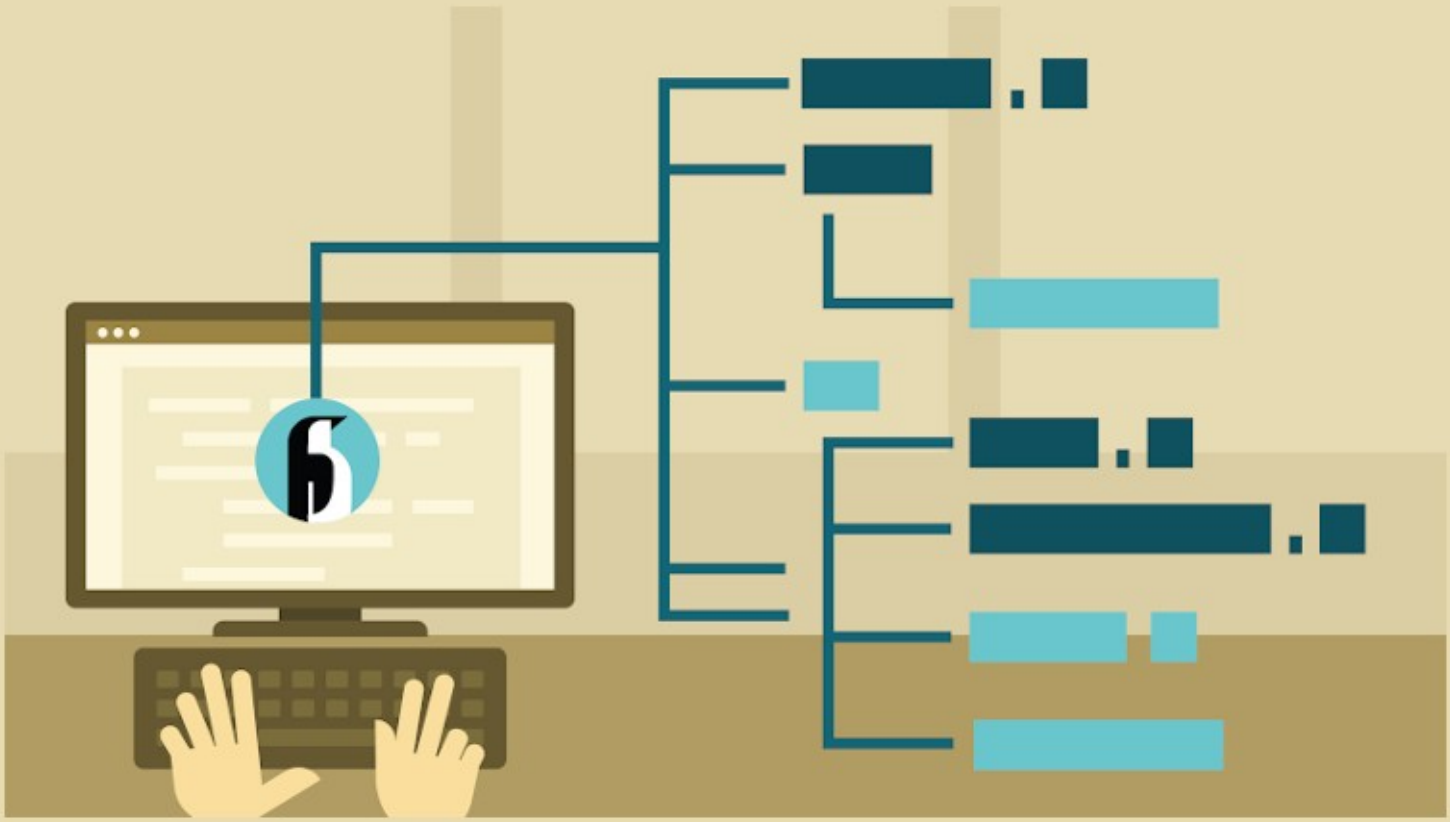
TECMINT
#1 LINUX BLOG

LEARN LINUX QUICKLY



Gabriel Canepa | Ravi Saive

Table of Contents

Copyright Notice

Important Notice

DISCLAIMER: We highly encourage you to become familiar with the commands used in this e-book and their man pages.

This e-book although it is as complete as possible - is intended as a starting point and not as an exhaustive guide system administration.

We hope you will enjoy reading this e-book as much as we enjoyed writing it and formatting it for distribution in PDF format.

You will probably think of other ideas that can enrich this e-book. If so, feel free to drop us a note at one of our social network profiles:

http://twitter.com/tecmint	https://www.facebook.com/tecmint
https://www.linkedin.com/company/tecmint	https://www.facebook.com/groups/linuxsysadmins/

In addition, if you find any typos or errors in this book, please let us know so that we can correct them and improve the material. Questions and other suggestions are appreciated as well – we look forward to hearing from you!

Chapter 1: Introduction to Linux

In this chapter, you will learn:

- What is Linux?
- Installing VirtualBox on Windows
- Importing Rocky Linux 9 virtual machine on VirtualBox
- VirtualBox extension pack and guest additions

What is Linux?

According to the Linux Foundation (an entity that is dedicated to fostering the growth of Linux):

> *Linux is, in simplest terms, an operating system. It is the software on a computer that enables applications and the computer operator to access the devices on the computer to perform desired functions. The operating system (OS) relays instructions from an application to, for instance, the computer's processor. The processor performs the instructed task, then sends the results back to the application via the operating system.*

Besides, Linux is both a *multi-user* (in that it allows multiple users on different computers or terminals to access a single system with one OS on it) and *multitasking* (available processor time is divided between several tasks automatically, creating the illusion, so to speak, that the tasks are running simultaneously).

A Little History

In 1991, in Helsinki, a computer science student by the name of Linus Torvalds began a project that later became the Linux kernel.

Development was done on Linus' computer, which was equipped with a 80386 processor, the MINIX operating system, and the GNU C compiler.

As he would write later in his book *Just for fun* [2001], Torvalds eventually realized that he had written an operating system kernel (a program to allocate resources and talk to the hardware, the 'heart' of the operating system, so to speak).

The first version of the Linux kernel was released under a personal license that restricted commercial activity. Later, in the middle of December 1992 he published version 0.99 using the GNU GPL.

The reason behind this change in this way of licensing had been explained earlier by Linus himself in the release notes of version 0.01:

> *Sadly, a kernel by itself gets you nowhere. To get a working system you need a shell, compilers, a library etc. These are separate parts and may be under a stricter (or even looser) copyright. Most of the tools used with linux are GNU software and are under the GNU copyleft. These tools aren't in the distribution - ask me (or GNU) for more info.*

That is why, from that moment on, Linus and GNU project developers worked to integrate GNU components (applications, libraries, and developer tools) with the Linux kernel (to make a fully functional and free operating system.

We must mention at this point that the GNU Project had been launched much earlier, (in 1984, to be accurate) by the Free Software Foundation (led by a former MIT AI scientist named Richard Stallman) to develop the GNU system.

Running Linux for the First Time

First off, let us clarify that when we use the word Linux nowadays, we usually refer to a Linux distribution, a fully-functional operating system that consists of the kernel and a series of application programs and libraries.

There are at least two ways to start experimenting with Linux. You can boot your computer using a live CD/DVD or USB stick, or installing a virtualization platform such as VirtualBox.

The first option (booting your computer using a live CD/DVD or USB stick) will not make any modifications to your storage devices.

Linux will run directly from the media device, and you will be able to test the operating system before actually installing it. If you decide you like it, you can later install it using the built-in wizard.

However, this book focuses on the second option: installing VirtualBox and setting up virtual machines. A virtual machine consists of an operating system (a Linux distribution, for example) running on top of the physical resources of your computer.

In the virtualization ecosystem, virtual machines are also known as *guests* or *appliances*, whereas the computer that runs the virtualization platform is known as the *host*.

To install VirtualBox, download it from here and then follow the instructions provided in Chapter 2 of the VirtualBox manual.

Importing a Virtual Machine

One of the advantages of using VirtualBox is that it allows us to export and import virtual machines previously created, and use them in a different host.

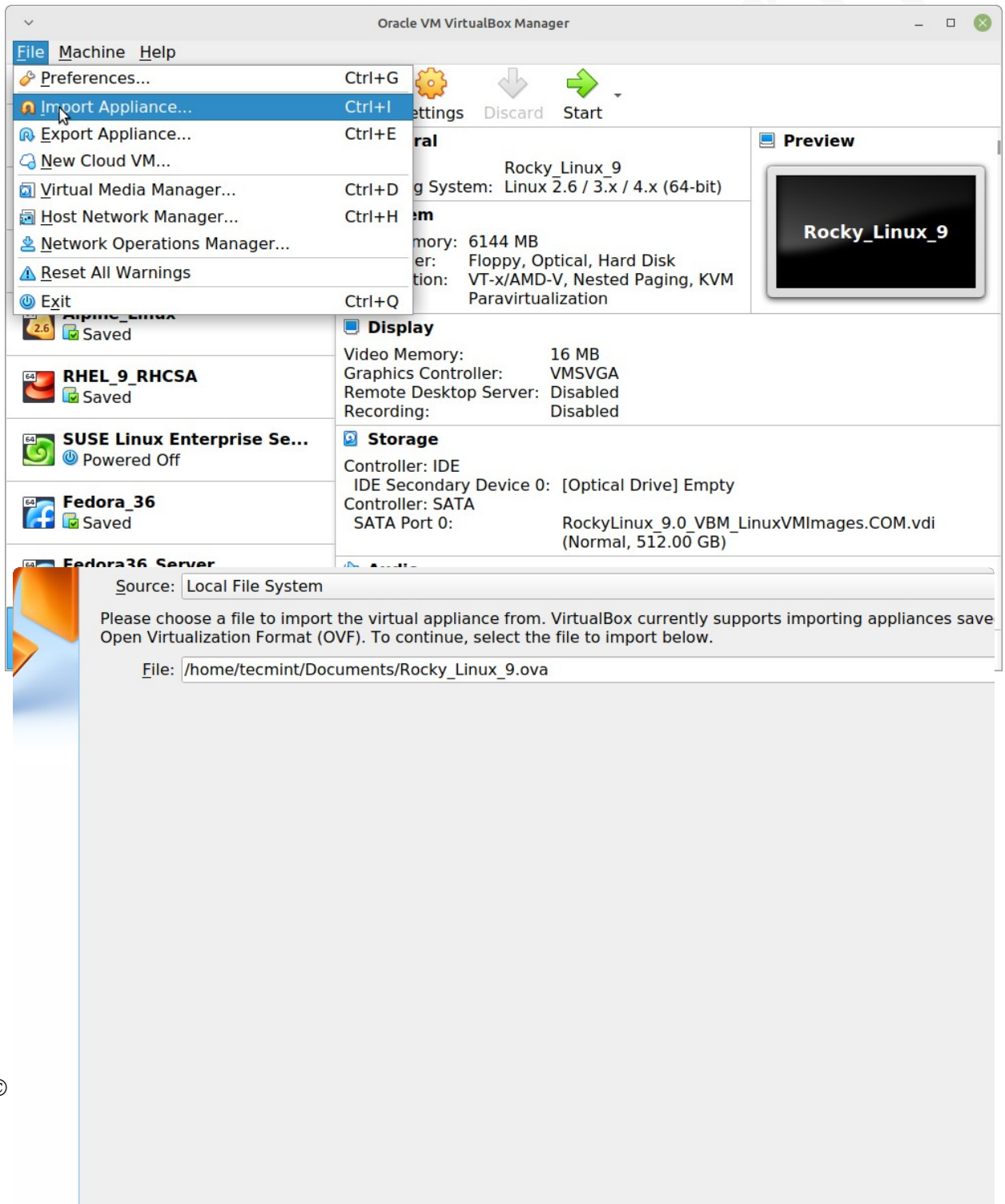
Thus, you will be able to start using Linux almost immediately by importing the virtual machines that are provided with this book.

