Series basic Linux

Category

Day 1:

1/ What is Linux Shell and Environment variables?

2/ Prompt statements

3/ Permissions on file/folders and Owner/Group

What is Linux Shell and Environment variables?

What is Linux Shell ?

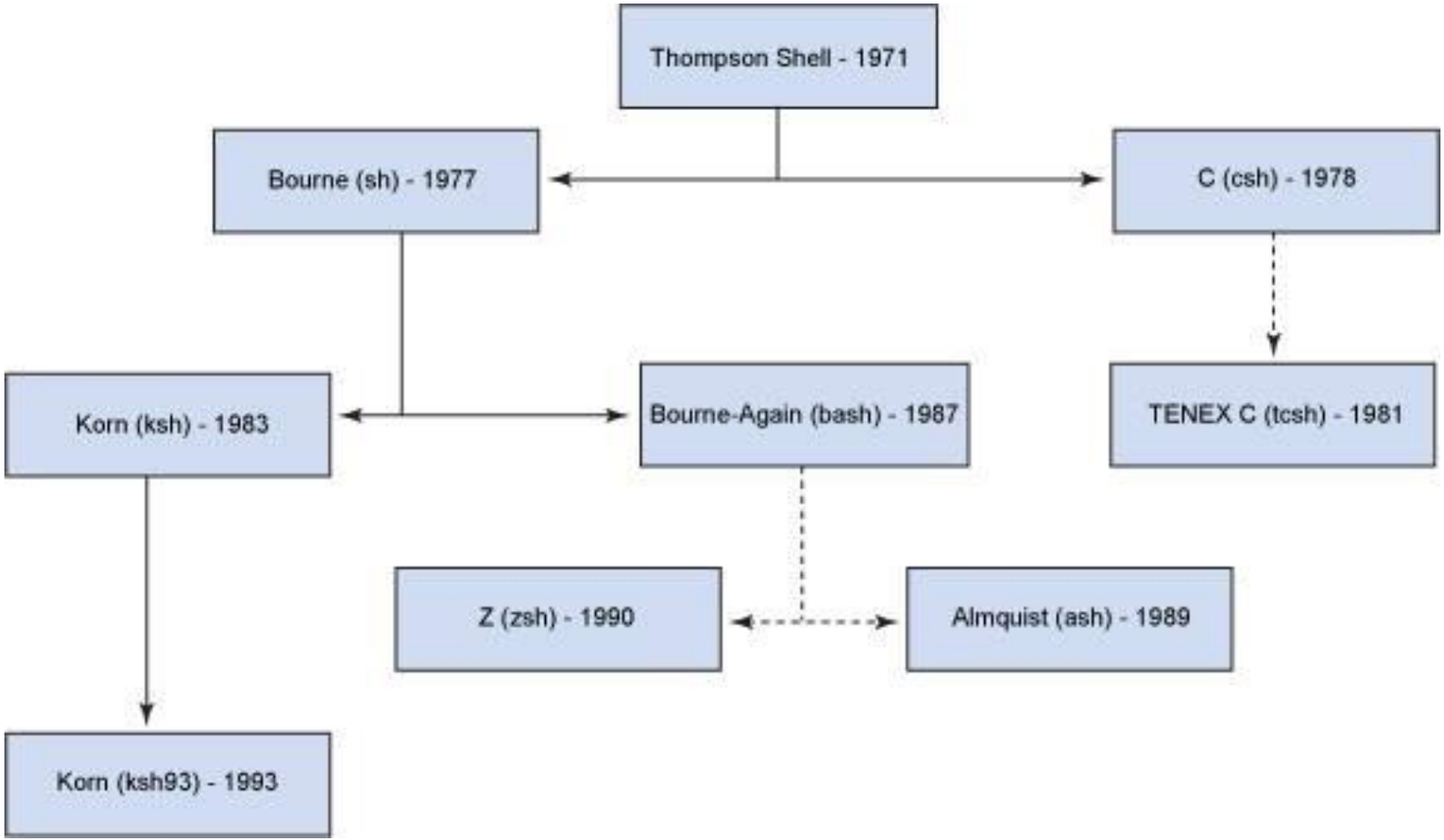
Computer understand the language of 0's and 1's called binary language.

