Chapter 12

Identifying the current user

As you know, Linux is a multi-user operating system, meaning more than one user can log on at the same time.

- To identify the current user, type **whoami**.
- To list the currently logged-on users, type **who**.

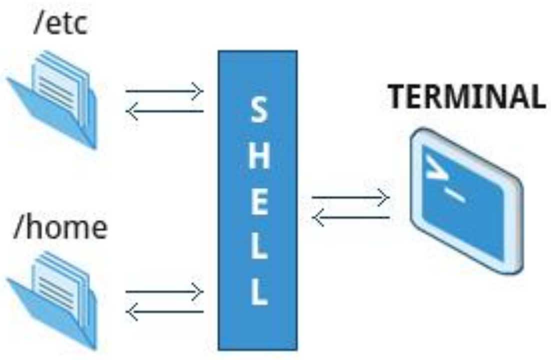
Giving **who** the **-a** option will give more detailed information.

User startup files

As you know, Linux is a multi-user operating system, meaning more than one user can log on at the same time.

- To identify the current user, type **whoami**.
- To list the currently logged-on users, type **who**.

Giving **who** the **-a** option will give more detailed information.



Order of the startup files

The standard prescription is that when you first login to Linux, **/etc/profile** is read and evaluated, after which the following files are searched (if they exist) in the listed order:

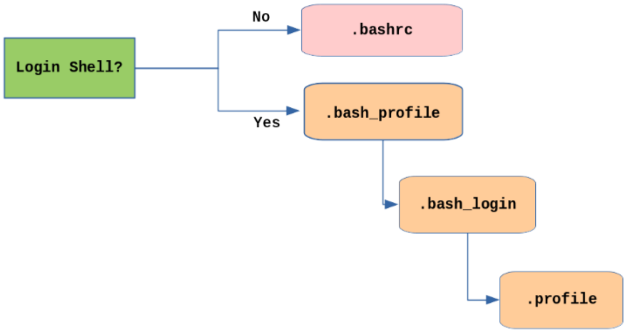
1. **~/.bash_profile**
2. **~/.bash_login**
3. **~/.profile**

where **~/.** denotes the user's home directory. The Linux login shell evaluates whatever startup file that it comes across first and ignores the rest. This means that if it finds **~/.bash_profile**, it ignores **~/.bash_login** and **~/.profile**. Different distributions may use different startup files.

However, every time you create a new shell, or terminal window, etc., you do not perform a full system login; only a file named **~/.bashrc** file is read and evaluated. Although this file is not read and evaluated along with the login shell, most distributions and/or users include the **~/.bashrc** file from within one of the three user-owned startup files.

Most commonly, users only fiddle with **~/.bashrc**, as it is invoked every time a new command line shell initiates, or another program is launched from a terminal window, while the other files are read and executed only when the user first logs onto the system.

Recent distributions sometimes do not even have **.bash_profile** and/or **.bash_login** , and some just do little more than include **.bashrc**.



Creating aliases

You can create customized commands or modify the behavior of already existing ones by creating **aliases**. Most often, these aliases are placed in your **~/.bashrc** file so they are available to any command shells you create. **unalias** removes an alias.

Typing **alias** with no arguments will list currently defined aliases.

Please note there should not be any spaces on either side of the equal sign and the alias definition needs to be placed within either single or double quotes if it contains any spaces.

Basics of users and groups

All Linux users are assigned a unique user ID (**uid**), which is just an integer; normal users start with a uid of 1000 or greater.

Linux uses **groups** for organizing users. Groups are collections of accounts with certain shared permissions. Control of group membership is administered through the **/etc/group** file, which shows a list of groups and their members. By default, every user belongs to a default or primary group. When a user logs in, the group membership is set for their primary group and all the members enjoy the same level of access and privilege. Permissions on various files and directories can be modified at the group level.

Users also have one or more group IDs (**gid**), including a default one which is the same as the user ID. These numbers are associated with names through the files **/etc/passwd** and **/etc/group**. Groups are used to establish a set of users who have common interests for the purposes of access rights, privileges, and security considerations. Access rights to files (and devices) are granted on the basis of the user and the group they belong to.

For example, **/etc/passwd** might contain **george:x:1002:1002:George Metesky:/home/george:/bin/bash** and **/etc/group** might contain **george:x:1002**.

