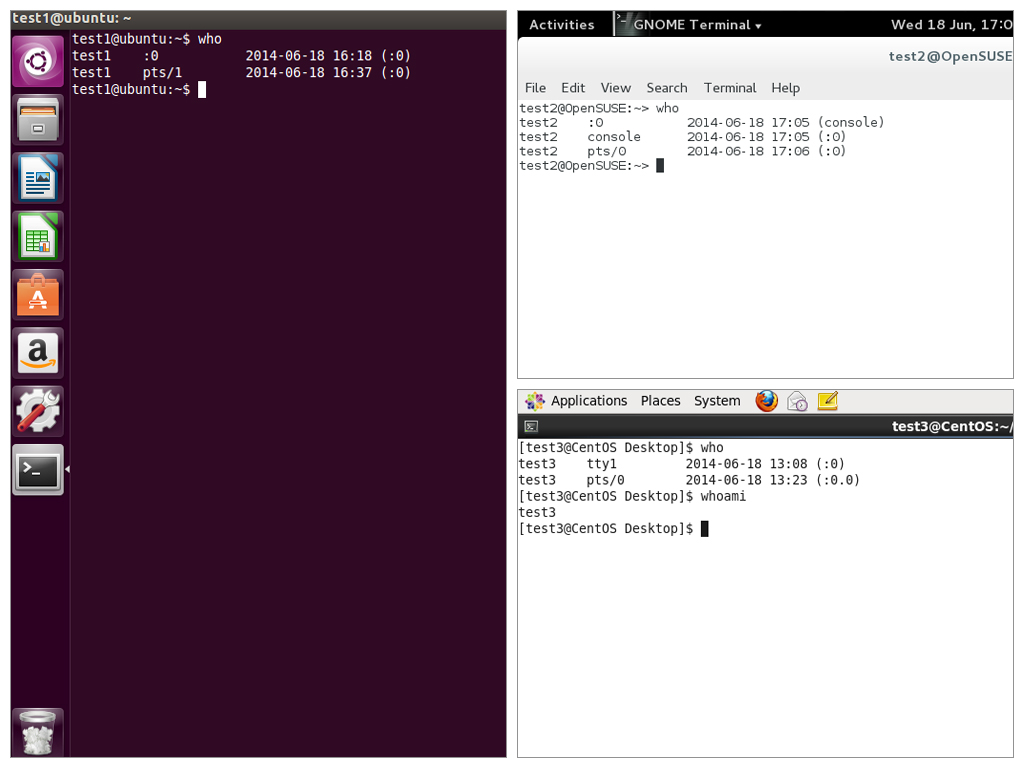
**Identifying the Current User**

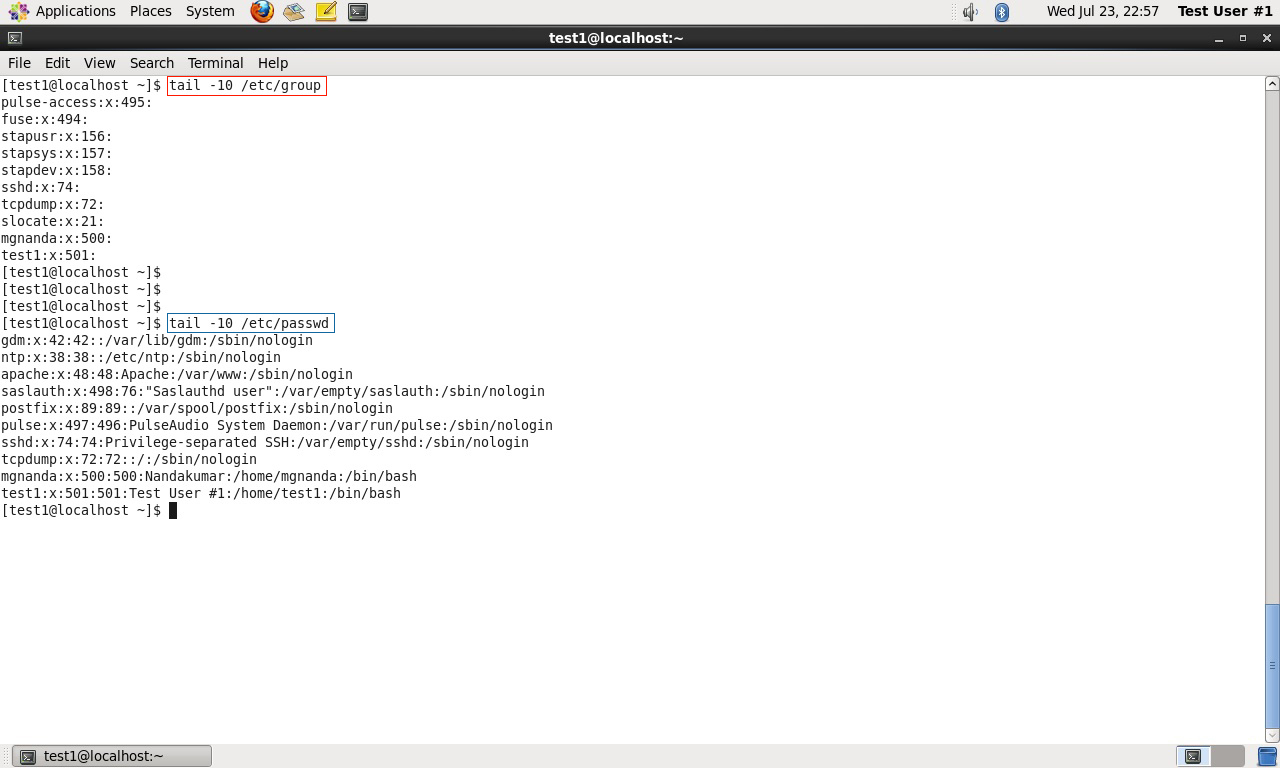
As you know, Linux is a multiuser operating system; i.e., more than one user can log on at the same time.

