Search...

# UNIX Tutorials for Beginners

These tutorials are derived from the excellent tutorials from the University of Surrey, UK, with some minor modifications for our site. The originals can be found here.

Last updated on Apr 29, 2019

# Introduction to Unix

## Typographical Conventions

In what follows, we shall use the following typographical conventions:

- Characters written in color-coded typewriter font are commands to be typed into the computer as they stand.
- Characters written in standard typewriter font indicate non-specific file or directory names.
- Words inserted within square brackets, e.g. [Ctrl], indicate keys to be pressed.

So, for example:

```
% ls anydirectory [Enter]
```

means "at the Unix prompt `%`, type `ls` followed by the name of some directory, then press the key marked [Enter]." Don't forget to press the [Enter] key: commands are not sent to the computer until this is done.

Note: Unix is case-sensitive, so "LS" is not the same as `ls`. The same applies to filenames, so `myfile.txt`, `MyFile.txt` and `MYFILE.TXT` are three separate files. Beware if copying files to a PC, since DOS and Windows do not make this distinction.

## Introduction to the Unix Operating System

An *operating system* is the suite of programs that make the computer work. Historically, Unix had two "flavors," System V from AT&T's Bell Labs, and BSD (for Berkeley Software Distribution). Today that distinction is less relevant but the dominant versions of Unix still have different roots. *Linux*, used for some workstations and many high-performance computing clusters, arose from System V. Mac OS was derived from a version of BSD. Android is a heavily-modified Linux.

The "windowing" (graphical) system on Linux is called X11. Modern Linux systems all provide at least one "desktop" environment, similar to Windows or the Mac OS desktop. However, working at the command line is still often the most efficient use of teh system, so users can benefit by learning their way around a terminal.

A Unix operating system is made up of three parts: the kernel, the shell and the programs (applications or "apps").

### The Kernel

The kernel of Unix is the hub of the operating system: it allocates time and memory to programs and handles the filesystem and communications in response to system calls.

### The Shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt. The adept user can customise his/her own shell, and users can use different shells on the same machine. The `bash` shell is the default on Linux. Mac OS formerly used bash, but on newer releases the `zsh` shell is the default. Most of the basic commands are the same.

As an illustration of the way that the shell and the kernel work together, suppose a user types `rm myfile` to delete the file `myfile`. The shell searches the filesystem, first for the file containing the program `rm`, and directs the kernel, through system calls, to *execute* the program `rm` on myfile. When the process `rm myfile` has finished, the shell then returns the prompt to the user, indicating that it is ready for further commands.

### Some Features

- Filename Tab Completion: By typing part of the name of a command, filename or directory and pressing the [Tab] key, the shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it will halt, prompting you to type a few more letters before pressing the [Tab] key again.
    - Both bash and zsh support tab completion, but zsh has a somewhat more sophisticated set of features.
- History: The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type`history` for a list of previous commands.
    - Several discussions of the differences between bash and zsh are available online, such as here. The rest of these tutorials are targeted to bash.

### Prompt

The prompt is a character or string of characters that indicates the shell is ready for a command. It varies by shell and system, and can be customized by more advanced users. Throughout these tutorials we will us the percent sign `%` as the prompt.

NEXT
Files in Unix

Last updated on Jan 1, 0001

Search...

# Files in Unix

## Files and Processes

Everything in Unix is either a file or a process. A process is an executing program identified by a unique PID (process identifier). A file is a collection of data. They are created by users using text editors, running compilers etc. Examples of files:

- a document (report, essay etc.)
- the text of a program written in some high-level programming language
- instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file)
- a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files

### The Directory Structure

In Unix, folders are generally called *directories*. Directories are arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called `root`.

## Listing Files and Directories

### `ls` (list)

When you first log in, your current working directory is your *home directory*. Your home directory has the same name as your username, for example, `mst3k`, and it is where your personal files and subdirectories are saved. To find out what is in your home directory, type

```
% ls
```

This is short for “list.” Most Unix commands are two to four letters, sometimes with nonintuitive meanings.

The `ls` command lists the contents of your current working directory. There may be no files visible in your home directory, in which case the prompt will be returned. Alternatively, there may already be some files or folders created when your account was set up.