Adding and removing users

Adding a new group is done with **groupadd**:

```
$ sudo /usr/sbin/groupadd anewgroup
```

The group can be removed with:

```
$ sudo /usr/sbin/groupdel anewgroup
```

Adding a user to an already existing group is done with **usermod**. For example, you would first look at what groups the user already belongs to:

```
$ groups rjsquirrel
bjmoose : rjsquirrel
```

and then add the new group:

```
$ sudo /usr/sbin/usermod -a -G anewgroup rjsquirrel
```

```
$ groups rjsquirrel
rjsquirrel: rjsquirrel anewgroup
```

These utilities update **/etc/group** as necessary. Make sure to use the **-a** option, for append, so as to avoid removing already existing groups. **groupmod** can be used to change group properties, such as the Group ID (gid) with the **-g** option or its name with then **-n** option.

Removing a user from the group is somewhat trickier. The **-G** option to usermod must give a complete list of groups. Thus, if you do:

```
$ sudo /usr/sbin/usermod -G rjsquirrel rjsquirrel
```

```
$ groups rjsquirrel
rjsquirrel : rjsquirrel
```

only the **rjsquirrel** group will be left.

The root account

The root account is very powerful and has full access to the system. Other operating systems often call this the administrator account; in Linux, it is often called the superuser account. You must be extremely cautious before granting full root access to a user; it is rarely, if ever, justified. External attacks often consist of tricks used to elevate to the root account.

However, you can use **sudo** to assign more limited privileges to user accounts:

- Only on a temporary basis
- Only for a specific subset of commands.

su and sudo

When assigning elevated privileges, you can use the command **su** (switch or substitute user) to launch a new shell running as another user (you must type the password of the user you are becoming). Most often, this other user is root, and the new shell allows the use of elevated privileges until it is exited. It is almost always a bad (dangerous for both security and stability) practice to use **su** to become root. Resulting errors can include deletion of vital files from the system and security breaches.

Granting privileges using **sudo** is less dangerous and is preferred. By default, **sudo** must be enabled on a per-user basis. However, some distributions (such as Ubuntu) enable it by default for at least one main user, or give this as an installation option.

Elevating to root account

To temporarily become the superuser for a series of commands, you can type **su** and then be prompted for the root password.

To execute just one command with root privilege type **sudo <command>**. When the command is complete, you will return to being a normal unprivileged user.

sudo configuration files are stored in the **/etc/sudoers** file and in the **/etc/sudoers.d/** directory. By default, the **sudoers.d** directory is empty.

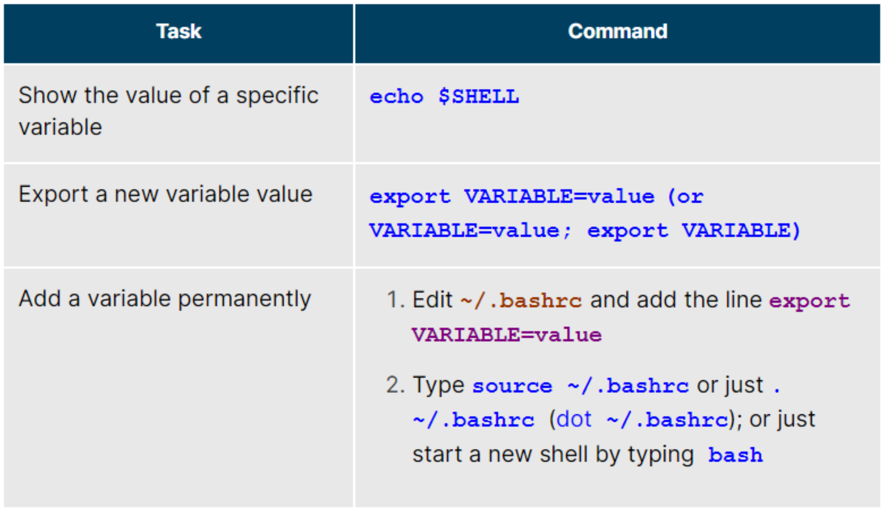
Environment variables

Environment variables are quantities that have specific values which may be utilized by the command shell, such as **bash**, or other utilities and applications. Some environment variables are given preset values by the system (which can usually be overridden), while others are set directly by the user, either at the command line or within startup and other scripts.