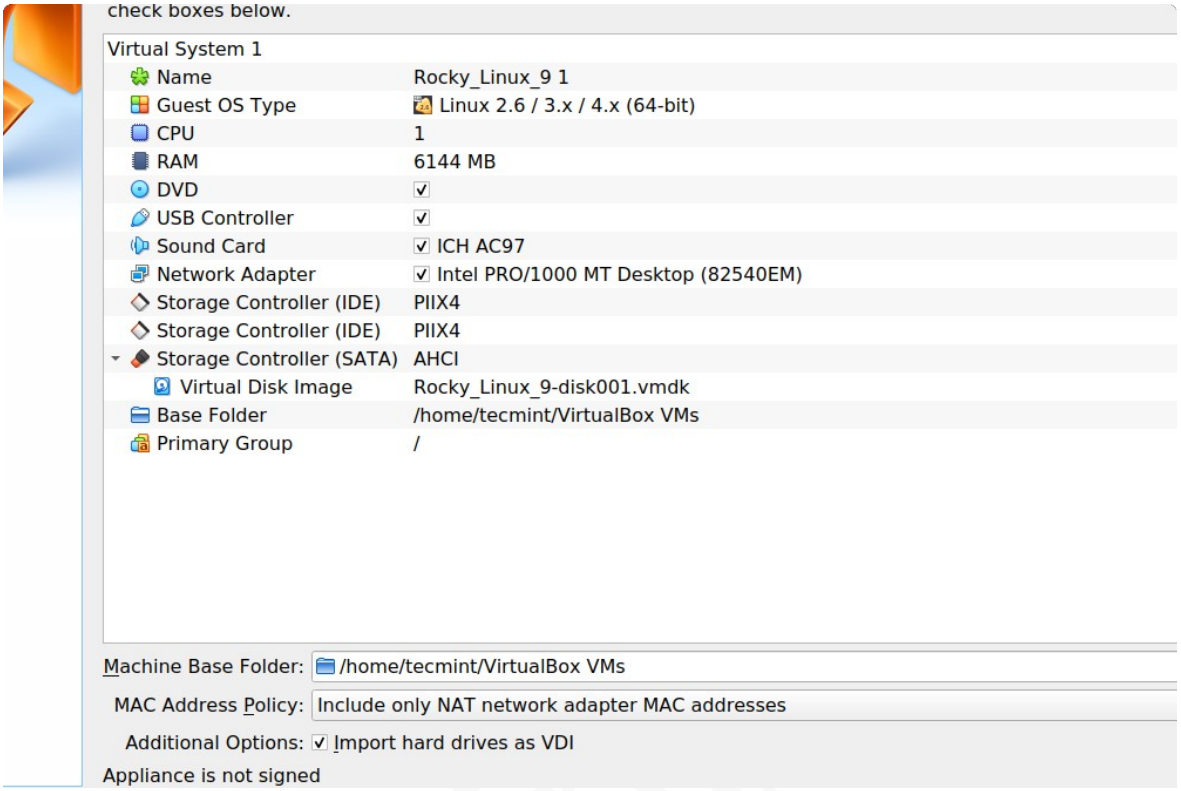
To download the virtual machine files, go to here. The **Rocky Linux 9 VM** is all in one file since it is relatively small (~830 MB).

Using VirtualBox, you can also add virtual disks as storage devices, increase the amount of memory allocated for the guest, and change the settings of virtual machines, among other things.

To import one of the virtual machines, follow these steps:

1. Go to File → Import appliance.
2. Select the .ova file.
3. Click Next and follow the instructions on the screen.
4. When the import process is completed, your virtual machine will show up in the main screen of VirtualBox. Right click on it and choose Start.





Guest Additions and the Extension Pack

As mentioned previously, VirtualBox makes it possible to run one or more guests on top of the host operating system.

To better leverage the hardware resources and to provide a better working experience with VirtualBox and the appliances running on top of it, the following software packages are provided separately:

Guest additions are drivers that can be installed on an desktop appliance-basis, and help improve the communication between the hosts and the guests.

They provide functionality such as shared clipboard / storage, and support for several screen resolutions.

To install guest additions, start a virtual machine [must have desktop environment installed] and then go to the Devices → Insert Guest Additions CD image menu, and follow the prompts.

To install desktop environment in Rocky Linux 9 minimal system run:

```
$ sudo dnf install epel-release
$ sudo dnf config-manager --set-enabled powertools
$ dnf copr enable stenstorp/lightdm
$ dnf groupinstall "xfce"
$ dnf install lightdm
$ systemctl disable gdm
$ systemctl enable lightdm
$ systemctl set-default graphical.target
$ reboot
```

The **Extension pack** provides integration between the host's USB 2.0 and 3.0 ports and the guest, encryption for disk images, and virtual remote desktop functionality.

To install them, go to the VirtualBox Downloads page, and click on Oracle VM VirtualBox Extension Pack.

Note that the installation file is operating system-agnostic, so it will work regardless of the operating system of the host where VirtualBox has been installed.

More information on how to install guest additions and the extension pack can be found in Chapter 1 of the VirtualBox manual.

Chapter 2: Files and Directories in Linux

In this chapter, you will learn:

- The Filesystem Hierarchy Standard
- What is the shell?
- Commands: pwd, cd, ls
- More commands: touch, echo, mkdir, rmdir, rm, cp, mv
- Redirection and pipelines
- History and tab completion in the command line
- Extra reading (links to related Tecmint articles)

The Filesystem Hierarchy Standard

A quick glance at distrowatch.com will give us but a glimpse of the vast number of Linux distributions available today. Some of them are maintained by well-known companies or free software initiatives, whereas others represent the contributions of individuals.

In order to provide a fundamental standard to be expected in modern Linux distributions, the Linux Foundation designed the Filesystem Hierarchy Standard (FHS) to be used by distribution maintainers, package developers, and system implementers.

The FHS consists of a document that provides an overall outline and reference of the Linux directory structure. Although it is not intended as a fixed set of instructions, it represents the specifications on what to reasonably expect in the directory tree of a given distribution.

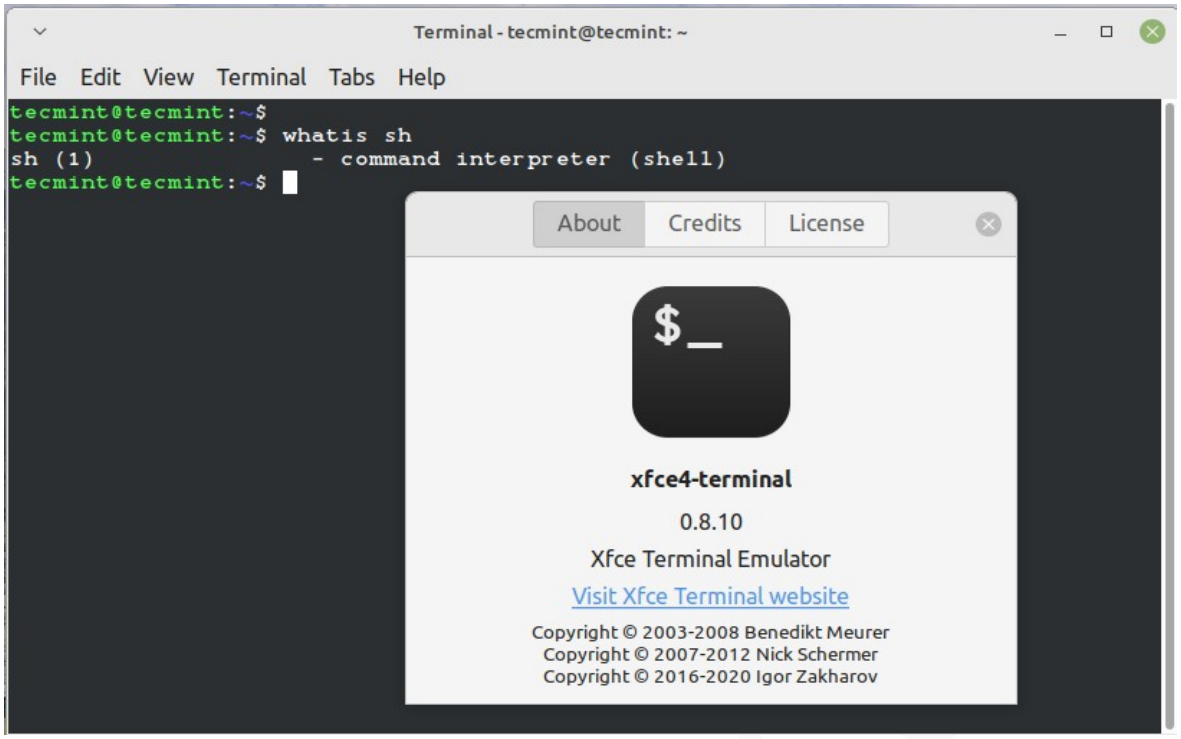
We will revisit this topic once we have learned the basic commands that are needed to navigate a Linux system.

What is the Shell?

A *shell i*s a program that accepts commands and gives them to the operating system to be executed.

