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.



Effects of permissions on files and directories

PERMISSION	EFFECT ON FILES	EFFECT ON DIRECTORIES
r (read)	Contents of the file can be read.	Contents of the directory (file names) can be listed.
w (write)	Contents of the file can be changed.	Any file in the directory may be created or deleted.
x (exec)	Files can be executed as commands.	Contents of the directory can be accessed (dependent on the permissions of the files in the directory).

VIEWING FILE/DIRECTORY PERMISSIONS AND OWNERSHIP

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

```
[student@desktopX ~]$ ls -l test  
-rw-rw-r--. 1 student student 0 Feb  8 17:36 test
```

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

```
[student@desktopX ~]$ ls -ld /home  
drwxr-xr-x. 5 root root 4096 Jan 31 22:00 /home
```

EXAMPLES: LINUX USER, GROUP, OTHER CONCEPTS

Users and their groups:

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

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

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

ALLOWED/DENIED BEHAVIOR	CONTROLLING PERMISSIONS
lucy is the only person who can change the contents of lfile1.	lucy has write permissions on the file lfile1 as the owner. No one is listed as a member of the group lucy. The permissions for other do not include write permissions.
ricky can view the contents of lfile2, but cannot modify the contents of lfile2.	ricky is a member of the group ricardo, and that group has read-only permissions on lfile2. Even though other has write permissions, group permissions take precedence.
ricky can delete lfile1 and lfile2.	ricky has write permissions on the directory containing both files, and as such, he can delete any file in that directory.
ethel can change the contents of lfile2.	Since ethel is not lucy, and is not a member of the ricardo group, other permissions apply to her, and those include write permission.
lucy can change the contents of rfile1.	lucy is a member of the ricardo group, and that group has both read and write permissions on rfile1.
ricky can view and modify the contents of rfile2. lucy can view but not modify the contents of rfile2. ethel and fred do not have any access to the contents of rfile2.	ricky owns the file and has both read and write access to rfile2. lucy is a member of the ricardo group, and that group has read-only access to rfile2. other permissions apply to ethel and fred, and those permissions do not include read or write permission.

MANAGING FILE SYSTEM PERMISSIONS FROM THE COMMAND LINE

OBJECTIVES

After completing this section, students should be able to change the permissions and ownership of files using command-line tools.

CHANGING FILE/DIRECTORY PERMISSIONS

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

Symbolic method keywords:

```
chmod whoWhatWhich file|directory
```

- *Who is u, g, o, a (for user, group, other, all)*
- *What is +, -, = (for add, remove, set exactly)*
- *Which is r, w, x (for read, write, execute)*

The *symbolic* method of changing file permissions uses letters to represent the different groups of permissions: **u** for user, **g** for group, **o** for other, and **a** for all.

With the symbolic method, it is not necessary to set a complete new group of permissions. Instead, it is possible to change one or more of the existing permissions. In order to accomplish this, use three symbols: **+** to add permissions to a set, **-** to remove permissions from a set, and **=** to replace the entire set for a group of permissions.

The permissions themselves are represented by a single letter: **r** for read, **w** for write, and **x** for execute. When using **chmod** to change permissions with the symbolic method, using a capital **X** as the permission flag will add execute permission only if the file is a directory or already has execute set for user, group, or other.

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.

Examples

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

```
[student@desktopX ~]$  
chmod go-rw file1
```

- Add execute permission for everyone on **file2**:

```
[student@desktopX ~]$  
chmod a+x file2
```

- Set read, write, and execute permission for user, read, and execute for group, and no permission for other on **sampledir**:

```
[student@desktopX ~]$  
chmod 750 sampledir
```

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 Red Hat Enterprise 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 (change owner). For example, to grant ownership of the file **foofile** to **student**, the following command could be used:

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

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

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

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

```
[root@desktopX ~]#  
chown :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@desktopX ~]#  
chown visitor:guests 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.

MANAGING DEFAULT PERMISSIONS AND FILE ACCESS

OBJECTIVES

After completing this section, students should be able to configure a directory in which newly created files are automatically writable by members of the group which owns the directory, using special permissions and default umask settings.

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@desktopX ~]$ ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root root 35504 Jul 16  2010 /usr/bin/passwd
```

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@desktopX ~]$ ls -ld /tmp
drwxrwxrwt. 39 root root 4096 Feb  8 20:52 /tmp
```

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

Lastly, **setgid** on a directory means that files created in the directory will inherit the group affiliation from the directory, rather than inheriting it from the creating user. This is commonly used on group collaborative directories to automatically change a file from the default private group to the shared group.

In a long listing, you can spot the **setgid** permissions by a lowercase **s** where you would normally expect the **x** (group execute permissions) to be. If the group does not have execute permissions, this will be replaced by an uppercase **S**.

Effects of special permissions on files and directories

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

Setting special permissions

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

Examples

- Add the setgid bit on **directory**:

```
[root@desktopX ~]#  
chmod g+s directory
```

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

```
[root@desktopX ~]#  
chmod 2770 directory
```

DEFAULT FILE PERMISSIONS

The default permissions for files are set by the processes that create them. For example, text editors create files so they are readable and writeable, but not executable, by everyone. The same goes for shell redirection. Additionally, binary executables are created executable by the compilers that create them. The **mkdir** command creates new directories with all permissions set—read, write, and execute.

Experience shows that these permissions are not typically set when new files and directories are created. This is because some of the permissions are cleared by the umask of the shell process. The **umask** command without arguments will display the current value of the shell's umask:

```
[student@desktopX ~]$ umask  
0002
```

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,

In this example, please follow along with the next steps while your instructor demonstrates the effects of **umask** on new files and directories.

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

```
[student@desktopX ~]$ touch newfile1[student@desktopX ~]$ ls -l newfile1
-rw-rw-r--. 1 student student 0 May  9 01:54 newfile1
[student@desktopX ~]$ mkdir newdir1
[student@desktopX ~]$ ls -ld newdir1
drwxrwxr-x. 2 student student 0 May  9 01:54 newdir1
```

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

```
[student@desktopX ~]$ umask 0[student@desktopX ~]$ touch newfile2[student@desktopX ~]$ ls -l newfile2
-rw-rw-rw-. 1 student student 0 May  9 01:54 newfile2
[student@desktopX ~]$ mkdir newdir2
[student@desktopX ~]$ ls -ld newdir2
drwxrwxrwx. 2 student student 0 May  9 01:54 newdir2
```

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

```
[student@desktopX ~]$ umask 007[student@desktopX ~]$ touch  
newfile3[student@desktopX ~]$ ls -l newfile3  
-rw-rw----. 1 student student 0 May  9 01:55 newfile3  
[student@desktopX ~]$ mkdir newdir3  
[student@desktopX ~]$ ls -ld newdir3  
drwxrwx---. 2 student student 0 May  9 01:54 newdir3
```

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

```
[student@desktopX ~]$ umask 027[student@desktopX ~]$ touch  
newfile4[student@desktopX ~]$ ls -l newfile4  
-rw-r-----. 1 student student 0 May  9 01:55 newfile4  
[student@desktopX ~]$ mkdir newdir4  
[student@desktopX ~]$ ls -ld newdir4  
drwxr-x---. 2 student student 0 May  9 01:54 newdir4
```

5. Log in as **root** to change the default umask for unprivileged users to prohibit all access for users not in their group.

Modify **/etc/bashrc** and **/etc/profile** to change the default umask for Bash shell users. Since the default umask for unprivileged users is 0002, look for the **umask** command in these files that sets the umask to that value. Change them to set the umask to 007.

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

# Only display echos from profile.d scripts if we are no login shell
```

```
[root@desktopX ~]# vim /etc/bashrc
[root@desktopX ~]# less /etc/bashrc
# You could check uidgid reservation validity in
# /usr/share/doc/setup-*/uidgid file
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
    umask 007
else
    umask 022
fi

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

for i in /etc/profile.d/*.sh ; do
```

```
[root@desktopX ~]# vim /etc/profile
[root@desktopX ~]# less /etc/profile
# You could check uidgid reservation validity in
# /usr/share/doc/setup-*/uidgid file
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
    umask 007
else
    umask 022
fi

for i in /etc/profile.d/*.sh ; do
```