An environment variable is actually just a character string that contains information used by one or more applications. There are a number of ways to view the values of currently set environment variables; one can type **set**, **env**, or **export.** Depending on the state of your system, **set** may print out many more lines than the other two methods.

Setting environment variables

By default, variables created within a script are only available to the current shell; child processes (sub-shells) will not have access to values that have been set or modified. Allowing child processes to see the values requires use of the **export** command.



Task	Command
Show the value of a specific variable	`echo $SHELL`
Export a new variable value	`export VARIABLE=value (or VARIABLE=value; export VARIABLE)`
Add a variable permanently	1. Edit `~/.bashrc` and add the line `export VARIABLE=value` 2. Type `source ~/.bashrc` or just `. ~/.bashrc` (dot `~/.bashrc`); or just start a new shell by typing `bash`

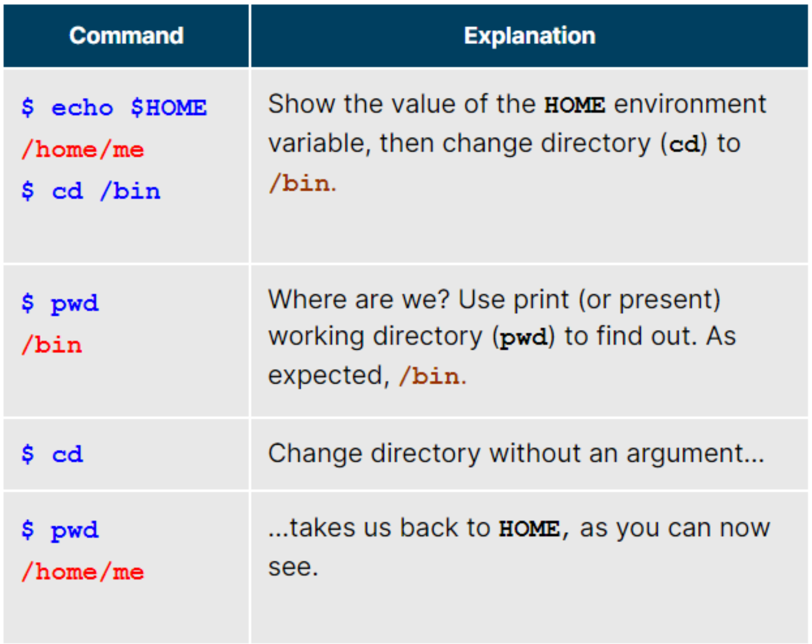
You can also set environment variables to be fed as a one shot to a command as in:

$ SDIRS=s_0* KROOT=/lib/modules/$(uname -r)/build make modules_install

which feeds the values of the **SDIRS** and **KROOT** environment variables to the command **make modules_install**.

The HOME Variable

HOME is an environment variable that represents the home (or login) directory of the user. **cd** without arguments will change the current working directory to the value of **HOME**. Note the tilde character (**~**) is often used as an abbreviation for **$HOME**. Thus, **cd $HOME** and **cd ~** are completely equivalent statements.



Command	Explanation

`$ echo $HOME`
`/home/me`
`$ cd /bin`
	Show the value of the `HOME` environment variable, then change directory (`cd`) to `/bin`.

`$ pwd`
`/bin`
	Where are we? Use print (or present) working directory (`pwd`) to find out. As expected, `/bin`.

`$ cd`	Change directory without an argument...

`$ pwd`
`/home/me`
	...takes us back to `HOME`, as you can now see.

The PATH variable

PATH is an ordered list of directories (the path) which is scanned when a command is given to find the appropriate program or script to run. Each directory in the path is separated by colons (**:**). A null (empty) directory name (or **./**) indicates the current directory at any given time.

- **:path1:path2**
- **path1::path2**

In the example **:path1:path2**, there is a null directory before the first colon (**:**). Similarly, for **path1::path2** there is a null directory between **path1** and **path2**.

To prefix a private **bin** directory to your path:

```
$ export PATH=$HOME/bin:$PATH
$ echo $PATH
/home/student/bin:/usr/local/bin:/usr/bin:/bin/usr
```

The SHELL variable

The environment variable **SHELL** points to the user's default command shell (the program that is handling whatever you type in a command window, usually bash) and contains the full pathname to the shell:

```
$ echo $SHELL
/bin/bash
$
```

The PS1 Variable and the command line prompt

Prompt Statement (**PS**) is used to customize your prompt string in your terminal windows to display the information you want.