In other words, the shell provides an interface layer between the Linux kernel and the end user.

Often used interchangeably, a *terminal* is a program that allows us to interact with the shell. For example:



When we first start a terminal, it presents a command prompt (also known as *the command line*), which tells us that the shell is ready to start accepting commands from the user.

Linux provides a range of options for shells, the following being the most common:

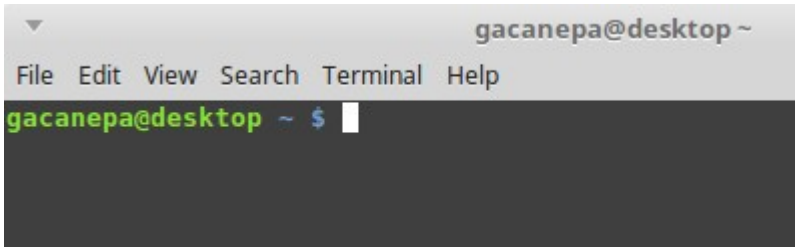
1) **bash**: Bash stands for **B**ourne **A**gain **SH**ell and is the default shell in most (if not all) modern Linux distributions. It incorporates tens of built-in commands, and an extensive documentation on how to use its wide variety of features. We will cover some of them in this book.

2) **sh**: The Bourne **SH**ell is the oldest shell and therefore has been the default shell of many UNIX-like operating systems for many years.

3) **ksh**: The **K**orn **SH**ell is a Unix shell which was developed by David Korn at Bell Labs in the early 1980s. It is backward-compatible with the Bourne shell and includes many features of the C shell.

To open a terminal, simply go to the applications menu and select *Terminal* (or a similar name) from the list.

Regardless of the distribution, you will be presented with a command prompt that should look as the below image:



where the *gacanepa* is the current username and *desktop* is the hostname. With that said, let's dive into the amazing world of the Linux command line!

Navigating the System

If we log into Linux using a text-mode login screen, chances are we will be dropped directly into our default shell.

On the other hand, if we log into Linux using a graphical user interface (GUI) login screen, we will have to open a shell manually by starting a terminal.

Either way, we will be presented with the user prompt and we can start typing and executing commands (a command is executed by pressing the Enter key after we have typed it).

Commands are composed of two parts:

1) the name of the command itself (along with one or more optional flags preceded by a hyphen), and

2) arguments.

To view the list of available options and / or required arguments, type **man** followed by the name of the command and press Enter.

A text document known as the man page of the command will open. To scroll down or up, use the arrows of your keyboard, and press the letter Q when you want to exit.

Let's consider the following examples to illustrate.

- To print the path of the current working directory, use **pwd**. This command does not take any optional flags nor requires an argument.
- To change to another location in the directory tree, use **cd** followed by a relative or absolute path. For example:
    - **cd Documents** will change the working directory to a directory named *Documents* inside the current one. In this case, a relative path is used since it is given starting at the current directory.

    - **cd /home/student/files** will change the working directory to */home/student/files*. This is an example of an absolute path since it is indicated starting at the root (/) directory.
    - In the above examples, both *Documents* and */home/student/files* are arguments passed to the **cd** command.
    - Additionally, **cd** accepts the following arguments:
        - **cd ~** or **cd** (without arguments) will change to the current user's *home* directory.
        - **cd ..** will change to the parent directory of the current one.
        - **cd ../../** will move two levels up the tree.

To view detailed information about files and directories, we will use **ls** followed by an optional flag (or more) and a file or directory name. When used without arguments, **ls** will return information about the objects inside the current working directory. These are the most used flags:

- **-l** returns a long directory listing. This includes not only the names of the objects (files and directories, for example) inside the current working directory, but also the access permissions (we will revisit this topic in the next chapter) and the owners of each object.
- **-a** causes **ls** to return hidden files, whose names start with a dot (.). Hidden files are often used as configuration files, and to store user preferences.
- **-t** tells **ls** to order the results by modification time.

For example,

- **ls -a** will display the names of all the objects inside the current working directory, hidden or not.
- **ls -la /var** will return a long directory listing of the contents of the */var* directory.
- **ls -lt ../** will show a long directory listing of the current working directory's parent.
- **ls -l /home/student/files/info.txt** will return a long directory listing of a file named *info.txt* located inside */home/student/files* if it exists, or an error message if it does not.

Exercises 1.1

In these exercises, you will write the commands to perform the following actions:

1.1.1: Change the current working directory to */etc*. Then move to */var/log* using a relative path.

1.1.2: Refer to the man page of **ls** to find out what option should be used to return a long directory listing of */var/log* (not its contents).

1.1.3: Display a long directory listing of */usr* ordered by modification time, newest first.

1.1.4: Navigate to the root directory using **cd /** and then list its contents with **ls**. Check the FHS documentation to learn more about each of the subdirectories found inside.

More commands to work with files and directories

The following commands are often used while working with files and directories in the command line:

- **touch** is used followed by a filename. If the file exists, **touch** will update its modification time (you can verify this with **ls -l** followed by the filename before and after doing **touch**). Otherwise, an empty file will be created with the name of the argument. For example, if *myfile.txt* does not exist inside the current directory, **touch myfile.txt** will create an empty file named *myfile.txt* in the said location.
- **rm** (short for remove) followed by a filename will remove the file in question. For example, rm myfile.txt will remove myfile.txt from the current working directory.
- **mkdir** followed by an argument will create a new directory. For example, **mkdir** me will create a directory named *me* inside the current directory. However, **mkdir me/otherdir** will fail if me does not already exist. To avoid this, use the **-p** option to create a directory structure that starts with a missing parent. In other words, **mkdir -p me/otherdir** will create a directory named *me* and a subdirectory called *otherdir* inside me in the same operation.
- **rmdir** is analogous to **rm**, but for directories. To remove a directory with this command, it must be empty.
- **cp**, besides the optional flags, requires two arguments. It is often used to copy the contents of one file (first argument) into another (second argument). For example, **cp file1.txt file2.txt** will copy the contents of *file1.txt* into *file2.txt*.
- **mv** operates almost identically than **cp**, only that it is used to move a file or directory (first argument) to a different location (second argument) when both arguments represent existing objects. If the second argument does not correspond to an existing object, **mv** will remove the first argument to the second one. For example, let us suppose that both *dir1* and *dir2* exist, whereas *dir3* does not. In that scenario, **mv dir1 dir2** will move *dir1* to *dir2*, thus becoming a subdirectory of the latter. On the other hand, **mv dir1 dir3** will rename *dir1* to *dir3*.

Note: You can use either relative or absolute paths with the above commands.

Exercises 1.2

1.2.1: The commands associated with removing or renaming files or directories should be used with caution. All of them provide an option that prompts the user for confirmation before actually performing the operation. Consult the man pages to find out what option should be used in this case.

1.2.2: Create an entire structure as files/personal/2017 inside the current directory using only one command.

1.2.3: Create an empty file inside files/personal/2017. Use **ls -l** to verify its last modification time and update it, then check again.

Redirection and Pipelines

To store the output of a given command in a file for later inspection, we will use the redirection operators. Depending on whether you want to overwrite the contents of a file or append to them, you will use > or >>, respectively. For example, **ls -l > longdirectorylisting.txt** will create a file named longdirectorylisting.txt inside the current directory (if it does not exist already) and write the output of **ls -l** to it. If the file exists and is not empty, use >> instead to avoid overwriting its contents, since > will replace them with the output of the command.

A classic use of redirection is writing a text string to a file using the **echo** command as follows:

```
echo "I love Tecmint.com" > testfile.txt
```

will write the message within quotes to a file named *testfile.txt* in the current directory.

On the other hand, we can chain several commands together using pipelines, represented by a vertical bar (|).

When two or more commands are thus chained together, the output of the first is sent as input to the second, and so on.

The advantage of using pipelines will become evident as we learn more and more commands. When we reach that point, we will recall what we have said in this chapter.

History and Tab Completion

After working with the command line for some time, you may want to repeat a command previously executed.

To facilitate this, the shell keeps a history that you can access using the **history** (surprise!) command.

When you type history in the command line and press Enter, the command history will be listed with line numbers. A command in the history list can be repeated by typing the command number preceded by an exclamation sign.

For example, the following excerpt shows some commands in the history list. In the following example, the output has been truncated for the sake of brevity and shows a portion of the command history:

```
...
253  cd /var/log
254  cd /home/student
255  ls -lt /home/student
...
```

If we want to execute cd /**home/student**, we can just type !254 and press Enter.

As we can see, this way of entering commands can come in handy when we want to repeat long entries in the history list.

Additionally, Bash provides a very useful feature known as tab completion. It allows you to autocomplete a command after typing a partial name and pressing the *Tab* key twice.

When multiple completions are possible, then all of them will be listed. For example, type *whi* and press Tab twice. Based on your system, you will get more or less results, but should be similar to:

```
which       which-nodejs  while      whiptail
```

On the other hand, if you do the same after typing *wha*, most likely you will only get **whatis** as result.

Exercises 1.3

1.3.1: Write a text string of your choice to a file named *mychoice.txt*. Use the right redirection operator to write another line without overwriting the existing contents of the said file. Then use the **cat** command, followed by the filename (**cat mychoice.txt**) to display the contents of the file.

1.3.2: Repeat the exercise of tab completion. Type *ch* and press *Tab* twice. Take note of 3 commands out of the resulting list and use the **whatis** command to find out what it is used for. Note you'll have to consult the man page of **whatis** first.

Further Reading

Links to articles published in Tecmint.com...

- http://www.tecmint.com/15-basic-ls-command-examples-in-linux/
- http://www.tecmint.com/ls-command-interview-questions/
- http://www.tecmint.com/echo-command-in-linux/

Chapter 3 - Permissions and Ownership

In this chapter, you will learn:

- Users and groups
- Important files: /etc/passwd, /etc/group, /etc/shadow
- Commands: chmod, chown, chgrp, visudo
- The /etc/sudoers file
- Extra reading (links to related Tecmint articles)
- Exercises

In Linux, users and groups are used to control access to files, directories, and other system resources.

As a system administrator you will need to know how to add, edit, suspend or delete user accounts and groups, and granting them the necessary permissions to perform their tasks.

These tasks must be performed as root, the superuser, or using the sudo command as a regular user. To learn how to use sudo, refer to the links in the *Further reading* section at the end of this chapter.

To switch to the root account, type

```
su -
```

and type root's password.

Adding and Modifying User Accounts and Groups

To add a new user called *me*, do:

```
useradd me
```

This will also create a group named *me*.