In most Unix systems, `ls` does not report *hidden files* by default. Files or directories with names beginning with a dot (.) are hidden and usually contain important program configuration information. They are hidden because you should not change them unless you understand what you are doing. To list all files in your home directory including those whose names begin with a dot, type

```
% ls -a
```

The -a is an example of a *command-line option*. The options change the behavior of the command. There are online manual pages that tell you which options a particular command can take, and how each option modifies the behavior of the command.

## Making Directories

### `mkdir` (make directory)

We will now make a subdirectory in your home directory to hold the files you will be creating and using in the course of this tutorial. To make a subdirectory called unixstuff in your current working directory type

```
% mkdir unixstuff
```

To see the directory you have just created, type

```
% ls
```

## Changing to a Different Directory

### `cd` (change directory)

The command `cd _directory_` means change the current working directory to "directory". The current working directory may be thought of as the directory you are in, i.e. your current position in the directory tree. To change to the directory you have just made, type

```
% cd unixstuff
```

Type `ls` to see the directory's contents (it should be empty).

### Exercise 1A

Make another directory inside the unixstuff directory called backups.

## The Directories `.` and `..`

While still in the unixstuff directory, type

```
% ls -a
```

The unixstuff directory (and in all other directories) contains two special directories called (.) and (..). In Unix, (.) means the current directory, so typing

```
% cd .
```

means stay where you are (the unixstuff directory). This may not seem very useful at first, but using (.) as the name of the current directory will save a lot of typing, as we shall see later in the tutorial. (..) means the parent of the current directory, so typing

```
% cd ..
```

will take you one directory up the hierarchy. Try it now.

Note: there is a space between cd and the dot or double dot. Also note: typing cd with no argument always returns you to your home directory. This is very useful if you are lost in the file system.

## Pathnames

### `pwd` (print working directory)

Pathnames enable you to work out where you are in relation to the whole filesystem. For example, to find out the absolute pathname of your home directory, type `cd` to get back to your home directory and then type

```
% pwd
```

Most of the time you should see `/home/mst3k`. On a multiuser central system like Rivanna, /home may be a *symbolic link*, i.e. an alias, for something else. To see it, type

```
% pwd -P
```

The full pathname may look something like this: /sfs/qumulo/qhome/mst3k.

### Exercise 1B

Use the commands `ls`, `pwd` and `cd` to explore the file system. (Remember, if you get lost, type cd with no argument to return to your home directory.)

## More About Home Directories and Pathnames

### Understanding Paths

First type cd to get back to your home directory, then type

```
% ls unixstuff
```

to list the contents of your unixstuff directory. Now type

```
% ls backups
```

You will get a message like this:

```
backups: No such file or directory
```

The reason is that "backups" is not in your current working directory. To use a command on a file (or directory) not in the current working directory (the directory you are currently in), you must either cd to the correct directory, or specify its full *pathname*. To list the contents of your backups directory, you must type

```
% ls unixstuff/backups
```

This path starts from the current location. A path may be *absolute* or *relative*. An absolute path starts from the root location. The absolute path is

```
ls /home/mst3k/unixstuff/backups
```

A relative path starts from the current working directory. The special symbols `.` and `..` are often used for relative paths.

### `~` (your home directory)

Home directories can also be referenced by the tilde `~` character. It can be used to specify paths starting at your home directory. So typing

```
% ls ~/unixstuff
```

will list the contents of your unixstuff directory, no matter where you currently are in the file system. What do you think

```
% ls ~
```

would list? What do you think

```
% ls ~/..
```

would list?

## Getting Help

### Online Manuals

There are online manuals which give information about most commands. The manual pages tell you which options a particular command can take, and how each option modifies the behaviour of the command. Type `man` command to read the manual page for a particular command. For example, to find out more about the wc (word count) command, type

```
% man wc
```

In newer Linux systems, most commands have a `--help` option

```
% wc --help
```

However, `man` sends output through a pager, whereas –help prints directly to the console.

Another useful command,

```
% whatis wc
```

gives a one-line description of the command, but omits any information about options, etc.