In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in Os there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is pass to kernel.

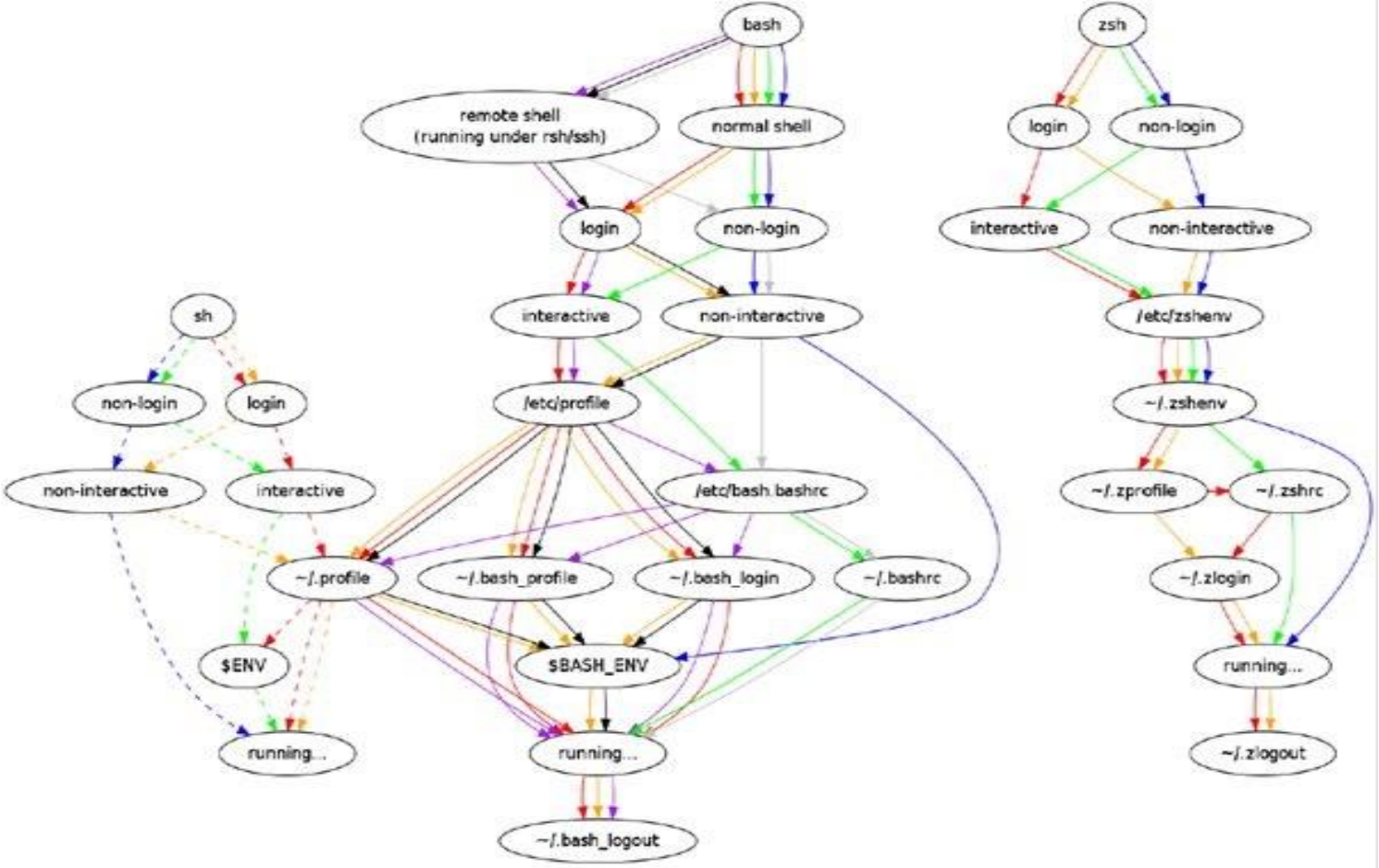
Shell is a user program or it's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