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

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

Click the image to view an enlarged version.

**Basics of Users and Groups**



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.

All Linux users are assigned a unique user ID (**uid**), which is just an integer, as well as one or more group ID’s (**gid**), including a default one which is the same as the user ID.

Historically **Fedora**-family systems start **uid**'s at 500; other distributions begin at 1000.

These numbers are associated with names through the files /etc/passwd and /etc/group.

For example, the first file might contain:

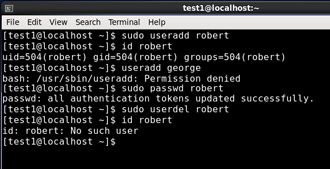
george:x:1002:1002:George Metesky:/home/george:/bin/bash

and the second george:x:1002

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.

Click the image to view an enlarged version.

**Adding and Removing Users**

Distributions have straightforward graphical interfaces for creating and removing users and groups and manipulating group membership. However, it is often useful to do it from the command line or from within shell scripts. Only the root user can add and remove users and groups.

Adding a new user is done with **useradd** and removing an existing user is done with **userdel**. In the simplest form an account for the new user turkey would be done with:

$ sudo useradd turkey

which by default sets the home directory to /home/turkey, populates it with some basic files (copied from /etc/skel) and adds a line to /etc/passwd such as:

turkey:x:502:502::/home/turkey:/bin/bash

and sets the default shell to /bin/bash. Removing a user account is as easy as typing userdel turkey However, this will leave the /home/turkey directory intact. This might be useful if it is a temporary inactivation. To remove the home directory while removing the account one needs to use the **-r** option to **userdel**.

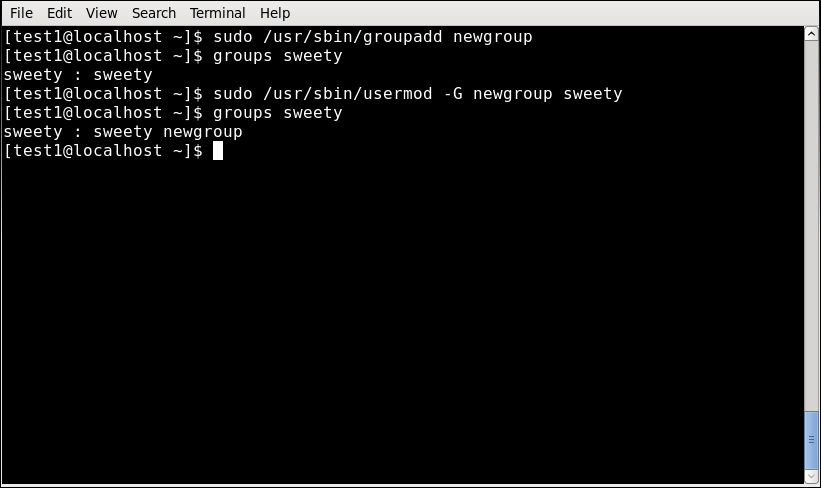
Typing **id** with no argument gives information about the current user, as in:

$ id

uid=500(george) gid=500(george) groups=106(fuse),500(george)

If given the name of another user as an argument, **id** will report information about that other user.

**Adding and Removing Groups**

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 turkey

turkey : turkey

and then add the new group:

$ sudo /usr/sbin/usermod -G anewgroup turkey

$ groups turkey

turkey: turkey anewgroup

These utilities update /etc/group as necessary. **groupmod** can be used to change group properties such as the Group ID (gid) with the -g option or its name with the -n option.

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

$ sudo /usr/sbin/usermod -G turkey turkey

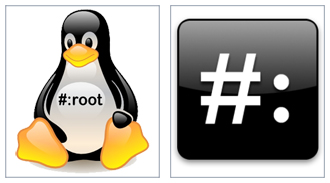
$ groups turkey

turkey : turkey

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

Click the image to view an enlarged version.

**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 the **sudo** feature to assign more limited privileges to user accounts:

* on only a temporary basis.
* only for a specific subset of commands.

**su and sudo**

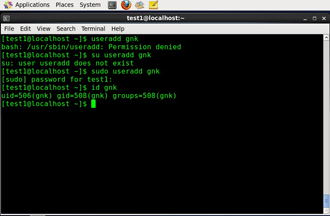


When assigning elevated privileges, you can use the command **su** (switch or substitute user) to launch a new shell running as another user (you must type the password of the user you are becoming). Most often this other user is root, and the new shell allows the use of elevated privileges until it 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.

In the chapter on Security that follows shortly, we will describe and compare **su** and **sudo** in detail.

**Elevating to root Account**

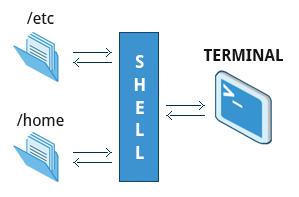


To fully become root, one merely types **su** and then is prompted for the root password.

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.

**Startup Files**



In Linux, the command shell program (generally **bash)**  uses one or more startup files to configure the environment. Files in the /etc directory define global settings for all users while Initialization files in the user's home directory can include and/or override the global settings.