Additionally, you can create other groups using groupadd. For example, the following command will add a group called *support*:

```
groupadd support
```

When a new account is added, you can find the information about it in */etc/passwd*. This file contains a record per system user account and has the following format (fields are delimited by a colon):

- User name
- User password (or the character x if the password is stored in */etc/shadow* in encrypted form)
- User ID (UID): an integer that identifies the account.
- Group ID (GID): another integer that identifies the group to which the user belongs.
- User info: this file is optional. If it is not empty, it will contain extra information about each user account.
- Absolute path to the user's home directory.
- Absolute path to the default shell for the user.

At any time after adding an account, you can edit the following information (and others as well) using **usermod**, whose basic syntax is as follows:

```
usermod [options] [username]
```

Examples:

- To set an expiration date, use the **--expiredate** flag followed by a date in the YYYY-MM-DD format.
- To add the user to supplementary groups, use the combined **-aG**, or the separate **--append** and **--groups** options, followed by a comma separated list of groups.

- To change the location of the user's home directory, use the **-d**, or **--home** options, followed by the absolute path to the new home directory.
- To change the user's default shell, use the **--shell** switch, followed by the path to the new shell. If you don't want to allow the user to be able to login (perhaps to temporarily suspend access to his / her account), you can use */usr/sbin/nologin* or */usr/bin/false* as shell.

Note that you can edit the user's information in one single command or separately. You can perform one or more of the above operations simultaneously. The following example illustrates the process of doing it all at once:

```
usermod --expiredate 2017-05-31 --append --groups users --home /tmp --
shell /bin/sh me
```

Let's examine what the above command does:

- First off, the account that is being modified appears at the end of the command (*me*).
- It will expire on May 31, 2017.
- The account will be added the the users group.
- Its home directory is changed to */tmp*, and its default shell to */bin/sh*.

From time to time, you may need to change other users' passwords or even your own. To do so, use the **passwd** command followed by the username for which you want to change the password. In case you want to reset it for yourself, just type **passwd** and press Enter.

Exercises 2.1

2.1.1: Create a new user account named *johndoe* and a group called *misc*.

2.1.2: Add *johndoe* to group *misc*.

2.1.3: Change the default shell to /bin/sh.

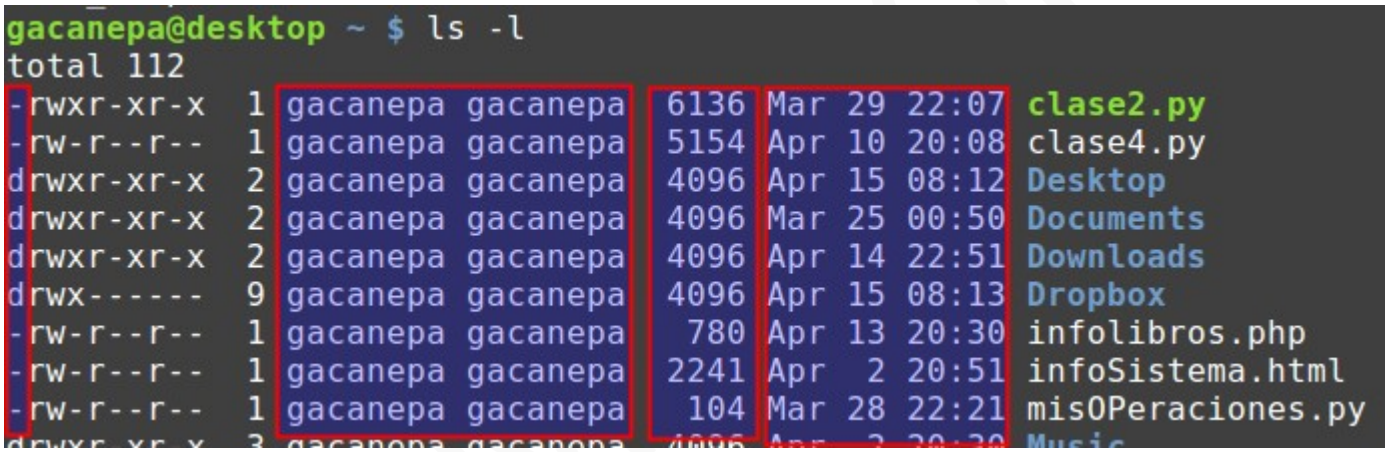
2.1.4: Reset the password of johndoe to *this#is$my%newpassword*.

File Permissions and Ownership

In **Chapter 2**, you learned to use ls to list files and directories. Let's take a look at the output of ls -l, which provides a detailed listing of a directory's contents or a file's properties.

The first character indicates the object type (first highlighted column in the below image):

- – represents a regular file.
- d indicates the object is a directory.
- l represents a symbolic link (a link, or shortcut, to another object).
- b indicates the object is a block device. Storage devices fall into this category.



The next nine characters of the file attributes represent the **file mode**. The first 3 characters in this group indicate whether the permissions of the file's owner on a given object.

The next 2 sets of 3 characters indicate the same information for the file's group owner, and the rest of the system users. For example, let's examine infoSistema.html in the above image:

- It is a regular file.
- The file's owner (*user: gacanepa*) has read and write permissions on the file.
- Other members of the file's group owner (*group: gacanepa*) will only be able to read the file. Same as other system users.

The easiest way to change a file's mode is using the chmod command followed by an expression that indicates the owner's rights with the letter u, the group owner's rights

with the letter g, and the rest with o. Permissions are then granted (or revoked) with the + or – signs, respectively.

For example,

- **chmod o-r infoSistema.html** revokes the read permission over the greeting file for users that are not members of the group owner.
- **chmod g+wx infoSistema.html** grants write and execute permissions over the file for members of the group owner.
- **chmod +x infoSistema.html** grants execution permissions for all users.

File and group ownership are changed with the **chown** and **chgrp** commands, respectively. The basic syntax for both commands is as follows:

1. **chown root infoSistema.html** will change the owner of the file to *root*.
2. **chgrp me infoSistema.html** will change the group owner of the file to *me*.

After the owner and group associated with the file, **ls -l** shows the size of files, the last modification date, and the file name.

Exercises 2.2

2.2.1: Create a new empty file named file1.sh and set user johndoe as its new owner.

2.2.2: Change the group owner of file1.sh to misc.

2.2.3: Grant execute permissions on file1.sh to all system users.

Removing Users and Groups

From time to time, you may also need to remove users or groups for good. Note that this is different than merely disabling them as we learned previously.

To remove an account, user **userdel** followed by the username.

Similarly, user **groupdel**, followed by the group name, to remove a group.

Exercises 2.3

2.3.1: Remove the user account johndoe and the group misc. Do a **ls -l** on *file1.sh*. What do you see in the columns that represent the owner and group owner of *file1.sh*?

Further Reading

Links to articles published in Tecmint.com...

http://www.tecmint.com/vi-editor-usage/

http://www.tecmint.com/su-vs-sudo-and-how-to-configure-sudo-in-linux/

http://www.tecmint.com/rhcsa-exam-manage-users-and-groups/2/

Chapter 4 - Finding and Describing Files

In this chapter, you will learn:

- Find files based on one or more search criterias
- Describing files
- Commands: find, type, file
- Extra reading (links to related Tecmint articles)
- Exercises

Finding Files in the System

The **find** command is used to search recursively through directory trees for files or directories that match certain characteristics. Subsequently, it can either print the matching files or directories or perform other operations on the matches. We can search for files by name, owner, group, type, permissions, date, and other criteria.

Searching for Files by Name

To search for a file by name, we will use the **-name** option. We can use wildcards if we enclose pattern in quotes.

As a result, **find** will locate files that match the wildcard filename. The following example will locate all files with a name ending in .sh inside */home/gacanepa*:

```
find /home/gacanepa -name "*.sh"
```


Searching by Permissions

If we need to find files that have certain permissions, we can do so by using the **-perm** option. If we precede mode with a + (plus sign), find locates files in which any of the specified permission bits are set.

If we precede mode with a - (minus sign), find locates files in which all the specified permission bits are set. To find all files that can be executed by any user, located inside the current directory (represented by a dot) and descend recursively down to 3 levels (using **-maxdepth**):

```
find . -maxdepth 3 -type f -perm -o=x
```

Searching by Size

We can search for a file of a given size with the **-size n** expression. Normally, n is specified in 512-byte blocks, but we can modify this by trailing the value with a letter code, such as c for bytes, k for kilobytes (units of 1024 bytes), M for megabytes (units of 1048576 bytes), or G for Gigabytes (units of 1073741824 bytes). To find all files that are larger than 100MB inside the current directory, we will do

```
find . -type f -size +100M
```

where **-type f** indicates we're searching for regular files. If we were searching for directories, we would use **-type d** instead.

Searching by Owner or Group Owner

The **-gid GID** or **-uid UID** expression searches for files whose group ID (GID) or user ID (UID) is set to GID or UID, respectively.

The -group option locates files whose group name is name, but the -gid option can come in handy if the GID has been orphaned and has no name, but the latter is generally easier to use. Same applies to **-user**.

Examples:

Find all directories inside */var* owned by group *lp*:

```
find /var -type d -group lp
```

Find all files owned by *root* in */etc*:

```
find /etc -type f -user root
```

Searching by Access or Modification Time

To avoid cluttering up your system with old files that you may never use again, it is a good idea to do a clean up once in a while. Modification and access times represent the date when a file was last modified or accessed.

- **-atime +30 or -mtime +30** means files that were last accessed or modified more than 30 days ago, respectively.
- **-atime -30 or -mtime -30** less than 30 days.
- **-atime 30 or -mtime 30** exactly 30 days.

To view the list of files that were last accessed more than 6 months ago (180 days) inside */home/me/Documents*, do

```
find /home/me/Documents -type f -atime +180
```

Performing Operations on the Search Results

Finally, we can do something with the results returned by find using the **-exec** option. This flag takes a command (along with its options) as an argument, followed by **{}** + which is a placeholder for the matches resulting from the search.

To update the timestamps of all .sh files inside the current directory:

```
find . -name "*.sh" -exec touch {} +
```

To remove the execute permission from all .sh files inside /opt, do

```
find /opt -type f -name "*.sh" -exec touch {} +
```

As you can see, you can combine several search criterias in the same command.

Determining a File's Type

As we explore the system, it will be useful to know what kind of contents a file has without opening it. T

o do this, we can use the **file** command to determine a file's type (unlike other operating systems, file extensions in Linux are not required to reflect a file's contents).

We invoke the file command followed by a filename. Similarly, the type command is used to determine the type of an executable file.