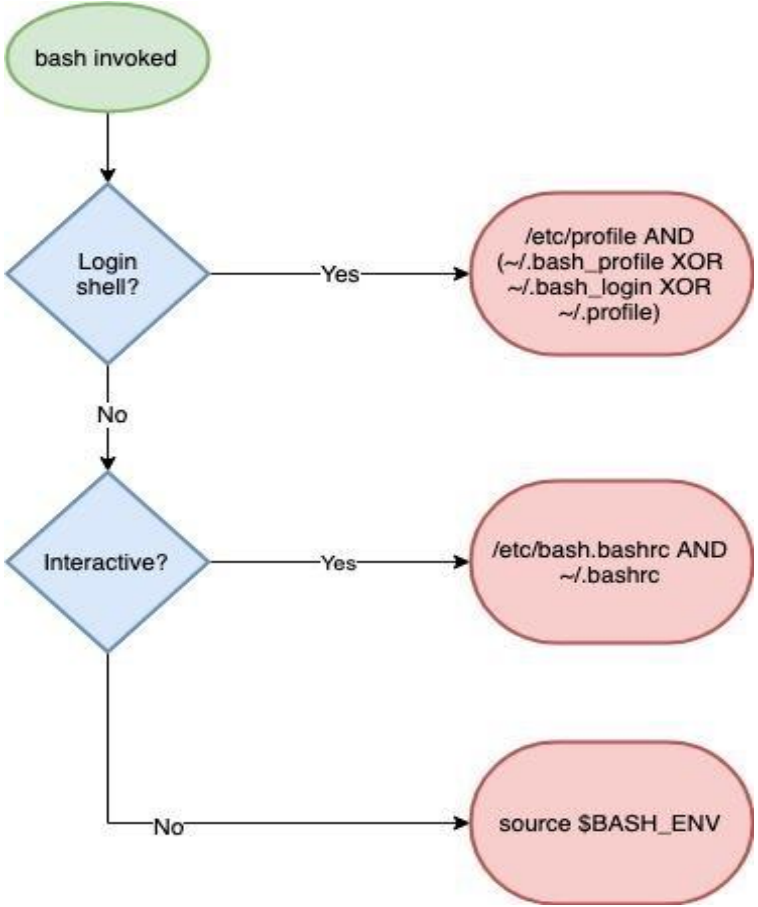
What is Linux Shell and Environment variables?



What is Linux Shell and Environment variables?



What is Linux Shell and Environment variables?



What is Linux Shell and Environment variables?

.bash_profile is executed for login shells

.bashrc is executed for interactive non-login shells.

What is Linux Shell and Environment variables?

When we use **.bashrc** ?

- Personal Linux PC.

What is Linux Shell and Environment variables?

When we use **.bash_profile** ?

- Remote user on servers (including .bahsrc for flexible.)
- User running crontab (--login = -l)

What is Linux Shell and Environment variables?

Environmental variables are variables that are defined for the current shell and are inherited by any child shells or processes. Environmental variables are used to pass information into processes that are spawned from the shell.

Shell variables are variables that are contained exclusively within the shell in which they were set or defined. They are often used to keep track of ephemeral data, like the current working directory.

By convention, these types of variables are usually defined using all capital letters. This helps users distinguish environmental variables within other contexts.

What is Linux Shell and Environment variables?

Environment Variable:

Edit/ Create variable:

bash$ export NAME=Value

bash$ env NAME=Value

Delete variable:

bash$ unset NAME

Edit $PATH:

bash$ export PATH=$PATH:/opt/software_path

or

bash$ export PATH=/opt/software_path:$PATH

Prompt statements

- **PS1**: environment variable which contains the value of the default prompt. It changes the shell command prompt appearance and environment.
- **PS2**: environment variable which contains the value the prompt used for a command continuation interpretation. You see it when you write a long command in many lines.
- **PS3**: environment variable which contains the value of the prompt for the select operator inside the shell script.
- **PS4**: environment variable which contains the value of the prompt used to show script lines during the execution of a bash script in debug mode.