The startup files can do anything the user would like to do in every command shell, such as:

* Customizing the user's prompt
* Defining command-line shortcuts and aliases
* Setting the default text editor
* Setting the **path** for where to find executable programs

**Order of the Startup Files**

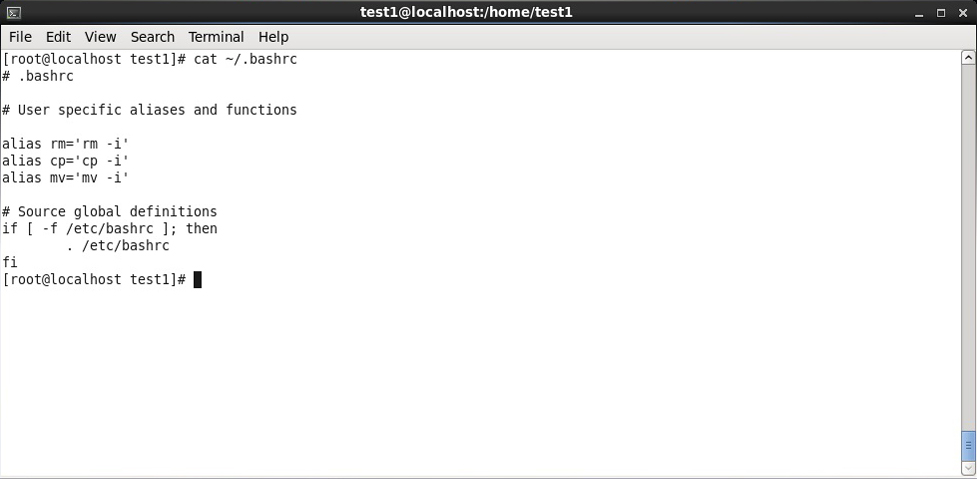
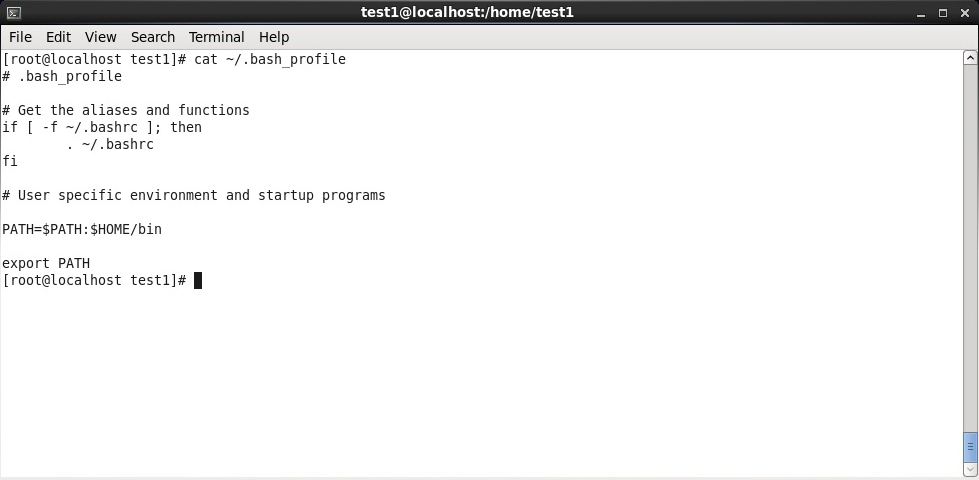
When you first login to Linux, /etc/profile is read and evaluated, after which the following files are searched (if they exist) in the listed order:

1. ~/.bash\_profile
2. ~/.bash\_login
3. ~/.profile

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 the ~/.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. In the **Ubuntu**, **openSuse**, and **CentOS** distros, the user must make appropriate changes in the ~/.bash\_profile file to include the ~/.bashrc file.

The .bash\_profile will have certain extra lines, which in turn will collect the required customization parameters from .bashrc.



**Environment Variables**

**Environment variables** are simply named quantities that have specific values and are understood by the command shell, such as **bash**. Some of these are pre-set (built-in) by the system, and others are set by the user either at the command line or within startup and other scripts. An environment variable is actually no more than 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.

$ set

BASH=/bin/bash

BASHOPTS=checkwinsize:cmdhist:expand\_aliases:extglob:extquote:force\_fignore

BASH\_ALIASES=()

...

$ env

SSH\_AGENT\_PID=1892

GPG\_AGENT\_INFO=/run/user/me/keyring-Ilf3vt/gpg:0:1

TERM=xterm

SHELL=/bin/bash

...