For example,

```
file RMD.py
type ps
type touch
type echo
```

```
gacanepa@desktop ~ $ file RMD.py
RMD.py: Python script, UTF-8 Unicode text executable
gacanepa@desktop ~ $ type ps
ps is /bin/ps
gacanepa@desktop ~ $ type touch
touch is /usr/bin/touch
gacanepa@desktop ~ $ type echo
echo is a shell builtin
gacanepa@desktop ~ $
```

Particularly, shell builtins are a special type of executables that are provided by the shell itself and may differ from one to another.

Exercises 3.1

3.1.1: Find all directories owned by user root in */var*. Only descend down to two levels.

3.1.2: Find all files that have been last modified more than 3 months ago in your home folder.

3.1.3: Find all empty files in the current directory using the **-empty** option of **find**.

3.1.4: Find out the type of the following files using **type**: **echo**, **which**, **find**.

Further Reading

Links to articles published in Tecmint.com...

http://www.tecmint.com/35-practical-examples-of-linux-find-command/

http://www.tecmint.com/find-linux-command-description-and-location/

http://www.tecmint.com/explanation-of-everything-is-a-file-and-types-of-files-in-linux/

Chapter 5 – Linux Processes

In this chapter, you will learn

- Definition of a process
- Daemons
- Signals
- Commands: ps, top, nice, renice, kill, killall
- Extra reading (links to related Tecmint articles)
- Exercises

Every program that runs on our Linux system is a process. The normal life cycle of a process includes starting, executing, and dying, and is automatically managed by the kernel, without need for user intervention.

However, once in a while one (or more) of the following exceptions may occur:

- The process dies due to a known or an unknown reason and needs to be restarted.
- The process "runs wild" and consumes system resources at an abnormal level and needs to be terminated.
- The process needs to be restarted in order for it to reread its configuration file after it has been modified.

Knowing how to handle those situations is at the very center of the Linux administration skills that every system administrator needs to posses.

To begin, let's introduce the following concepts related with processes:

- Every process has a number assigned to it when it starts. It is called **Process ID (PID)** and is an integer unique among all running processes.
- Processes must have associated privileges, and a process' **UID (User ID)** and **GID (Group ID)** are associated with the user who started it, and only have access to the system resources owned by it and also others, depending on the file permissions.
- The first process started by the kernel at boot time is a program called **systemd**. This process has PID 1 and is the **parent process** of all other processes on the system. The parent process ID (PPID) is the PID of the process that created the process in question.

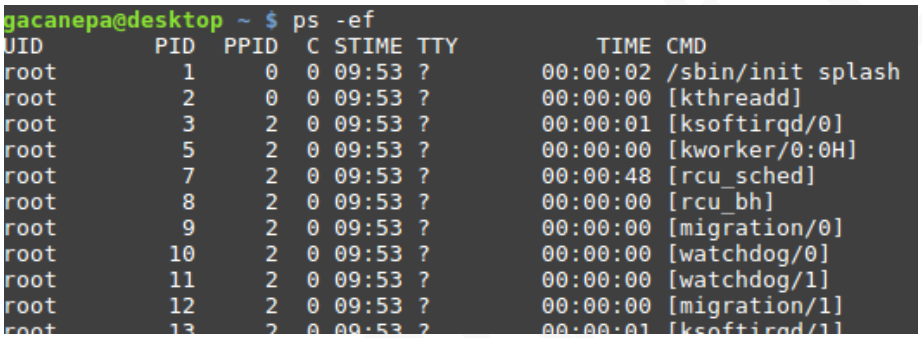
At any time, there could be tens or even hundreds of processes running on our Linux system. Monitoring these processes is done using three convenient tools: **ps**, **pstree**, and **top**.

Reporting Current Processes with ps

The **ps** command allows us to take a snapshot of processes currently running on our system. Depending on the options used, it can return different types of information as we will see in the examples.

Without options, **ps** will list the processes owned by the current user. On the other hand, it is important to note that you can combine several options into one. For example, instead of typing **ps -e -f** you can do **ps -ef**.

To display the full list of processes using the standard format, do **ps -ef** as shown in the following image where the output has been truncated for the sake of brevity:



In the above image, the *TIME* column indicates the accumulated CPU time used by the process. You'll learn the meaning of the other columns in the *Exercises* section in this chapter.

It is important to note that you can filter the output of **ps -ef** using a pipeline followed by the **grep** command and a filter pattern. For example, to display all the processes with the word *firefox* in the associated command, do:

```
ps -ef | grep -i firefox | grep -v grep
```

where the **-i** option will cause **grep** to ignore the case and treat **Firefox**, **firefox**, or **FiReFoX** indistinctly. Also, the second pipeline followed by **grep -v grep** will remove the actual **grep** command from the results.

Another well-known method to print a list of processes is using an user-defined format. This is explained in greater detail in the *STANDARD FORMAT SPECIFIERS* section in the man page. To illustrate, let's suppose we only want to display the PID, PPID, the command, and the memory usage. To do this, use the combined **-eo** option followed by **pid,ppid,cmd,%mem** as follows:

```
ps -eo pid,ppid,cmd,%mem
```

If you want to sort the results returned by the above command, ps also provides a --sort option that must be followed by an equal sign and the desired sort field. For example, the following example returns the same information as the previous one, only that it's ordered by memory usage in descending form (that's what -%**mem** is for):

```
ps -eo pid,ppid,cmd,%mem --sort=-%mem
```

To sort in ascending form, omit the minus sign as follows:

```
ps -eo pid,ppid,cmd,%mem --sort=-%mem
```

Displaying a Tree of Processes with pstree

To view a hierarchical list of processes in a tree format, you can use pstree. This tool is very handy for understanding parent / child process relationships.

If the PID is specified after the -p option, the displayed tree is rooted at that process. Otherwise, it is rooted at the process with PID 1. If user (a valid username) is specified at the end, trees for all processes owned by user are shown.

The following image shows processes starting at PID 1954:

```
pstree -p 1954
```

and a portion of the list of processes owned by gacanepa:

```
pstree gacanepa -p
```

```
gacanepa@desktop ~ $ pstree -p 1954
pulseaudio(1954)─┬─{alsa-sink-VT170}(2172)
                 ├─{alsa-source-USB}(2168)
                 └─{alsa-source-VT1}(2173)
gacanepa@desktop ~ $ pstree gacanepa -p
at-spi-bus-laun(1848)─┬─dbus-daemon(1856)
                      ├─{dconf worker}(1849)
                      ├─{gdbus}(1852)
                      └─{gmain}(1850)

at-spi2-registr(1864)─┬─{gdbus}(1868)
                      └─{gmain}(1867)

clock-applet(2067)─┬─{dconf worker}(2108)
                   └─{gdbus}(2101)
```

Exercises 4.1

4.1.1: Use **man ps** to identify the meaning of the columns *USER*, *%CPU*, *%MEM*, and *STAT* in the output of **ps aux**, which is another standard variant to display all processes.

4.1.2: Use ps and the required options to return only the PID and CMD columns of the process list.

4.1.3: Use pstree to display all processes owned by root. Make sure to display the PIDs as well.

Monitoring Linux Processes with top

Similar to **ps**, **top** lists the process currently running on our system - but updates by default every *5* seconds. This is useful when we want to watch the status of one or more processes or to see how they are using our system.

This listing is ordered by default by CPU usage. Additionally, a header of useful information (current time, uptime, number of users and processes, load, CPU status, and memory, to name a few examples) is also displayed, as seen in the below image:

```
top - 13:44:28 up  3:51,  1 user,  load average: 0,51, 0,56, 0,54
Tasks: 196 total,   7 running, 189 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4,4 us,  1,4 sy,  0,0 ni, 93,4 id,  0,9 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem :  7916896 total,  4494424 free,  2118908 used,  1303564 buff/cache
KiB Swap:  8126460 total,  8126460 free,        0 used.  5434676 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1498 root      20   0  320224  64156  50516 S   4,0  0,8   8:29.96 Xorg
 2505 gacanepa  20   0 3363436 1,287g 118700 S   4,0 17,0  46:43.83 Web Content
 7324 gacanepa  20   0  863048 134628  37312 R   2,0  1,7   0:07.41 shutter
 2065 gacanepa  20   0  519900  30536  21112 R   1,7  0,4   0:02.33 wnck-applet
 2367 gacanepa  20   0 2361376 452172 125896 R   1,7  5,7  15:08.33 firefox
```

If you want to use a different refresh rate, use the the interval (-d) option. For example, **top -d 3** will bring up **top** and refresh its output every 3 seconds.

Killing Processes

in more precise terms, killing a process means terminating it by sending it a signal to either finish its execution gracefully (SIGTERM=15) or immediately (SIGKILL=9) through the **kill** or **pkill** commands. The difference between these two tools is that the former is used to terminate a specific process or a process group altogether, while the latter allows you to do the same based on the process name and other attributes.

In addition, **pkill** comes bundled with **pgrep**, which shows you the PIDs that will be affected should pkill be used. For example, before running

```
pkill -u gacanepa
```

it may be useful to view at a glance which are the PIDs owned by gacanepa:

```
pgrep -l -u gacanepa
```

```
[root@rhel7 ~]# pgrep -l -u gacanepa
2333 sshd
2334 bash
10053 bash
10079 top
[root@rhel7 ~]#
```

By default, both **kill** and **pkill** send the SIGTERM signal to the process. As we mentioned above, this signal can be ignored (while the process finishes its execution or for good), so when you seriously need to stop a running process with a valid reason, you will need to specify the *SIGKILL* signal on the command line:

To force the termination of a process identified by PID=1954, do

```
kill -9 1954
```

or

```
kill -s SIGKILL 1954
```

Additionally, you can also terminate processes based on the owner or other attributes, such as all the children of a given process. For example,

```
pgrep -l -P 9892
```

will list the processes that would be kill if we run

```
pkill -P 9892
```

Exercises 4.2

4.2.1: Use **kill -l** and **man 7 signal** to a) list the available signals, and b) learn about what SIGTERM, SIGSTOP, and SIGKILL.

4.2.2: Use **pgrep** to identify a list of processes that would be killed if you run **pkill** to terminate a) all processes owned by a given user, and b) all the child processes of a given one.

Modifying Process Execution Priorities

Part of Linux's flexibility is to let users and system administrators prioritize process execution. This feature can come in handy when we have a high-load machine and want to make sure some special process(es) get more rights (or "priority") to use system resources than others.