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

```
[student@desktopX ~]$ umask
0007
```

End of Chapter 7

Chapter 8: Monitoring And Managing Linux Processes

MONITORING AND MANAGING LINUX PROCESSES

GOAL

To evaluate and control processes running on a Red Hat Enterprise Linux system.

OBJECTIVES

- List and interpret basic information about processes running on the system.
- Control processes in the shell's session using bash job control.
- Terminate and control processes using signals.
- Monitor resource usage and system load due to process activity.

SECTIONS

- Processes (and Practice)
- Controlling Jobs (and Practice)
- Killing Processes (and Practice)
- Monitoring Process Activity (and Practice)

PROCESSES



OBJECTIVES

After completing this section, students should be able to:

- Define the life cycle of a process.
 - Define process states.
 - View and interpret process listings.
- 

WHAT IS A PROCESS?

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

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

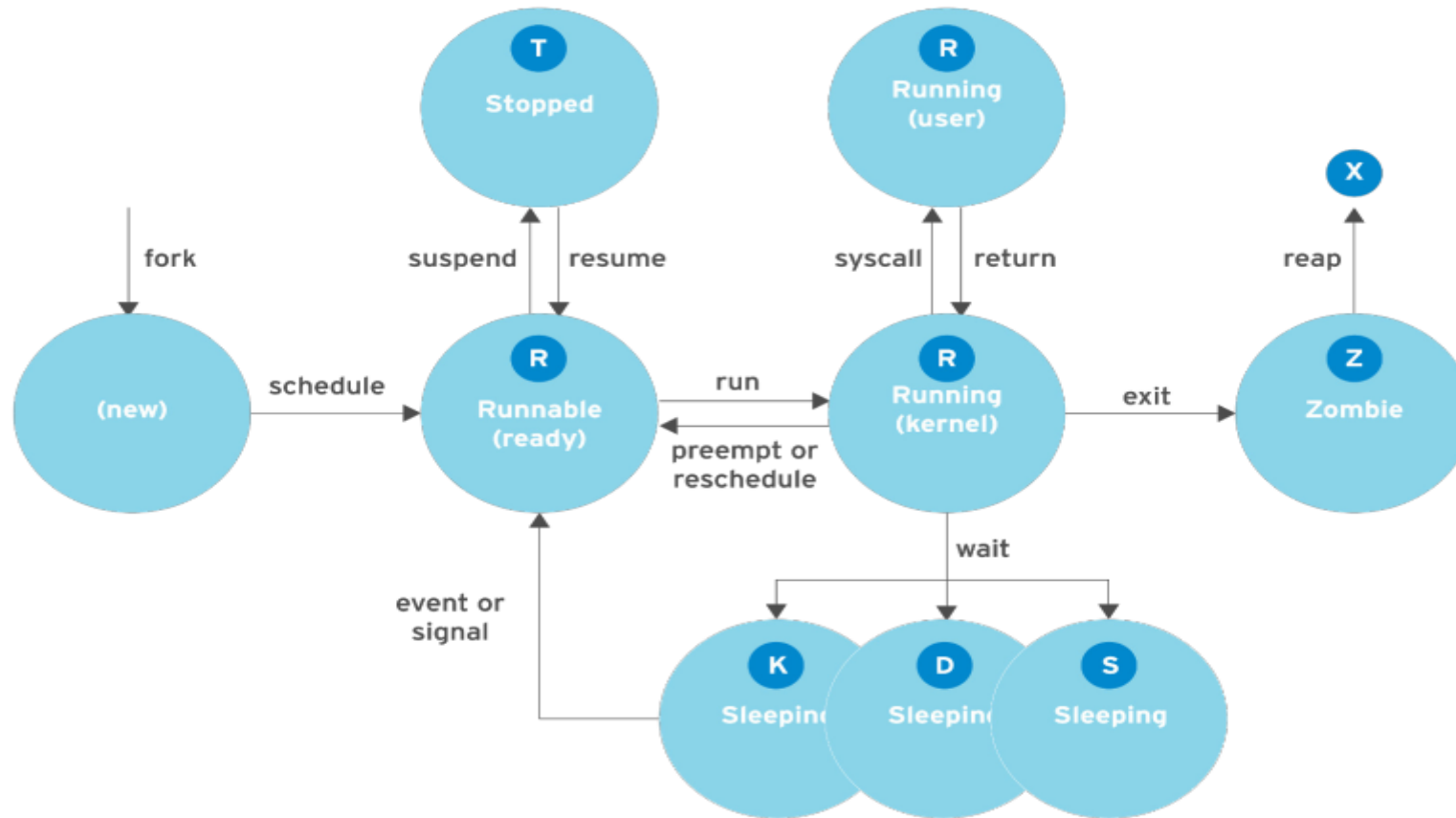
The *environment* of a process includes:

- 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)** on a Red Hat Enterprise Linux 7 system.

PROCESS STATES

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



Linux process states

NAME	FLAG	KERNEL-DEFINED STATE NAME AND DESCRIPTION
Running	R	TASK_RUNNING: The process is either executing on a CPU or waiting to run. Process can be executing user routines or kernel routines (system calls), or be queued and ready when in the <i>Running</i> (or <i>Runnable</i>) 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 <i>Running</i> .
	D	TASK_UNINTERRUPTIBLE: This process is also <i>Sleeping</i> , 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 <i>Killable</i> processes as D state.
Stopped	T	TASK_STOPPED: The process has been <i>Stopped</i> (suspended), usually by being signaled by a user or another process. The process can be continued (resumed) by another signal to return to <i>Running</i> .
	T	TASK_TRACED: A process that is being debugged is also temporarily <i>Stopped</i> 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 (<i>reaps</i>) the remaining child process structure, the process is now released completely. This state will never be observed in process-listing utilities.

LISTING PROCESSES

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 location of process **STDOUT**, known as the *controlling terminal*, and
- 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.

```
[student@serverX ~]$ ps aux
```

[illegible]

```
[student@serverX ~]$ ps lax
```

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
4	0	1	0	20	0	51648	7504	ep_pol	Ss	?	0:03	/usr/lib/ systemd/
1	0	2	0	20	0	0	0	kthrea	S	?	0:00	[kthreadd]
1	0	3	2	20	0	0	0	smpboo	S	?	0:00	[ksoftirqd/0]
1	0	5	2	0	-20	0	0	worker	S<	?	0:00	[kworker/0:0H]
1	0	7	2	-100	-	0	0	smpboo	S	?	0:00	[migration/0]

```
-- output truncated --
```

```
[student@serverX ~]$ ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	17:45	?	00:00:03	/usr/lib/systemd/systemd -- switched-ro
root	2	0	0	17:45	?	00:00:00	[kthreadd]
root	3	2	0	17:45	?	00:00:00	[ksoftirqd/0]
root	5	2	0	17:45	?	00:00:00	[kworker/0:0H]
root	7	2	0	17:45	?	00:00:00	[migration/0]

```
-- output truncated --
```

CONTROLLING JOBS

OBJECTIVES

After completing this section, students should be able to:

- Explain the terms foreground, background, and controlling terminal.
- Use job control to manage multiple command-line tasks.

JOBS AND SESSIONS

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

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

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 redisplay the shell prompt.