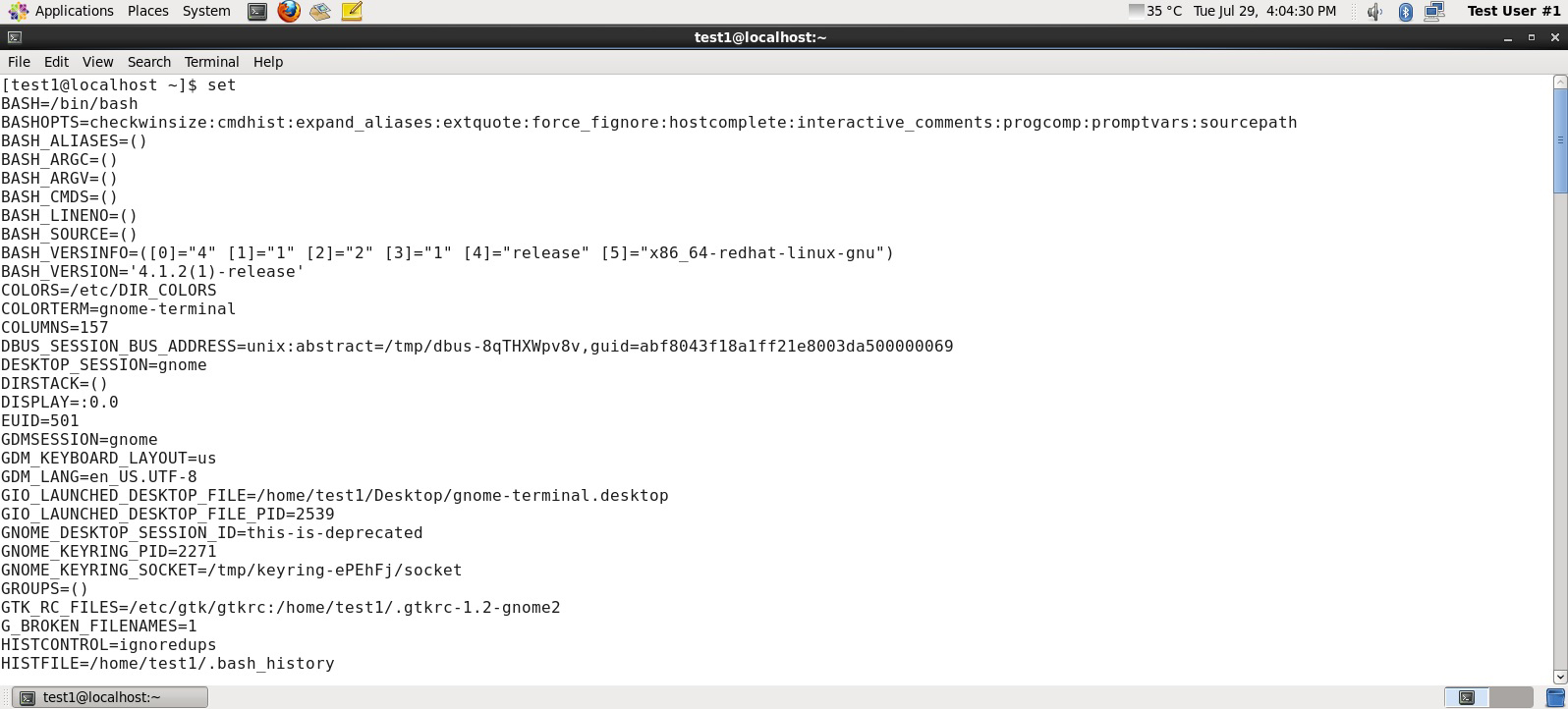
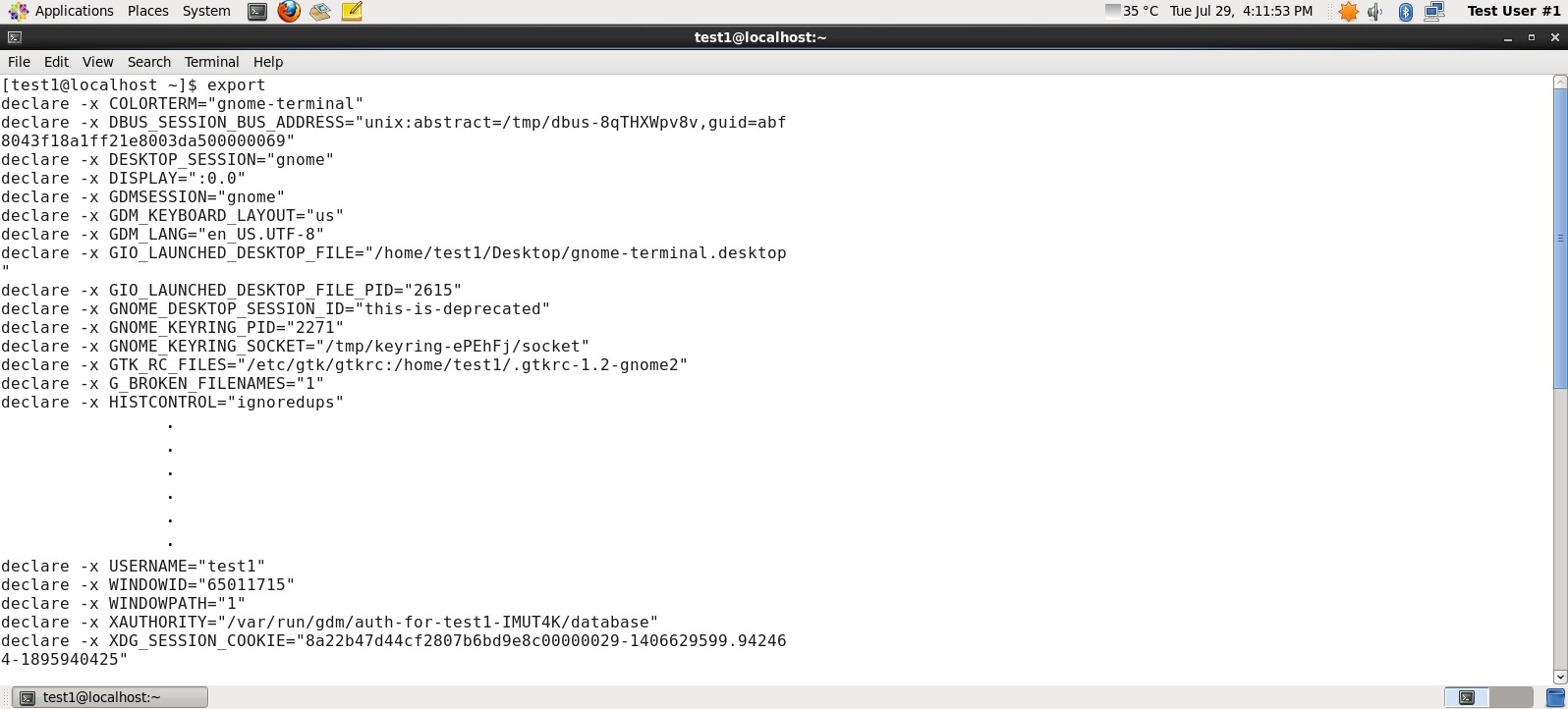
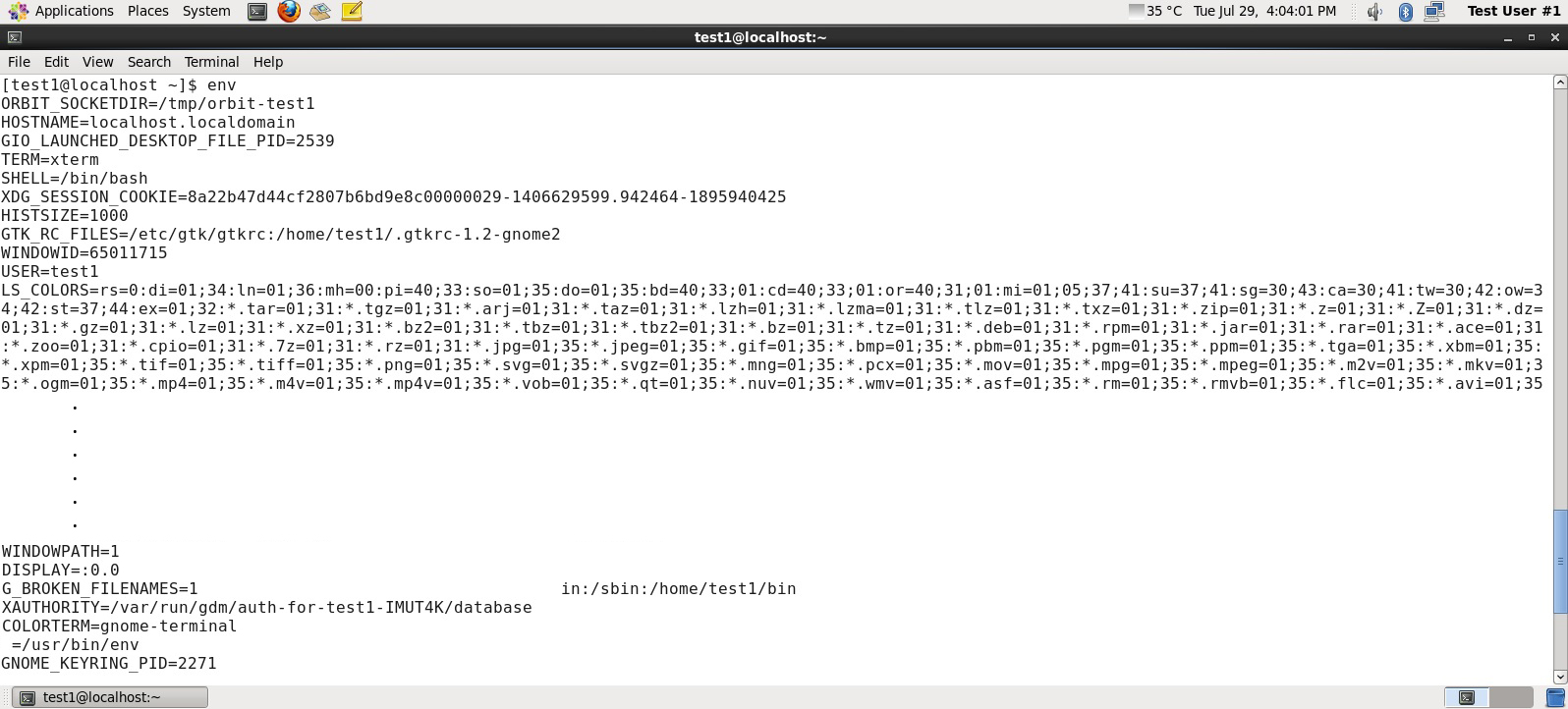
$ export