However, we must note that under normal circumstances we don't need to worry about execution priority because the kernel handles it automatically. The usual method of performing a change of execution priority is through the **nice** and **renice** commands. We can use nice to launch a program with a specified priority or use renice to alter the priority of a running one.

Using nice

When you start a process using nice, the kernel will allocate more or less system resources to the process based on the assigned priority (a number commonly known as "niceness" in a range from -20 to 19).

The lower the value, the greater the execution priority. Regular users (other than root) can only modify the niceness of processes they own to a higher value (meaning a

lower execution priority), whereas root can modify this value for any process, and may increase or decrease it.

The default priority value is 0. To start a script named *backup.sh* with a different priority, use nice -n followed by the desired priority. For example:

```
nice -n -10 backup.sh
```

or

```
nice -n 5 backup.sh
```

While the first example will increase the priority to -10, the second one will decrease it to 5.

Using renice

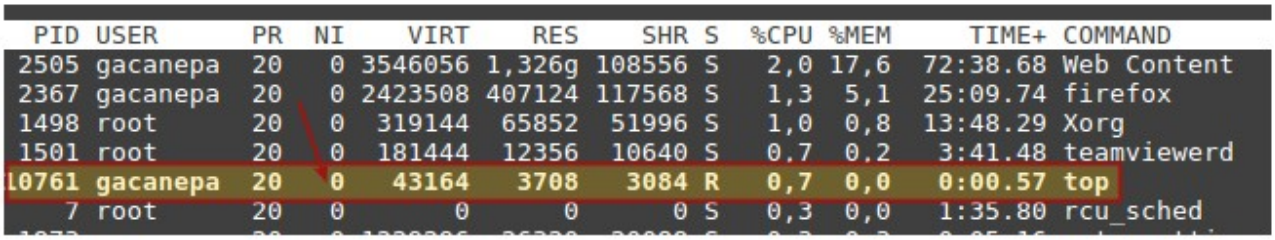
If a process is already running, you can modify its execution priority using **renice -n** followed by the new priority and -p to indicate the PID. For example, let's suppose that backup.sh is still running with PID 9892 and you want to change its priority to 10. To do this, use the following command:

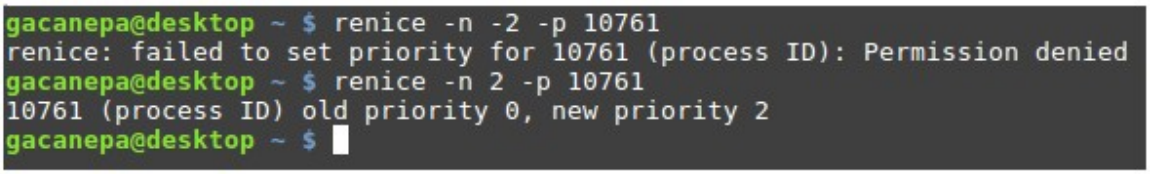
```
renice -n 10 -p 9892
```

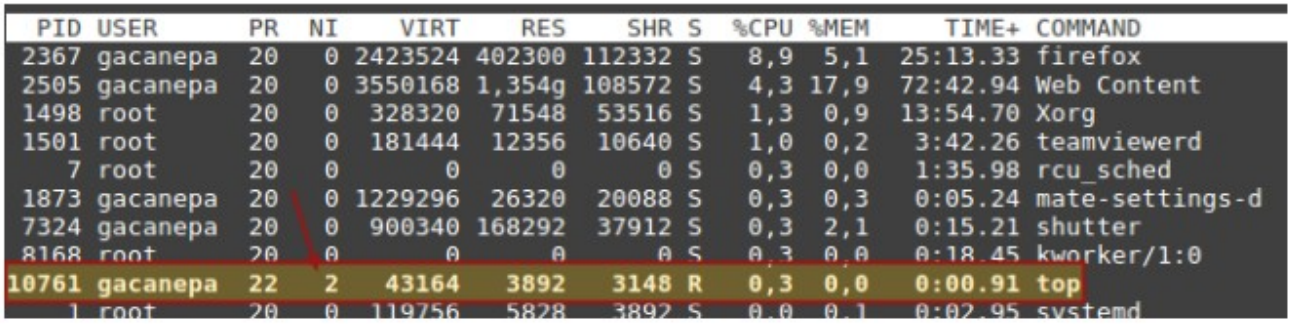
Exercises 4.3

4.3.1: Try to change the niceness of a process not owned by you. Do you get any error message(s)? If so, what does it say?

4.3.2: Launch **top** and identify the column that lists the niceness of processes. From another terminal, change the priority of this process and verify that the niceness change is reflected in the output of the first terminal.







Chapter 6 – Linux Shell Scripts

In this chapter, you will learn:

- Shell scripts with Bash
- Environment Variables
- Variable substitution
- Shell expansion
- Extra reading (links to related Tecmint articles)
- Exercises

In simple words, a shell script is nothing less and nothing more than a plain text file. Thus, it can be created and edited using our preferred text editor. We will use vim in the following examples, but you may choose whatever one you like.

Type

```
vim myscript.sh
```

and press Enter.

The very first line of a shell script must be as follows (also known as a *shebang*):

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

It "tells" the operating system the name of the interpreter that should be used to run the text that follows.

Now it's time to add our commands. We can clarify the purpose of each command, or the entire script, by adding comments as well. Note that the shell ignores those lines beginning with a pound sign # (explanatory comments).

```
#!/bin/bash
DATE=$(date +%Y-%m-%d)
echo "I am learning shell scripting"
echo "Today is $DATE"
```

In line #2, we create a variable named **DATE** with the contents of the command enclosed within **$()**. This syntax tells Bash to run whatever command we specify between **$()** and assign the output to the variable.

Once the script has been written and saved, we need to make it executable:

```
chmod +x myscript.sh
```

Before running our script, we need to say a few words about the $PATH environment variable, a special kind of variable that is available from the moment you login using a console or a terminal. If we run

```
echo $PATH
```

from the command line, we will see the contents of $PATH: a colon-separated list of directories that are searched when we enter the name of a executable program. It is called an environment variable because it is part of the shell environment - a set of information that becomes available for the shell and its child processes when the shell is first started.

When we type a command and press Enter, the shell searches in all the directories listed in the $PATH variables and executes the first instance that is found. Let's see an example:

```
[gacanepa@dev1 ~]$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/gacanepa/.local/bin:/home/gacanepa/bin
[gacanepa@dev1 ~]$
```


If there are two executable files with the same name, one in /usr/local/bin and another in /usr/bin, the one in the first directory will be executed first, whereas the other will be disregarded.

If we haven't saved our script inside one of the directories listed in the $PATH variable, we need to prepend ./ to the file name in order to execute it:

```
./myscript.sh
```

Otherwise, we can run it just as we would do with a regular command:

```
myscript.sh
```

Flow Control

Whenever you need to specify different courses of action to be taken in a shell script, as result of the success or failure of a command, you will use the if construct to define such conditions. Its basic syntax is:

```
if CONDITION; then
      COMMANDS
else
      OTHER-COMMANDS
fi
```

where CONDITION can be one of the following (only the most frequent conditions are cited here) and evaluates to true when:

- **[-a file]** → file exists.
- **[-d file]** → file exists and is a directory.
- **[-f file]** →file exists and is a regular file.
- **[-u file]** →file exists and its SUID (set user ID) bit is set.
- **[-g file]** →file exists and its SGID bit is set.
- **[-k file]** →file exists and its sticky bit is set.
- **[-r file]** →file exists and is readable.
- **[-s file]**→ file exists and is not empty.
- **[-w file]**→file exists and is writable.
- **[-x file]** is true if file exists and is executable.
- **[string1** = **string2]** → the strings are equal.
- **[string1** != **string2]** →the strings are not equal.
- **[int1** ***op*** **int2]** where op is one of the following comparison operators:
    - **-eq** --> is true if int1 is equal to int2.
    - **-ne** --> true if int1 is not equal to int2.
    - **-lt** --> true if int1 is less than int2.

- **-le** --> true if int1 is less than or equal to int2.
- **-gt** --> true if int1 is greater than int2.
- **-ge** --> true if int1 is greater than or equal to int2.

Loops

This loop allows to execute one or more commands for each value in a list of values. Its basic syntax is:

```
for item in SEQUENCE; do
      COMMANDS;
done
```

where *item* is a generic variable that represents each value in *SEQUENCE* during each iteration.

While Loops

This loop allows to execute a series of repetitive commands as long as the control command executes with an exit status equal to zero (successfully). Its basic syntax is:

```
while EVALUATION; do
      EXECUTE_COMMANDS
done
```

where *EVALUATION_COMMAND* can be any condition that can evaluate to true or false, and *EXECUTE_COMMANDS* can be any program, script or shell construct, including other nested loops.

Putting it all Together

We will demonstrate the use of the **if** construct and the **for** loop with the following example.

Determining if a service is running in a systemd-based distro

Let's create a file with a list of services that we want to monitor at a glance:


```
[gacanepa@dev1 ~]$ cat myservices.txt
sshd
mariadb
httpd
crond
firewalld
[gacanepa@dev1 ~]$
```

Our shell script (*for_demo.sh*) should look like:

```
#!/bin/bash

# This script iterates over a list of services and
# is used to determine whether they are running or not.
LIST=$(cat myservices.txt)

for service in $LIST; do
      systemctl --quiet is-active $service
      if [ $? -eq 0 ]; then
            echo "$service is [ACTIVE]"
      else
            echo "$service is [INACTIVE, NOT INSTALLED, OR UNKNOWN]"
      fi
done
```

Let's explain how the script works:

1) The *LIST* variable is populated with the output of

```
cat myservices.txt
```

2) During each iteration, service represents the value of an element inside *LIST*. For each element of *LIST*, the following command will be executed:

```
systemctl status $service | grep --quiet "running"
```

This time we need to precede our generic variable (which represents each element in LIST) with a dollar sign to indicate it's a variable and thus its value in each iteration should be used.