PS1 is the primary prompt variable which controls what your command line prompt looks like. The following special characters can be included in **PS1**:

\u - User name
\h - Host name
\w - Current working directory
\! - History number of this command
\d - Date

They must be surrounded in single quotes when they are used, as in the following example:

```
$ echo $PS1
$
$ export PS1='\u@\h:\w$ '
student@example.com:~$ # new prompt
```

To revert the changes:

```
student@example.com:~$ export PS1='$ '
$
```

An even better practice would be to save the old prompt first and then restore, as in:

```
$ OLD_PS1=$PS1
```

change the prompt, and eventually change it back with:

```
$ PS1=$OLD_PS1
$
```

Previous commands

bash keeps track of previously entered commands and statements in a history buffer. You can recall previously used commands simply by using the **Up** and **Down** cursor keys. To view the list of previously executed commands, you can just type **history** at the command line.

The list of commands is displayed with the most recent command appearing last in the list. This information is stored in **~/.bash_history**. If you have multiple terminals open, the commands typed in each session are not saved until the session terminates.

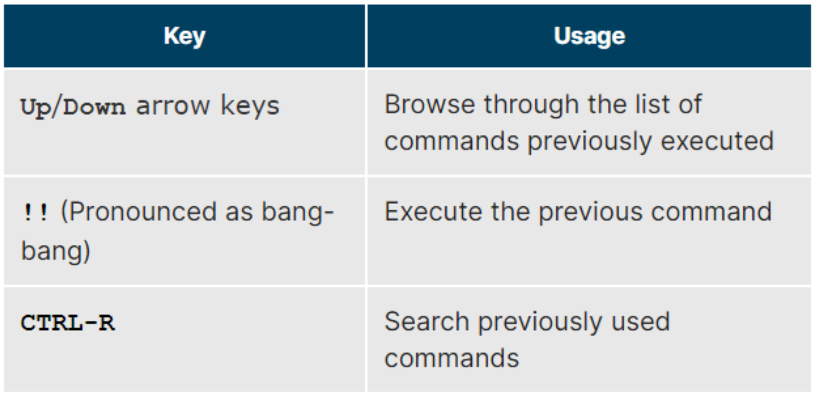
Using history environment variables

Several associated environment variables can be used to get information about the **history** file.

- **HISTFILE**
  The location of the history file.
- **HISTFILESIZE**
  The maximum number of lines in the history file (default 500).
- **HISTSIZE**
  The maximum number of commands in the history file.
- **HISTCONTROL**
  How commands are stored.
- **HISTIGNORE**
  Which command lines can be unsaved.

For a complete description of the use of these environment variables, see **man bash**.

Finding and using previous commands



Key	Usage
`Up`/`Down` arrow keys	Browse through the list of commands previously executed
`!!` (Pronounced as bang-bang)	Execute the previous command
`CTRL-R`	Search previously used commands

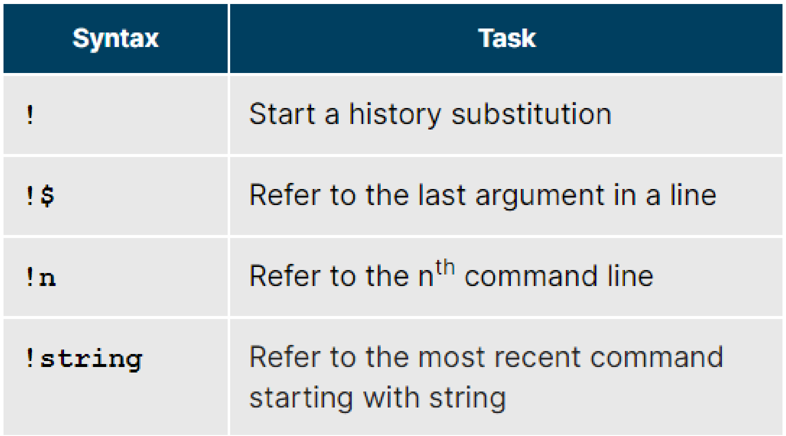
If you want to recall a command in the history list, but do not want to press the arrow key repeatedly, you can press **CTRL-R** to do a reverse intelligent search.

As you start typing, the search goes back in reverse order to the first command that matches the letters you have typed. By typing more successive letters, you make the match more and more specific.