declare -x COLORTERM=gnome-terminal

declare -x COMPIZ\_BIN\_PATH=/usr/bin /

declare -x COMPIZ\_CONFIG\_PROFILE=ubuntu

...



Click the image to view an enlarged version.

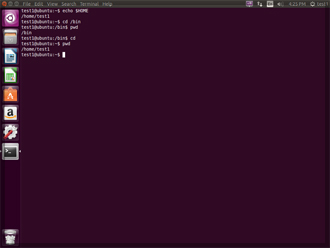
**Setting Environment Variables**

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 |

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/3d99ae48ff724532a3ba43c73080ead7/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/3d99ae48ff724532a3ba43c73080ead7/1#)

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

To prefix a private bin directory to your path:

$ export PATH=$HOME/bin:$PATH

$ echo $PATH

/home/me/bin:/usr/local/bin:/usr/bin:/bin/usr

Click the image to view an enlarged version.

**The PS1 Variable**

**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$ '

me@example.com:~$ # new prompt

me@example.com:~$

To revert the changes:

me@example.com:~$ export PS1='$ '

$

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:

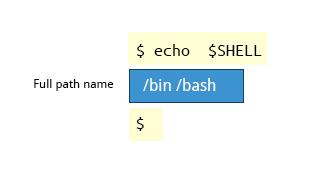
$ PS1=$OLD\_PS1

$

Click the image to view an enlarged version.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/3d99ae48ff724532a3ba43c73080ead7/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/3d99ae48ff724532a3ba43c73080ead7/1#)

**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:

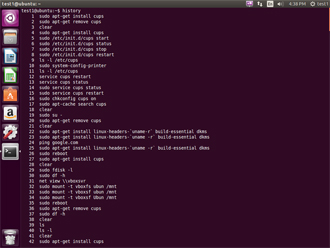
$ echo $SHELL

/bin/bash

$

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/3d99ae48ff724532a3ba43c73080ead7/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/3d99ae48ff724532a3ba43c73080ead7/1#)

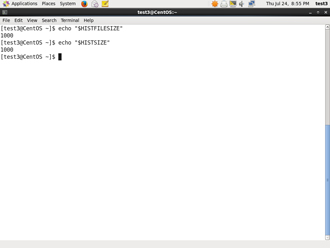
**Recalling Previous Commands**

**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.

Click the image to view an enlarged version.