## Summary

| Command | Operation |
|---|---|
| `ls` | list files and directories |
| `ls -a` | list all files and directories |
| `mkdir` | make a directory |
| `cd directory` | change to directory |
| `cd` | change to home directory |
| `cd ~` | change to home directory |
| `cd ..` | change to parent directory |
| `pwd` | display the path of the current directory |
| `man` | display the manual pages for the specified command |
| `whatis` | display a description of the specified command |

**PREVIOUS**
Introduction to Unix

**NEXT**
Working with Files

Last updated on Jan 1, 0001

University of Virginia | Research Computing

# Working with Files

## Copying Files

### `cp` (copy)

`cp` file1 file2 is the command which makes a copy of file1 in the current working directory and calls it file2.

For our example we will create a file *science.txt*. Click the down-arrow icon to download the file. Use whatever method you know to place this file into your home directory.

```
In the space of one hundred and seventy-six years the Lower Mississippi has shortened itself

- Mark Twain, Life on the Mississippi
```

```
% cd ~/unixstuff
```

Then at the Unix prompt, type,

```
% cp ../science.txt .
```

Note: Don't forget the dot (.) at the end. Remember, in UNIX, the dot means the current directory.

The above command means copy the file science.txt from the parent directory to the current directory, keeping the name the same. To change the name, use

```
% cp ../science.txt ./newname
```

### Exercise 2A

Create a backup of your science.txt file by copying it to a file called science.bak.

## Moving Files

### `mv` (move)

`mv file1 file2` moves (or renames) `file1` to `file2`. To move a file from one place to another, use the `mv` command. This has the effect of moving rather than copying the file, so you end up with only one file rather than two. It can also be used to rename a file, by moving the file to the same directory, but giving it a different name. We are now going to move the file science.bak to your backup directory. First, change directories to your unixstuff directory (can you remember how?). Then, inside the unixstuff directory, type

```
% mv science.bak backups/.
```

Type `ls` and `ls backups` to see if it has worked.

## Removing Files and Directories

### `rm` (remove), `rmdir` (remove directory)

To delete (remove) a file, use the `rm` command. As an example, we are going to create a copy of the science.txt file then delete it. Inside your unixstuff directory, type

```
% cp science.txt tempfile.txt
```

Confirm the file was created:

```
% ls
```

Now delete it:

```
% rm tempfile.txt
% ls
```

You can use the `rmdir` command to remove a directory, but only if it is empty. Try to remove the backups directory. You will not be able to since Unix will not let you remove a non-empty directory.

To remove a non-empty directory use

```
rm -rf directory
```

⚠ The above command will remove the directory without confirming anything! Be extremely careful with it!

You can request confirmation with

```
% rm -if directory
```

though this may be tedious. The `-i` option (inquire) also works for `rm`

```
% rm -i myfile
```

### Exercise 2B

Create a directory called tempstuff using mkdir, then remove it using the rmdir command.

## Displaying the Contents of a File on the Screen

### `cat` (concatenate)

The `cat` command can show a text file's contents

```
% cat science.txt
```

Be sure to use the correct path to the file. Cat can also join two text files, hence its name.

```
% cat file1 file2 > file3
```

THe `>` sign is a *redirection*, which we will discuss later.

### `clear` (clear screen)

Clear the screen.

### `more`

The command `more` prints the contents of a file onto the screen a page at a time. Type

```
% more science.txt
```

Press the [spacebar] if you want to see another page. Type [q] if you want to quit reading.

The `more` command is an example of a **pager**, a program that "pages" through a text file.

### `head`

The head command writes the first ten lines of a file to the screen.

```
% head science.txt
```

Now type

```
% head -5 science.txt
```

What difference did the `-5` make to the `head` command?

### `tail`

The `tail` command writes the last ten lines of a file to the screen. Clear the screen and type

```
% tail science.txt
```

How can you view the last 15 lines of the file?

## Searching the Contents of a File

### Simple Searching Using `more`

Using `more`, you can search through a text file for a keyword (pattern). For example, to search through science.txt for the word 'science', type

```
% more science.txt
```

then, still in `more` (i.e. don't press [q] to quit), type a forward slash [`/`] followed by the word to search

```
/science
```

The `more` command finds and highlights the keyword. Type [n] to search for the next occurrence of the word.