The following is an example of how you can use the **CTRL-R** command to search through the command history:

```
$ ^R                                              (This all happens on 1 line)
(reverse-i-search)'s': sleep 1000   (Searched for 's'; matched "sleep")
$ sleep 1000                                  (Pressed Enter to execute the searched
command)
$
```

Executing previous commands



Syntax	Task
!	Start a history substitution
!$	Refer to the last argument in a line
!n	Refer to the nth command line
!string	Refer to the most recent command starting with string

All history substitutions start with **!**. When typing the command: **ls -l /bin /etc** will refer to **/var**, the last argument to the command.

Here are more examples:

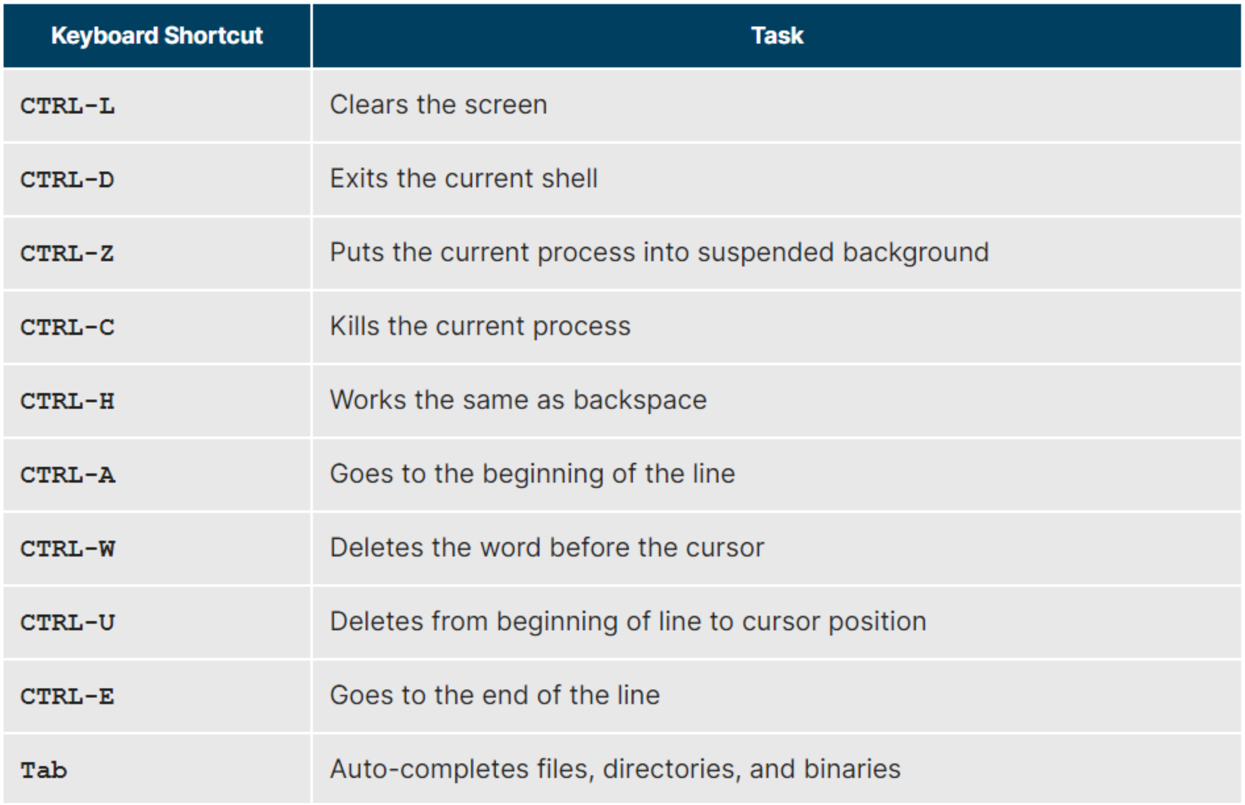
$ history

1. **echo $SHELL**
2. **echo $HOME**
3. **echo $PS1**
4. **ls -a**

5. **ls -l /etc/ passwd**
6. **sleep 1000**
7. **history**

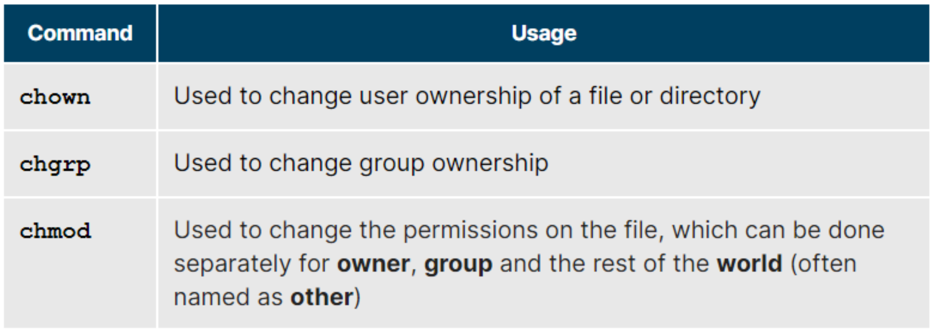
```
$ !1                                (Execute command #1 above)
echo $SHELL
/bin/bash
$ !sl                          (Execute the command beginning with "sl")
sleep 1000
$
```

Keyboard shortcuts



Keyboard Shortcut	Task
CTRL-L	Clears the screen
CTRL-D	Exits the current shell
CTRL-Z	Puts the current process into suspended background
CTRL-C	Kills the current process
CTRL-H	Works the same as backspace
CTRL-A	Goes to the beginning of the line
CTRL-W	Deletes the word before the cursor
CTRL-U	Deletes from beginning of line to cursor position
CTRL-E	Goes to the end of the line
Tab	Auto-completes files, directories, and binaries

File ownership



Command	Usage
chown	Used to change user ownership of a file or directory
chgrp	Used to change group ownership
chmod	Used to change the permissions on the file, which can be done separately for **owner**, **group** and the rest of the **world** (often named as **other**)

File permission modes and chmod

Files have three kinds of permissions: read (**r**), write (**w**), execute (**x**). These are generally represented as in **rwx**. These permissions affect three groups of owners: user/owner (**u**), group (**g**), and others (**o**).

As a result, you have the following three groups of three permissions:

```
rwx: rwx: rwx
 u:   g:   o
```

There are a number of different ways to use **chmod**. For instance, to give the owner and others execute permission and remove the group write permission:

```
$ ls -l somefile