**Using History Environment Variables**



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

HISTFILE stores the location of the history file.

HISTFILESIZE stores the maximum number of lines in the history file.

HISTSIZE stores the maximum number of lines in the history file for the current session.

Click the image to view an enlarged version.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/1bdb9eaf134a426085d62c5bb75d3b1b/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/1bdb9eaf134a426085d62c5bb75d3b1b/1#)

**Finding and Using Previous Commands**

Specific keys to perform various tasks:

|  |  |
| --- | --- |
| **Key** | **Usage** |
| Up/Down arrow key | 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've 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

$

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/1bdb9eaf134a426085d62c5bb75d3b1b/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/1bdb9eaf134a426085d62c5bb75d3b1b/1#)

**Executing Previous Commands**

The table describes the syntax used to execute previously used 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 !. In the line $ ls -l /bin /etc /var !$ refers to /var, which is the last argument in the line.

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

$

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/1bdb9eaf134a426085d62c5bb75d3b1b/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/17e43829df2840b997fa26e1bc5602af/1bdb9eaf134a426085d62c5bb75d3b1b/1#)

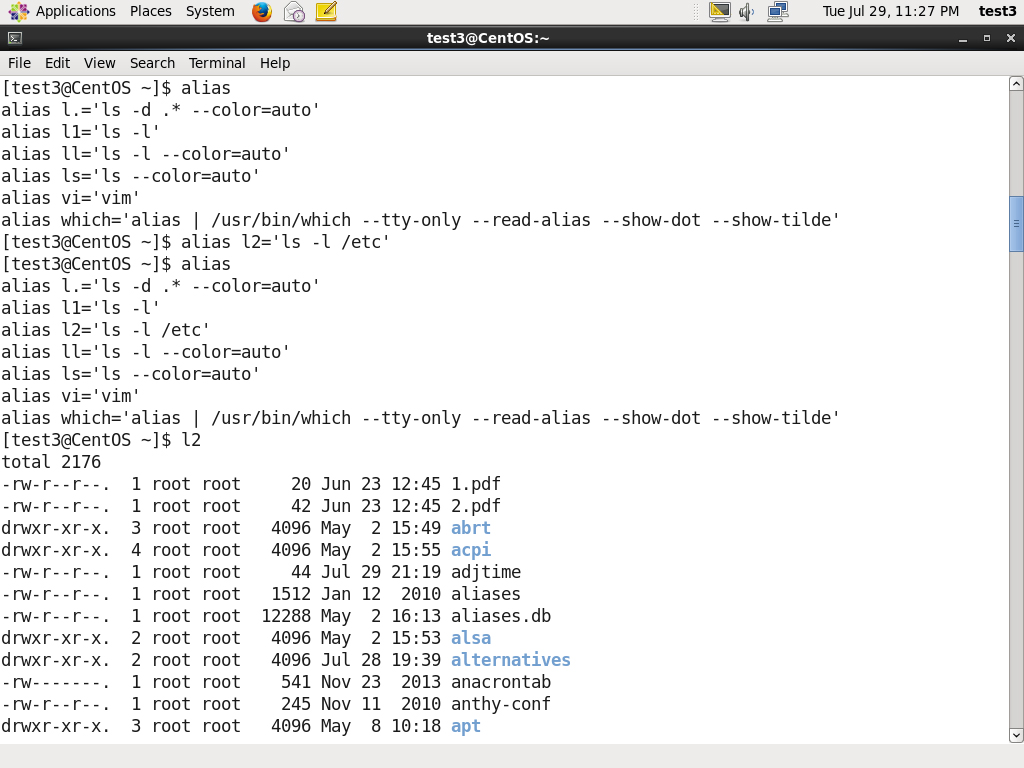
**Keyboard Shortcuts**

You can use keyboard shortcuts to perform different tasks quickly. The table lists some of these keyboard shortcuts and their uses.

|  |  |
| --- | --- |
| **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 |

**Note: The next screen covers the Try-It-Yourself activity through which you can practice the procedure.**

**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.

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.

Click the image to view an enlarged version for an example.

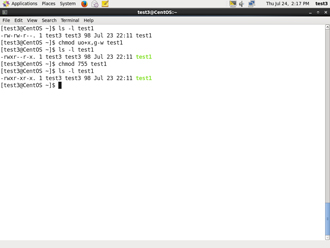
**File Ownership**

In Linux and other UNIX-based operating systems, every file is associated with a user who is the **owner**. Every file is also associated with a **group** (a subset of all users) which has an interest in the file and certain rights, or permissions: read, write, and execute.

The following utility programs involve user and group ownership and permission setting.

|  |  |
| --- | --- |
| **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 a\_file

-rw-rw-r-- 1 coop coop 1601 Mar 9 15:04 a\_file

$ chmod uo+x,g-w a\_file

$ ls -l a\_file

-rwxr--r-x 1 coop coop 1601 Mar 9 15:04 a\_file

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.