The output is then piped to grep. When that happens, the above command returns an exit status of 0 (represented by **$?** in the if construct), thus verifying that the service is running. An exit status different than 0 indicates that the service is not running.


```
[gacanepa@dev1 ~]$ ./for_demo.sh
sshd is [ACTIVE]
mariadb is [INACTIVE or NOT INSTALLED]
httpd is [INACTIVE or NOT INSTALLED]
crond is [ACTIVE]
firewalld is [ACTIVE]
[gacanepa@dev1 ~]$
```

We could go one step further and check for the existence of myservices.txt before even attempting to enter the for loop

```
#!/bin/bash

# This script iterates over a list of services and
# is used to determine whether they are running or not.
LIST=$(cat myservices.txt)

if [ -f myservices.txt ]; then
      for service in $LIST; do
            systemctl status $service | grep --quiet "running"
            if [ $? -eq 0 ]; then
                  echo $service "is [ACTIVE]"
            else
                  echo $service "is [INACTIVE or NOT INSTALLED]"
            fi
      done
else
      echo "myservices.txt is missing"
fi
```

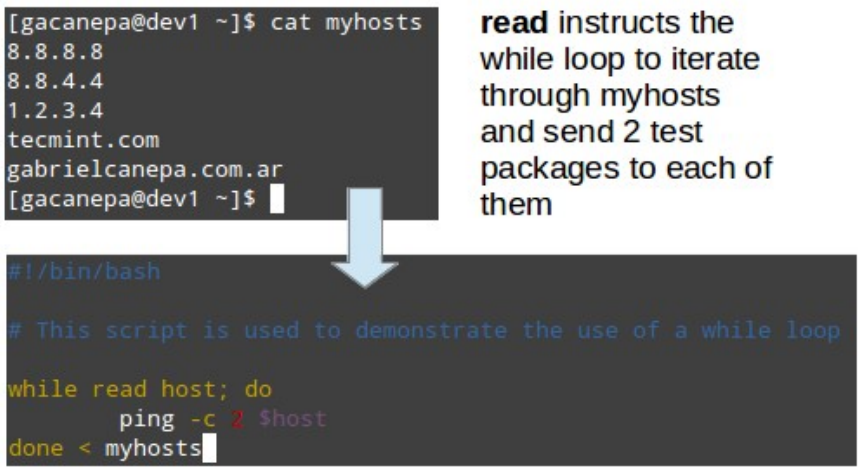
Pinging a series of network or internet hosts for reply statistics

You may want to maintain a list of hosts in a text file and use a script to determine every now and then whether they're pingable or not (feel free to replace the contents of myhosts and try for yourself). The read shell built-in command tells the while loop to read myhosts line by line and assigns the content of each line to variable host, which is then passed to the ping command.

```
#!/bin/bash

# This script is used to demonstrate the use of a while loop

while read host; do
      ping -c 2 $host
done < myhosts
```

Exercises 5.1

5.1.1: Write a shell script that saves the list of running processes to a file named *process_list.txt* in the current directory. Use the custom format to display only the PID, PPID, CMD, %CPU, and %MEM, sorted by %CPU in descending form.

5.1.2: Write a shell script that finds all files with permissions 777 (read, write, and execute permissions allowed for all users in the system) and changes them to 644 (read and write permissions for the owner, and only read privileges for the rest). Hint: use the -perm option as explained in Chapter 4.

Further Reading

http://www.tecmint.com/category/bash-shell/

Chapter 7 - Maintaining Software using APT and YUM

In this chapter, you will learn:

- Use **aptitude** or **yum** to search for, install, update, or remove packages.
- Extra reading (links to related Tecmint articles)
- Exercises

When we speak of package management in Linux, we refer to a method of *installing* and *maintaining* (which includes *updating* and probably *removing* as well) software on the system.

In the early days of Linux, programs were only distributed as source code, along with the required man pages, the necessary configuration files, and pretty much nothing more. Nowadays, most Linux distributors use by default pre-built programs or sets of programs called **packages**, which are presented to users ready for installation on that distribution.

How Package Management Systems Work

If a certain package requires a certain resource -such as a shared library, or another package-, it is said to have a *dependency*. All modern package management systems provide some method of dependency resolution to ensure that when a package is installed, all of its dependencies are installed as well.

Almost all the software that is installed on a modern a Linux system will be found on the Internet. It can either be provided by the distribution vendor through central repositories (which can contain several thousands of packages, each of which has been specifically built, tested, and maintained for the distribution) or be available in source code that can be installed downloaded and installed manually.

Because different distribution families use different packaging systems (Debian: *.deb / CentOS: *.rpm), a package intended for one distribution will not be compatible with another distribution.

In order to perform the task of package management effectively, you need to be aware that you will have available two types of utilities: **low-level tools** (which handle in the backend the actual installation, upgrade, and removal of package files), and

high-level tools (which are in charge of ensuring that the tasks of dependency resolution and metadata searching -"data about the data"- are performed).

DISTRIBUTION	LOW-LEVEL TOOL	HIGH-LEVEL TOOL
Debian and derivatives	dpkg	apt-get / aptitude
CentOS	rpm	yum

The most frequent tasks that you will do with high and low level package management tools are listed in the following sections.

Installing a Package from a compiled (*.deb or *.rpm) file

The downside of this installation method is that no dependency resolution is provided. You will most likely choose to install a package from a compiled file when such package is not available in the distribution's repositories and therefore cannot be downloaded and installed through a high-level tool.

Since low-level tools do not perform dependency resolution, they will exit with an error if we try to install a package with unmet dependencies. To illustrate, we will use a fictitious file named file.deb for Debian and derivatives, and file.rpm for CentOS and similar distributions:

```
dpkg -i file.deb
```

```
rpm -i file.rpm
```

Upgrading a Package from a Compiled File

Again, you will only upgrade an installed package manually when it is not available in the central repositories.

```
dpkg -i file.deb
```

```
rpm -Uvh file.rpm
```

Listing Installed Packages

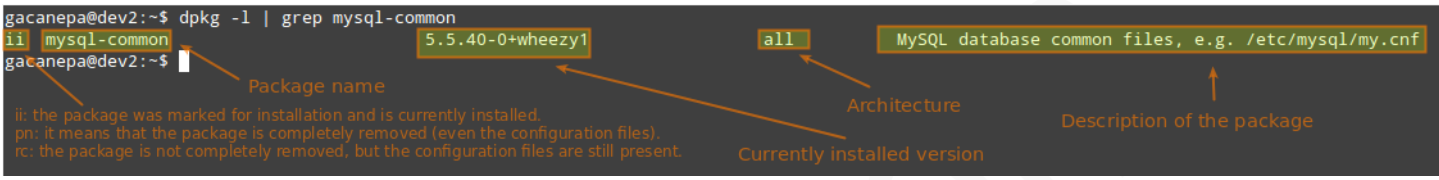
When you first get your hands on an already working system, chances are you'll want to know what packages are installed:


```
dpkg -l
```

```
rpm -qa
```

If you want to know whether a specific package is installed, you can pipe the output of the above commands to grep. Suppose we need to verify if package *mysql-common* is installed on an Ubuntu system:

```
dpkg -l | grep mysql-common
```



Another way to determine if a package is installed:

```
dpkg --status package_name
```

```
rpm -q package_name
```

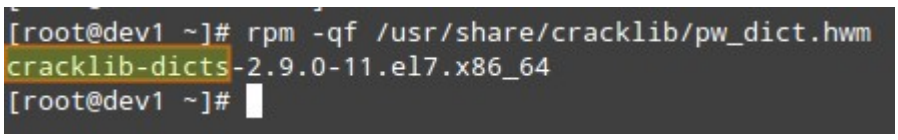
For example, let's find out whether package *sysdig* is installed on our system

Finding out which package installed a file

```
dpkg --search file_name
```

```
rpm -qf file_name # CentOS / openSUSE
```

For example, which package installed pw_dict.hwm?



Searching for a Package

To search for a package by name, do

```
aptitude search package_name
```

```
yum search package_name
```

```
yum search all package_name
```

When you use **search all**, **yum** will search for **package_name** not only in package names, but also in package descriptions.

Installing or updating a package from a repository

While installing a package, you may be prompted to confirm the installation after the package manager has resolved all dependencies. Note that running update or refresh (according to the package manager being used) is not strictly necessary, but keeping installed packages up to date is a good sysadmin practice for security and dependency reasons.

Depending on your chosen distro, you will use one of the following options to install **package_name**:

```
aptitude install package_name
```

```
yum install package_name
```

If you want to update a package that is already installed, replace **install** with **safe-upgrade** in *aptitude* and **install** with **update** in *yum*. Next, we'll learn how to remove a package using the same tools.

Removing a Package

In Debian and derivatives, do

```
aptitude remove package_name
```

or

```
aptitude purge package_name
```

where **remove** will uninstall the package but leave its configuration files intact, whereas **purge** will erase every trace of the program from your system.

Most (if not all) package managers will prompt you, by default, if you're sure about proceeding with the uninstallation before actually performing it. So read the onscreen messages carefully to avoid running into unnecessary trouble!

Displaying information about a package

After performing a search for a package, you may want to view additional information about it. Both aptitude and yum provide a method to do it:

```
aptitude show package_name
```

```
yum info package_name
```

The following command will display information about the *birthday* and *htop* packages:

```
aptitude show birthday
```

```
yum info htop
```

Exercises 6.1

6.1.1: Use **aptitude** to find information about **htop**, and then install it.

6.1.2: Use **yum search** and **search all** to find all packages with the word *tools* in their name or description.

Further Reading

Links to articles published in Tecmint.com...

http://www.tecmint.com/linux-package-management/

https://www.tecmint.com/20-practical-examples-of-rpm-commands-in-linux/

http://www.tecmint.com/dpkg-command-examples/

http://www.tecmint.com/20-linux-yum-yellowdog-updater-modified-commands-for-package-mangement/

https://www.tecmint.com/apt-advanced-package-command-examples-in-ubuntu/

Chapter 8 - Network Operations

In this chapter, you will learn:

- Installing and configuring an SSH server
- Copying files securely over the network
- Extra reading (links to related Tecmint articles)
- Exercises

As a system administrator you will often have to log on to remote systems to perform a variety of administration tasks using a terminal emulator. You will rarely sit in front of a real (physical) terminal, so you need to set up a way to log on remotely to the machines that you will be asked to manage.

In fact, that may be the last thing that you will have to do in front of a physical terminal. For security reasons, using Telnet for this purpose is not a good idea, as all traffic goes through the wire in unencrypted, plain text. Instead, we will use SSH - a network protocol that provides a secure way to access a remote server.

Installing and Securing a SSH Server

For you to be able to log on remotely to a remote system using SSH, you will have to do:

```
yum update && yum install openssh openssh-servers
```

```
or
```

```
aptitude install openssh-server
```

After installation, there is a couple of basic things that you need to take into account if you want to secure remote access to your SSH server. The following settings should be present in the */etc/ssh/sshd_config* file.