```
[student@serverX ~]$ sleep 10000 &  
[1] 5947  
[student@serverX ~]$
```

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

```
[student@serverX ~]$ jobs
[1]+  Running                  sleep 10000 &
[student@serverX ~]$
```

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

```
[student@serverX ~]$ fg %1
sleep 10000
—
```

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

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

```
sleep 10000
^Z
[1]+  Stopped                  sleep 10000
[student@serverX ~]$
```

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

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

```
sleep 10000
^Z
[1]+  Stopped                  sleep 10000
[student@serverX ~]$
```

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

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

```
[student@serverX ~]$ ps j
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
2764	2768	2768	2768	pts/0	6377	Ss	1000	0:00	/bin/bash
2768	5947	5947	2768	pts/0	6377	T	1000	0:00	sleep 10000
2768	6377	6377	2768	pts/0	6377	R+	1000	0:00	ps j

```
[student@serverX ~]$
```

KILLING PROCESSES

OBJECTIVES

After completing this section, students should be able to:

- Use commands to kill and communicate with processes.
- Define the characteristics of a daemon process.
- End user sessions and processes.

PROCESS CONTROL USING SIGNALS

A signal is a software interrupt delivered to a process. Signals report events to an executing program. Events that generate a signal can be an *error*, *external event* (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.

Fundamental process management signals

SIGNAL NUMBER	SHORT NAME	DEFINITION	PURPOSE
1	HUP	Hangup	Used to report termination of the controlling process of a terminal. Also used to request process reinitialization (configuration reload) without termination.
2	INT	Keyboard interrupt	Causes program termination. Can be blocked or handled. Sent by pressing INTR key 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+\).
9	KILL	Kill, unblockable	Causes abrupt program termination. Cannot be blocked, ignored, or handled; always fatal.
15 <i>default</i>	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.

Commands for sending signals by explicit request

Users signal their current foreground process by pressing a keyboard control sequence to suspend (**Ctrl+z**), kill (**Ctrl+c**), or core dump (**Ctrl+**) the process. To signal a background process or processes in a different session requires a signal-sending command.

Signals can be specified either by name (e.g., **-HUP** or **-SIGHUP**) or by number (e.g., **-1**). Users may kill their own processes, but root privilege is required to kill processes owned by others.

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

```
[student@serverX ~]$ kill PID[student@serverX ~]$ kill -signal PID[student@serverX ~]$ kill -l
```

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP

```
-- output truncated --
```

- 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@serverX ~]$  
killall command_pattern  
[student@serverX ~]$  
killall -signal command_pattern  
[root@serverX ~]#  
killall -signal -u username command_pattern
```

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@serverX ~]$
```

```
kill command_pattern
```

```
[student@serverX ~]$
```

```
kill -signal command_pattern
```

```
[root@serverX ~]#
```

```
kill -G GID command_pattern
```

```
[root@serverX ~]#
```

```
kill -P PPID command_pattern
```

```
[root@serverX ~]#
```

```
kill -t terminal_name -U UID command_pattern
```

LOGGING USERS OUT ADMINISTRATIVELY

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

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

```
[student@serverX ~]$ w -f
 12:43:06 up 27 min,  5 users,  load average: 0.03, 0.17, 0.66
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU WHAT
student   :0        :0               12:20    ?xdm?  1:10   0.16s gdm-session-wor
student   pts/0     :0               12:20    2.00s  0.08s  0.01s w -f
root      tty2             12:26    14:58   0.04s  0.04s -bash
bob       tty3             12:28    14:42   0.02s  0.02s -bash
student   pts/1     desktop2.example.12:41  1:07   0.03s  0.03s -bash
[student@serverX ~]$
```

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

```
[root@serverX ~]# pgrep -l -u bob
6964 bash
6998 sleep
6999 sleep
7000 sleep
[root@serverX ~]# kill -SIGKILL -u bob
[root@serverX ~]# pgrep -l -u bob
[root@serverX ~]#
```