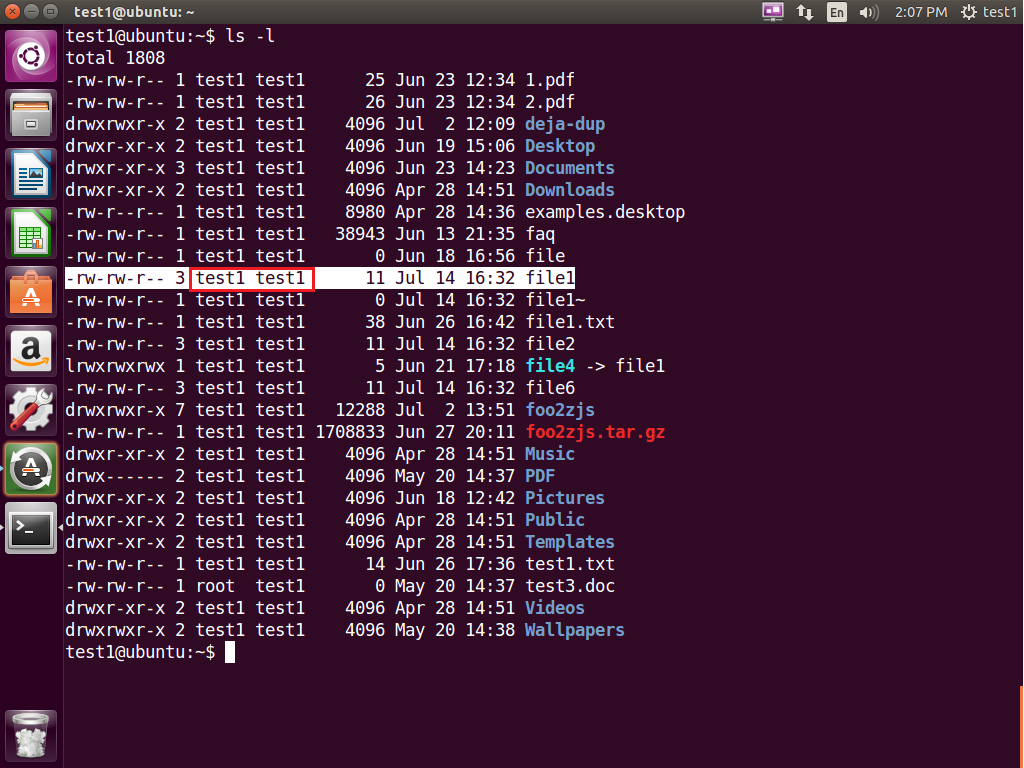
Thus 7 means read/write/execute, 6 means read/write, and 5 means read/execute.

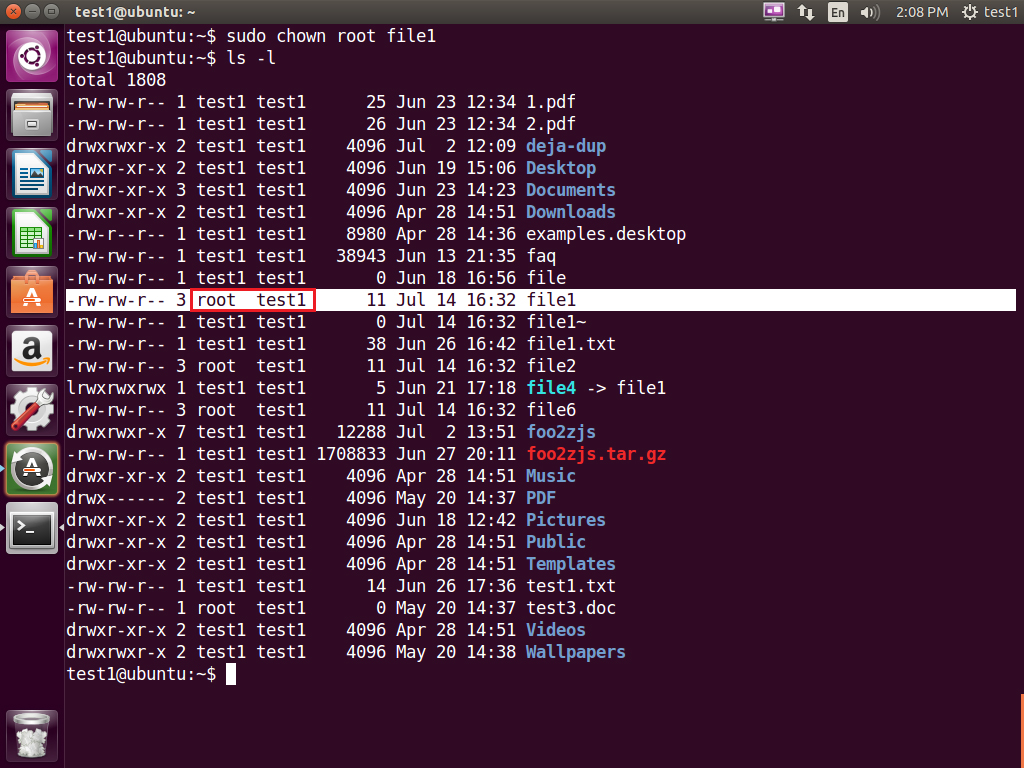
When you apply this to the **chmod** command you have to give three digits for each degree of freedom, such as in

$ chmod 755 a\_file

$ ls -l a\_file

-rwxr-xr-x 1 coop coop 1601 Mar 9 15:04 a\_file





Let's see an example of changing file ownership using **chown:**

The first image shows the permissions for owners/groups/all users on 'file1'. The second image shows the change in permissions for the different users on "file1"

$ ls -l

total 4

-rw-rw-r--. 1 bob bob 0 Mar 16 19:04 file-1

-rw-rw-r--. 1 bob bob 0 Mar 16 19:04 file-2

drwxrwxr-x. 2 bob bob 4096 Mar 16 19:04 temp

$ sudo chown root file-1

[sudo] password for bob:

$ ls -l

total 4

-rw-rw-r--. 1 root bob 0 Mar 16 19:04 file-1

-rw-rw-r--. 1 bob bob 0 Mar 16 19:04 file-2

drwxrwxr-x. 2 bob bob 4096 Mar 16 19:04 temp

**Example of chgrp**

Now let’s see an example of changing group ownership using **chgrp**:

The image on LHS shows the group with their permissions on 'file1'.