Prompt statements

PS1 is the default prompt we see every time when we log in the console. For the most news Linux systems, the defaults values have **\u@\h:\w\$** which show the **username, hostname, the current working directory** and **the user privilege.**

Example 1: Display only the username and the hostname with the separation character “-”:
export PS1="\u - \h$ "
server01$ echo $PS1
\u - \h$

Example 2: Passing string to PS1 variable
export PS1=happy-test$

Example 3: Add time to the prompt with **\t** and the working directory with **\w**
export PS1="[\t]\n\u@\h:\w\$ "
export PS1="[\t]\u@\h:\w\$ "

Prompt statements

PS2 When we are in the console, we can need to associate many commands in one command. It makes the command too long for one line, so it can be broken down into multiple lines by giving “\” at the end of each line. The default interactive PS2 value prompt for a multi-line command is “>“ which indicates that you can continue the command on the second line and so on.

```
$ echo $PS2
>
```

Example 1: The default usage of PS2

```
# apt-get update && \
> apt-get -y install mysql-client python-setuptools curl git unzip apache2 php && \
> apt-get upgrade
```

Example 2: You will replace the value by a sentence. Note the escape before the last quote

```
# export PS2=”incomplete? continue here-> ”
# apt-get update && \
incomplete? continue here-> apt-get -y install curl git unzip apache2 && \
incomplete? continue here-> apt-get upgrad
```

Prompt statements

PS3 is used by the select operator inside a bash script. It is difficult to have its value on a simple console. To show what we are talking about, we need to write a simple bash script which will help us to see the value. The default PS3 value prompt is “#?“

Example 1: We will copy the content below on a file named ps3-value.sh

```
#!/bin/bash
echo "please select a value to display a month on the list below"
select i in jan feb mar exit
do
      case $i in
      jan) echo "January";;
        feb) echo "February";;
         mar) echo "March";;
      exit) exit;;
      esac
   done
```

Prompt statements

Modify the default value to “choice” and display the it by executing the script

```
$ export PS3="choice: "
$./ps3-value.sh
Select a value to display a month on the below
1) jan
2) feb
3) mar
4) exit
choice: 2
February
   choice:
```

See that **choice** is our new value

Prompt statements

The **PS4** shows each line of a bash script when we are in debug mode before executing lines. It helps to know which line doesn’t give the result attend. We can display the value of PS4 with echo command. But in a bash script, we can see it using **bash -x** for the execution. The default **PS4** value prompt is “**+**“

Prompt statements

Example 1: Create a file ps4-value.sh with the same content of ps3-value.sh and modify the default value

```
$ echo $PS4
+
$ export PS4="[script line->:] "
$ chmod +x ps4-value.sh
$ bash -x ps4-value.sh
[script line->: ]echo 'Please select a value to display a month on the list below'
Please select a value to display a month on the list below
[script line->: ]select i in jan feb mar apr bye
1) jan
2) feb
3) mar
4) apr
5) bye
#? 3
[script line->: ]case $i in
[script line->: ]echo March
March
#? 1
[script line->: ]case $i in
[script line->: ]echo January
   January
```

Permissions on file/folders and Owner/Group

owner: the owner of the object - the default is the user who created the object initially.