1) Change the port where the sshd daemon will listen on from 22 (the default value) to a high port (~2000 or greater), but first make sure the chosen port is not being used. For example, let's suppose you choose port 2500. Use **netstat** (a tool included in the

net-tools package in CentOS, and out of the box in Debian and derivatives) in order to check whether the chosen port is being used or not:

```
netstat -npltu | grep 2500
```

If the above command does not return anything, you can safely use port 2500 for sshd, and you should change the Port setting in the configuration file as follows:

```
Port 2500
```

2) Only allow SSH protocol 2:

```
Protocol 2
```

3) Configure the authentication timeout to 2 minutes, do not allow root logins, and restrict to a minimum the list of users which are allowed to login via ssh:

```
LoginGraceTime 2m
PermitRootLogin no
AllowUsers gacanepa
```

4) If possible, use key-based and disable password authentication:

PasswordAuthentication no
RSAAuthentication yes
PubkeyAuthentication yes

Before this, you will need to create a pair of private and public keys and copy the public one from your local machine to the server. To do it, use

```
ssh-keygen -t rsa
```

After running the above command, press Enter a few times to accept the default options until you get taken back to the command prompt. Next, go to your home directory and type

```
ssh-copy-id -i .ssh/id_rsa.pub192.168.0.100
```

to transfer the public key to the remote server with IP 192.168.0.100. You will be prompted to enter the password for your current account in the remote server once. Afterwards, you will be able to login without a password by doing

```
ssh 192.168.0.100
```


Transferring files securely over the network

If you need to ensure security while transferring or receiving files over a network, and specially if you need to perform that operation over the Internet, you will want to resort to 2 secure methods for file transfers: scp and sftp. Both should have been installed along with openssh.

Transferring files with scp (secure copy)

Use the **-P** flag if SSH on the remote hosts is listening on a port other than the default 22. The -p switch will preserve the permissions of local_file after the transfer, which will be made with the credentials of remote_user on remote_hosts. You will need to make sure that /absolute/path/to/remote/directory is writeable by this user.

```
scp -P XXXX -p local_file
remote_user@remote_host:/absolute/path/to/remote/directory
```

Receiving files with scp

You can also download files with scp from a remote host:

```
scp remote_user@remote_host:myFile.txt /absolute/path/to/local/directory
```

Or even between two remote hosts (in this case, copy the file *myFile.txt* from *remote_host1* to *remote_host2*):

```
scp
remote_user1@remote_host1:/absolute/path/to/remote/directory1/myFile.txt
remote_user1@remote_host2:/absolute/path/to/remote/directory2/
```

Don't forget to use scp -P followed by the port number if SSH is listening on a port other than the default 22.

Sending and receiving files with SFTP

Unlike SCP, SFTP does not require previously knowing the location of the file that we want to download or send.

This is the basic syntax to connect to a remote host using SFTP:

```
sftp -oPort=XXXX username@host
```


where *XXXX* represents the port where SSH is listening on host, which can be either a hostname or its corresponding IP address. You can omit the **-oPort** flag if SSH is listening on its default port (22).

Once the connection is successful, you can issue the following commands to send or receive files:

```
get -Pr [remote file or directory] # Receive files
put -r [local file or directory] # Send files
```

In both cases, the -r switch is used to recursively receive or send files, respectively. In the first case, the -P option will also preserve the original file permissions.

To close the connection, simply type “exit” or “bye”.

Exercises 7.1

7.1.1: What are the permissions of the *id_rsa* and *id_rsa.pub* files? Hint: these files are located in a hidden subdirectory named *.ssh* inside your */home* directory. Use the **file** command to determine their corresponding file types.

7.1.2: Transfer the file created in 5.1.1 to a directory named rem_files on the remote server (IP 192.168.0.100) via scp. Assume SSH is listening on port 10543 on the remote host. We are assuming the current local user account has write permissions on that directory

Further Reading

Links to articles published in Tecmint.com...

http://www.tecmint.com/ssh-interview-questions/

http://www.tecmint.com/sftp-command-examples/

http://www.tecmint.com/scp-commands-examples/

http://www.tecmint.com/sftp-upload-download-directory-in-linux/

Appendix A - Compression and Archiving

A file archiving tool groups a set of files into a single standalone file that we can backup to several types of media, transfer across a network, or send via email. The most frequently used archiving utility in Linux is **tar**.

When an archiving utility is used along with a compression tool, it allows to reduce the disk size that is needed to store the same files and information.

The tar utility

tar bundles a group of files together into a single archive (commonly called a **tar file** or **tarball**). The name originally stood for **t**ape **ar**chiver, but we must note that we can use this tool to archive data to any kind of writeable media (not only to tapes).

Tar is normally used with a compression tool such as **gzip**, **bzip2**, or **xz** to produce a compressed tarball.

Basic syntax:

```
tar [options] [path ...]
```

where … represents the expression used to specify which files should be acted upon.

Most commonly used tar commands

Long option	Abbreviation	Description
--create	c	Creates a tar archive
--concatenate	A	Appends tar files to an archive
--append	r	Appends files to the end of an archive
--update	u	Appends files newer than copy in archive
--diff or --compare	d	Find differences between archive and file system
--file ARCHIVE	f	Use archive file or device ARCHIVE
--list	t	Lists the contents of a tarball

--extract or --get	x	Extracts files from an archive

Normally used operation modifiers

Long option	Abbreviation	Description
--directory dir	C	Changes to directory **dir** before performing operations
--same-permissions	p	Preserves original permissions
--verbose	v	Lists all files read or extracted. When this flag is used along with --list, the file sizes, ownership, and time stamps are displayed.
--verify	W	Verifies the archive after writing it
--exclude file	---	Excludes file from the archive
--exclude=pattern	X	Exclude files, given as a PATTERN
--gzip or --gunzip	z	Processes an archive through gzip
--bzip2	j	Processes an archive through bzip2
--xz	J	Processes an archive through xz

The Gzip utility

Gzip is the oldest compression tool and provides the least compression, while bzip2 provides improved compression. In addition, xz is the newest but (usually) provides the best compression.

This advantages of best compression come at a price: the time it takes to complete the operation, and system resources used during the process.

Normally, tar files compressed with these utilities have .gz, .bz2, or .xz extensions, respectively. In the following examples we will be using these files: file1, file2, file3, file4, and file5.

EXAMPLE 1: Group all the files in the current working directory and compress the resulting bundle with gzip, bzip, and xz (please note the use of a regular expression to specify which files should be included in the bundle - this is to prevent the archiving tool to group the tarballs created in previous steps)

```
tar czf myfiles.tar.gz file[0-9]
tar cjf myfiles.tar.bz2 file[0-9]
tar cJf myfile.tar.xz file[0-9]
```

```
gacanepa@debian:~/LFCS/lab3$ ls -lh | grep tar
-rw-r--r-- 1 gacanepa gacanepa  79K Oct 13 09:44 myfiles.tar.bz2
-rw-r--r-- 1 gacanepa gacanepa 101K Oct 13 09:43 myfiles.tar.gz
-rw-r--r-- 1 gacanepa gacanepa  84K Oct 13 09:44 myfiles.tar.xz
gacanepa@debian:~/LFCS/lab3$
```

EXAMPLE 2: List the contents of a tarball and display the same information as a long directory listing. Note that update or append operations cannot be applied to compressed files directly (if you need to update or append a file to a compressed tarball, you need to uncompress the tar file and update / append to it, then compress again).

```
tar tvf [tarball]
```

```
gacanepa@debian:~/LFCS/lab3$ tar tvf myfiles.tar.gz
-rw-r--r-- gacanepa/gacanepa 7986 2014-10-13 09:16 file1
-rw-r--r-- gacanepa/gacanepa 2232 2014-10-13 09:16 file2
-rw-r--r-- gacanepa/gacanepa 3214 2014-10-13 09:17 file3
-rw-r--r-- gacanepa/gacanepa 321072 2014-10-13 09:20 file4
-rw-r--r-- gacanepa/gacanepa  37271 2014-10-13 09:21 file5
gacanepa@debian:~/LFCS/lab3$
```

Run any of the following commands:

- gzip -d myfiles.tar.gz
- bzip2 -d myfiles.tar.bz2
- xz -d myfiles.tar.xz

Then

```
tar --delete --file myfiles.tar file4 (deletes the file inside the tarball)
```

```
tar --update --file myfiles.tar file4 (adds the updated file)
```

and

1. gzip myfiles.tar, if you choose #1 above.
2. bzip2 myfiles.tar, if you choose #2.
3. xz myfiles.tar, if you choose #3.

Finally, do

```
tar tvf [tarball]
```

again

and compare the modification date and time of file4 with the same information as shown earlier.

EXAMPLE 3: Suppose you want to perform a backup of users' home directories. A good sysadmin practice would be (may also be specified by company policies) to exclude all video and audio files from backups.

Maybe your first approach would be to exclude from the backup all files with an .mp3 or .mp4 extension (or other extensions). What if you have a clever user who can change the extension to .txt or .bkp, your approach won't do you much good. In order to detect an audio or video file, you need to check its file type with file. The following shell script will do the job:

```
#!/bin/bash
# Pass the directory to backup as first argument.
DIR=$1
# Create the tarball and compress it. Exclude files with the MPEG string in its file type.
# -If the file type contains the string mpeg, $? (the exit status of the most recently
#  executed command) expands to 0, and the filename is redirected to the exclude option.
#  Otherwise, it expands to 1.
# -If $? equals 0, add the file to the list of files to be backed up.
tar X <(for i in $DIR/*; do file $i | grep -i mpeg; if [ $? -eq 0 ]; then echo $i; fi;done) -cjf backupfile.tar.bz2 $DIR/*
```


```
#!/bin/bash
# Pass the directory to backup as first argument.
DIR=$1
# Create the tarball and compress it. Exclude files with the MPEG string in its file type.
# -If the file type contains the string mpeg, $? (the exit status of the most recently executed command) expands to 0, and the filename is redirected to the exclude option. Otherwise, it expands to 1.
# -If $? equals 0, add the file to the list of files to be backed up.
tar X <(for i in $DIR/*; do file $i | grep -i mpeg; if [ $? -eq 0 ]; then echo $i; fi;done) -cjf backupfile.tar.bz2 $DIR/*
```


Summary

This book is intended as an introduction to the Linux command line and as a reference guide for intermediate and advanced users.

We believe learning Linux should not be difficult, and should not cost you an exaggerate amount of time or money.

We are not only passionate about Linux and other Free and Open Source technologies, but also about teaching those topics.

That is why, by buying this material, you don't just get the ebook to learn on your own – you also get our support to answer questions and free updates when we release them.

Happy learning!