-rw-rw-r-- 1 student student 1601 Mar 9 15:04 somefile
$ chmod uo+x,g-w somefile
$ ls -l somefile
-rwxr--r-x 1 student student 1601 Mar 9 15:04 somefile
```

where **u** stands for user (owner), **o** stands for other (world), and **g** stands for group.

This kind of syntax can be difficult to type and remember, so one often uses a shorthand which lets you set all the permissions in one step. This is done with a simple algorithm, and a single digit suffices to specify all three permission bits for each entity. This digit is the sum of:

- **4** if read permission is desired
- **2** if write permission is desired
- **1** if execute permission is desired.

Thus, **7** means read/write/execute, **6** means read/write, and **5** means read/execute.

When you apply this to the **chmod** command, you have to give three digits for each degree of freedom, such as in:

```
$ chmod 755 somefile
$ ls -l somefile
-rwxr-xr-x 1 student student 1601 Mar 9 15:04 somefile
```

Summary

- Linux is a multi-user system.
- To find the currently logged on users, you can use the **who** command.
- To find the current user ID, you can use the **whoami** command.
- The **root** account has full access to the system. It is never sensible to grant full root access to a user.
- You can assign root privileges to regular user accounts on a temporary basis using the **sudo** command.
- The shell program (bash) uses multiple startup files to create the user environment. Each file affects the interactive environment in a different way. **/etc/profile** provides the global settings.
- Advantages of startup files include that they customize the user's prompt, set the user's terminal type, set the command-line shortcuts and aliases, and set the default text editor, etc.
- An environment variable is a character string that contains data used by one or more applications. The built-in shell variables can be customized to suit your requirements.
- The **history** command recalls a list of previous commands, which can be edited and recycled.
- In Linux, various keyboard shortcuts can be used at the command prompt instead of long actual commands.
- You can customize commands by creating aliases. Adding an alias to ~**/.bashrc** will make it available for other shells.
- File permissions can be changed by typing **chmod permissions filename**.

- File ownership is changed by typing **chown owner filename**.
- File group ownership is changed by typing **chgrp group filename**.