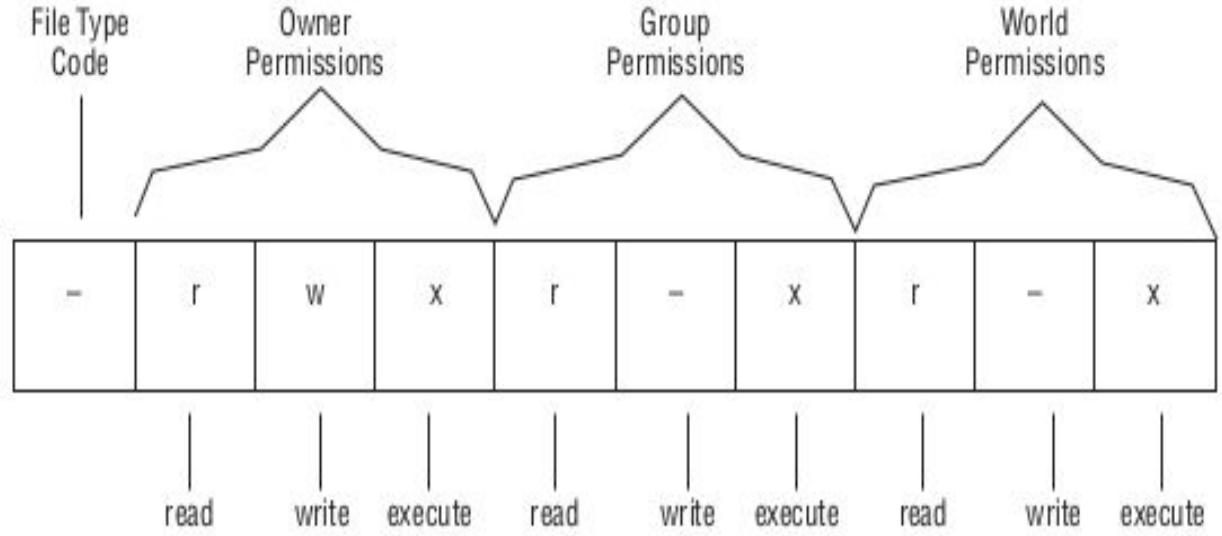
group: group of users sharing the same access rights - the default is the group the owner belongs to above.

others: all users are not in the above two groups.

Permissions on file/folders and Owner/Group

-: Normal file
d: Directory
l: Link
c: Special file
s: Socket
p: Named pipe
b: Equipment

Permissions on file/folders and Owner/Group



1: execute permission
2: write permission
4: read permission

Permissions on file/folders and Owner/Group

Permission set code	Meaning	Change type code	Meaning	Permission to modify code	Meaning
u	Owner	+	Add	r	Read
g	Group	-	Remove	w	Write
o	Other	=	Set equal to	x	Execute
a	All			X	Execute only if the file is a directory or already has execute permission
				s	SUID or SGID
				t	Sticky bit
				u	Existing owner's permissions
				g	Existing group permissions
				o	Existing other permissions

Permissions on file/folders and Owner/Group

example:

Command	Initial permissions	End permissions
`chmod a+x bigprogram`	`rw-r--r--`	`rwxr-xr-x`
`chmod ug=rw report.tex`	`r--------`	`rw-rw----`
`chmod o-rwx bigprogram`	`rwxrwxr-x`	`rwxrwx---`
`chmod g=u report.tex`	`rw-r--r--`	`rw-rw-r--`
`chmod g-w,o-rw report.tex`	`rw-rw-rw-`	`rw-r-----`

Permissions on file/folders and Owner/Group

Setting the Default Mode and Group

When a user creates a file, that file has default ownership and permissions. The default owner is, understandably, the user who created the file. The default group is the user's primary group.

The default permissions are configurable. These are defined by the user mask, which is set by the umask command. This command takes as input an octal value that represents the bits to be removed from 777 permissions for directories, or from 666 permissions for files, when a new file or directory is created.

Permissions on file/folders and Owner/Group

The default umask 002 used for normal user. With this mask default directory permissions are 775 and default file permissions are 664.

The default umask for the root user is 022 result into default directory permissions are 755 and default file permissions are 644.