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

```
[root@serverX ~]# pgrep -l -u bob
7391 bash
7426 sleep
7427 sleep
7428 sleep
[root@serverX ~]# w -h -u bob
bob      tty3      18:37    5:04    0.03s   0.03s  -bash
[root@serverX ~]# pkill -t tty3
[root@serverX ~]# pgrep -l -u bob
7391 bash
[root@serverX ~]# pkill -SIGKILL -t tty3
[root@serverX ~]# pgrep -l -u bob
[root@serverX ~]#
```

The same selective process termination can be applied using parent and child process relationships. Use the **ps tree** command to view a process tree for the system or a single user. Use the parent process's PID to kill all children they have created. This time, the parent **bash** login shell survives because the signal is directed only at its child processes.

```
[root@serverX ~]# pstree -p bob
bash(8391)─┬sleep(8425)
           ├──sleep(8426)
           └─sleep(8427)
[root@serverX ~]# pkill -P 8391
[root@serverX ~]# pgrep -l -u bob
```

```
bash(8391)
[root@serverX ~]# pkill -SIGKILL -P 8391
[root@serverX ~]# pgrep -l -u bob
bash(8391)
[root@serverX ~]#
```

MONITORING PROCESS ACTIVITY

OBJECTIVES

After completing this section, students should be able to:

- Interpret uptime and load averages.
- Monitor real-time processes.

LOAD AVERAGE

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

- *Active requests* are counted from per-CPU queues for running threads and threads waiting for I/O, as the kernel tracks process resource activity and corresponding process state changes.
- *Load number* is a calculation routine run every five seconds by default, which accumulates and averages the active requests into a single number for all CPUs.
- *Exponential moving average* is a mathematical formula to smooth out trending data highs and lows, increase current activity significance, and decrease aging data quality.
- *Load average* is the load number calculation routine result. Collectively, it refers to the three displayed values of system activity data averaged for the last 1, 5, and 15 minutes.

Understanding the Linux load average calculation

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

- Linux counts not only processes, but threads individually, as separate tasks. CPU request queues for running threads (*nr_running*) and threads waiting for I/O resources (*nr_iowait*) reasonably correspond to process states **R** (*Running*) and **D** (*Uninterruptable Sleeping*). Waiting for I/O includes tasks sleeping for expected disk and network responses.
- The load number is a global counter calculation, which is sum-totaled for all CPUs. Since tasks returning from sleep may reschedule to different CPUs, accurate per-CPU counts are difficult, but an accurate cumulative count is assured. Displayed load averages represent all CPUs.
- Linux counts each physical CPU core and microprocessor hyperthread as separate execution units, logically represented and referred to as individual CPUs. Each CPU has independent request queues. View **/proc/cpuinfo** for the kernel representation of system CPUs.

```
[student@serverX ~]$ grep "model name" /proc/cpuinfo
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
model name : Intel(R) Core(TM) i5 CPU          M 520  @ 2.40GHz
[student@serverX ~]$ grep "model name" /proc/cpuinfo | wc -l
4
```

Interpreting displayed load average values

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

- **top**, **uptime**, **w**, and **gnome-system-monitor** display load average values.

```
[student@serverX ~]$ uptime
15:29:03 up 14 min,  2 users,  load average: 2.92, 4.48, 5.20
```

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

```
# From /proc/cpuinfo, system has four logical CPUs, so divide by 4:
#                               load average: 2.92, 4.48, 5.20
#       divide by number of logical CPUs:    4      4      4
#                               ----  ----  ----
#                               per-CPU load average: 0.73  1.12  1.30
#
# This system's load average appears to be decreasing.
# With a load average of 2.92 on four CPUs, all CPUs were in use ~73% of the time.
# During the last 5 minutes, the system was overloaded by ~12%.
# During the last 15 minutes, the system was overloaded by ~30%.
```

REAL-TIME PROCESS MONITORING

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

Default output columns are recognizable from other resource tools:

- The process ID (**PID**).
- User name (**USER**) is the process owner.

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

End of Chapter 8

*Thank
you*