The image on RHS shows the change in groups and thier permissions on "file1"

$ sudo chgrp bin file-2

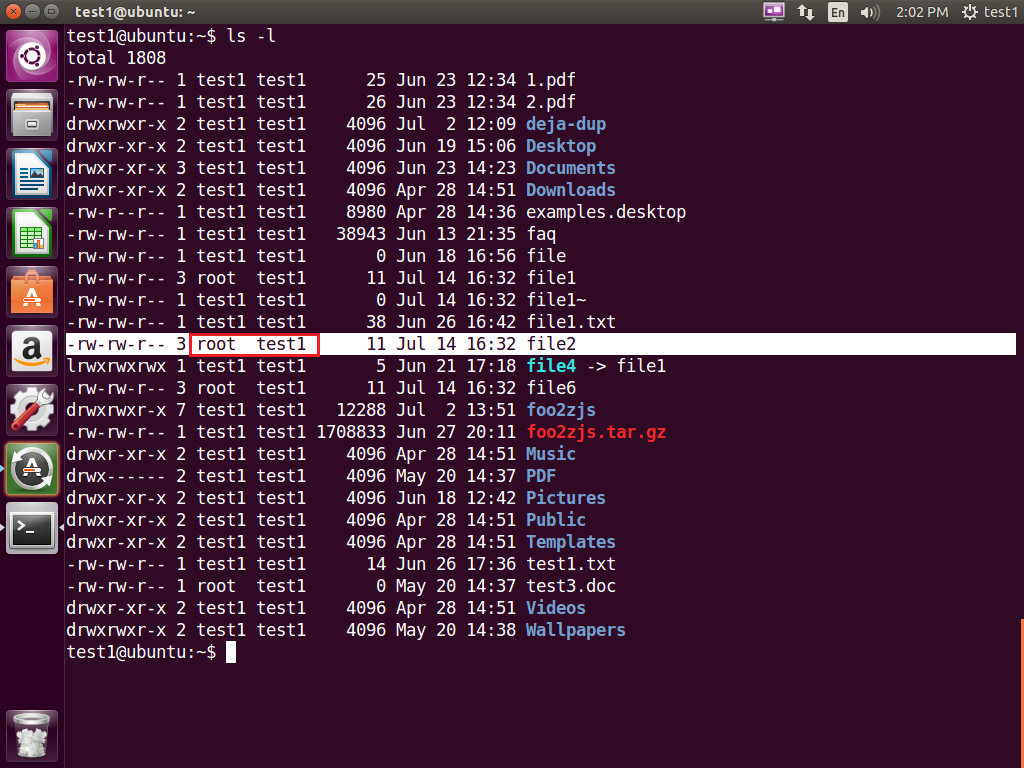
$ ls -l

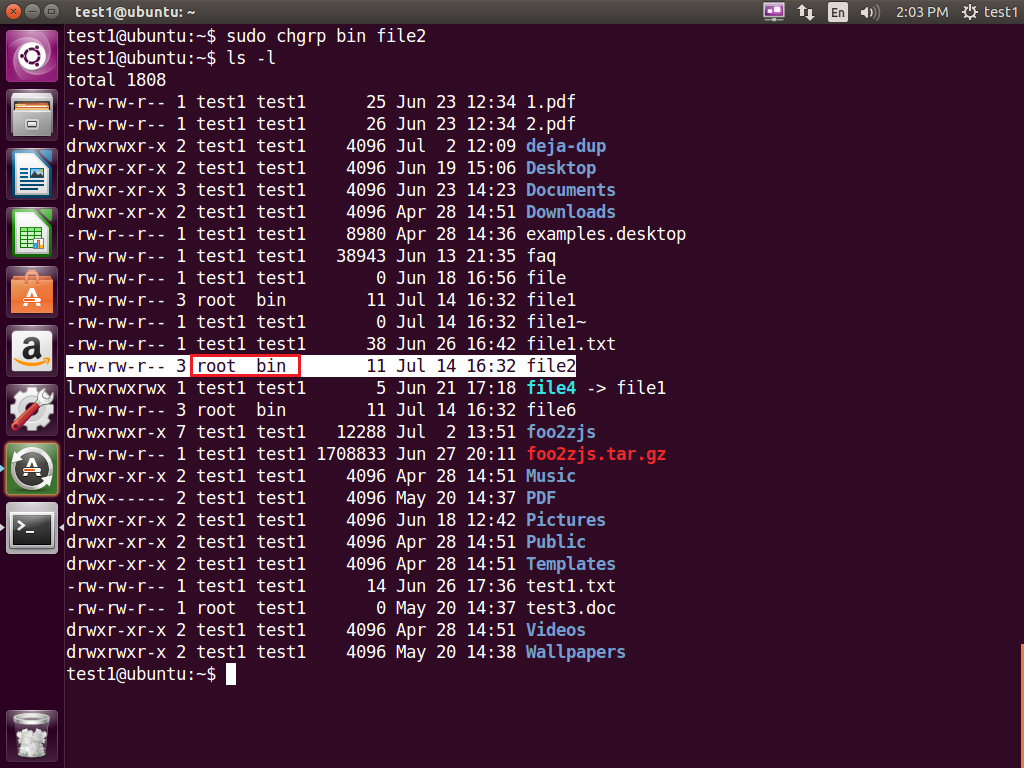
total 4

-rw-rw-r--. 1 root bob 0 Mar 16 19:04 file-1

-rw-rw-r--. 1 bob bin 0 Mar 16 19:04 file-2

drwxrwxr-x. 2 bob bob 4096 Mar 16 19:04 temp





Click the image to view an enlarged version.