For directories, the base permissions are (rwxrwxrwx) 0777 and for files they are 0666 (rw-rw-rw

Permissions on file/folders and Owner/Group

Sample umask values and their effects

umask	Created files	Created directories
000	666 (rw-rw-rw-)	777 (rwxrwxrwx)
002	664 (rw-rw-r--)	775 (rwxrwxr-x)
022	644 (rw-r--r--)	755 (rwxr-xr-x)
027	640 (rw-r-----)	750 (rwxr-x---)
077	600 (rw-------)	700 (rwx------)
277	400 (r--------)	500 (r-x------)

Permissions on file/folders and Owner/Group

SETUID

When the **setuid** bit is used, the behavior described above it's modified so that when an executable is launched, it does not run with the privileges of the user who launched it, but with that of the file owner instead.

```
ls -l /bin/passwd
-rwsr-xr-x. 1 root root 27768 Feb 11  2017 /bin/passwd
```

Permissions on file/folders and Owner/Group

GUID

Unlike the **setuid** bit, the **setgid** bit has effect on both files and directories. In the first case, the file which has the setgid bit set, when executed, instead of running with the privileges of the group of the user who started it, runs with those of the group which owns the file: in other words, the group ID of the process will be the same of that of the file.

```
ls -ld test
drwxrwsr-x. 2 egdoc egdoc 4096 Nov  1 17:25 test
```

This time the `s` is present in place of the executable bit on the group sector.

Permissions on file/folders and Owner/Group

STICKY BIT

The **sticky bit** works in a different way: while it has no effect on files, when used on a directory, all the files in said directory will be modifiable only by their owners. A typical case in which it is used, involves the **/tmp** directory. Typically this directory is writable by all users on the system, so to make impossible for one user to delete the files of another one, the sticky bit is set:

```
$ ls -ld /tmp
drwxrwxrwt. 14 root root 300 Nov  1 16:48 /tmp
```

In this case the owner, the group, and all other users, have full permissions on the directory (read, write and execute). The sticky bit is identifiable by a t which is reported where normally the executable x bit is shown, in the "other" section. Again, a lowercase t implies that the executable bit is also present, otherwise you would see a capital T.

Permissions on file/folders and Owner/Group

How to set special bits?

Apply setuid to a file:

```
$ chmod 4555 [path_to_file]
$ chmod u+s file
```

Apply setgid to a file:

```
$ chmod 4555 [path_to_file]
$ chmod g+s file
```

Apply stickybit:

```
$ chmod 1777 [path_to_directory]
$ chmod o+t test
$ chmod +t [path_to_directory]
```

Permissions on file/folders and Owner/Group

Changing File Attributes

Append Only The **a** attribute sets append mode, which disables write access to the file except for appending data. This can be a security feature to prevent accidental or malicious changes to files that record data, such as log files.

Compressed The **c** attribute causes the kernel to compress data written to the file automatically and uncompress it when it's read back.

Immutable The **i** flag makes a file immutable, which goes a step beyond simply disabling write access to the file. The file can't be deleted, links to it can't be created, and the file can't be renamed.

Data Journaling The **j** flag tells the kernel to journal all data written to the file. This improves recoverability of data written to the file after a system crash but can slow performance. This flag has no effect on ext2 filesystems.

Secure Deletion Ordinarily, when you delete a file, its directory entry is removed and its inode is marked as being available for recycling. The data blocks that make up the bulk of the file aren't erased. Setting the **s** flag changes this behavior; when the fi le is deleted, the kernel zeros its data blocks, which may be desirable for files that contain sensitive data.

No Tail-Merging Tail-merging is a process in which small data pieces at a fi le's end that don't fi ll a complete block are merged with similar pieces of data from other files. The result is reduced disk space consumption, particularly when you store many small files rather than a few big ones. Setting the **t** flag disables this behavior, which is desirable if certain non-kernel drivers will read the filesystem, such as those that are part of the Grand Unified Boot Loader (GRUB).

Permissions on file/folders and Owner/Group

For instance, to add the immutable flag to the important.txt fi le, you enter the following command:

 # chattr +i important.txt

The result is that you'll be unable to delete the file, even as root . To delete the file, you must first remove the immutable flag:

 # chattr -i important.txt