### `grep` (don't ask why it is called grep)

`grep` is one of many standard Unix utilities. It searches files for specified words or patterns. First clear the screen, then type

```
% grep science science.txt
```

As you can see, grep has printed out each line containg the word science… or has it? Try typing

```
% grep Science science.txt
```

The `grep` command is case sensitive; it distinguishes between Science and science. To ignore upper/lower case distinctions, use the -i option, i.e. type

```
% grep -i science science.txt
```

To search for a phrase or pattern, you must enclose it in single quotes (the apostrophe symbol). For example to search for spinning top, type

```
% grep -i 'spinning top' science.txt
```

Some of the other options of `grep` are: `-v` (display those lines that do NOT match); `-n` (precede each maching line with the line number); and `-c` (print only the total count of matched lines). Try some of them and see the different results. Don't forget, you can use more than one option at a time, for example, the number of lines without the words `science` or `Science` is

```
% grep -ivc science science.txt
```

#### `wc` (word count)

A handy little utility is the wc command, short for word count. To do a word count on `science.txt`, type

```
% wc -w science.txt
```

To find out how many lines the file has, type

```
% wc -l science.txt
```

## Summary

| Command | Operation |
|---|---|
| cp file1 file2 | copy file1 and call it file2 |
| mv file1 file2 | move or rename file1 to file2 |
| rm file | remove file |
| rmdir directory | remove directory |
| cat file | display file |
| more file | display file a page at a time |
| head file | display the first few lines of a file |
| tail file | display the last few lines of file |
| grep 'keyword' file | search file for keywords |
| wc file` | count number of lines/words/characters in file |

PREVIOUS
Files in Unix

NEXT
More About Files

Last updated on Apr 29, 2019

# More About Files

## Redirection

Most processes initiated by Unix commands write to the *standard output* (that is, they write to the terminal screen), and many take their input from the *standard input* (that is, they read it from the keyboard). There is also the *standard error*, where processes write their error messages, by default to the terminal screen. Type `cat` without specifing a file to read

```
% cat
```

Then type a few words on the keyboard and press the [Return] key. Finally hold the [CTRL] key down and press `[d]` (written as `^D` for short) to end the input. What has happened? If you run the cat command without specifing a file to read, it reads the standard input (the keyboard), and on receiving the 'end of file' (`^D`), copies it to the standard output (the screen). In UNIX, we can redirect both the input and the output of commands.

## Redirecting the Output

We use the `>` symbol to redirect the output of a command. For example, to create a file called list1 containing a list of fruit, type

```
% cat > list1
```

Then type in the names of some fruit as follows. Press [Return] after each one. Terminate with the end-of-file marker control-d (`^D`).

```
pear
banana
apple
^D
```

The `cat` command reads the standard input (the keyboard) and the > redirects the output, which normally goes to the screen, into a file called list1. To read the contents of the file, type

```
% cat list1
```

## Exercise 3A

Using the above method, create another file called list2 containing the following fruit: orange, plum, mango, grapefruit. The form » *appends* standard output to a file. So to add more items to the file list1, type

```
% cat >> list1
```

Then type in the names of more fruit

```
peach
grape
strawberry
^D
```

To read the contents of the file, type

```
% cat list1
```

You should now have two files. One contains six fruit names, the other contains four fruits. We will now use the cat command to join (concatenate) list1 and list2 into a new file called biglist. Type

```
% cat list1 list2 > biglist
```

This reads the contents of list1 and list2 in turn, then outputs the text to the file biglist. To read the contents of the new file, type

```
% cat biglist
```

## Redirecting the Input

We use the `<` symbol to redirect the input of a command. The command sort alphabetically or numerically sorts a list. Type

```
% sort
```

Using `<` you can redirect the input to come from a file rather than the keyboard. For example, to sort the list of fruit, type

```
% sort < biglist
```

and the sorted list will be output to the screen. To output the sorted list to a file, type

```
% sort < biglist > slist
```

Use cat to read the contents of the file slist.

## Pipes

To see who is on the system with you, type

```
% who
```

One method to get a sorted list of names is to type

```
% who > names.txt
% sort < names.txt
```

This is a bit slow and you have to remember to remove the temporary file called names.txt when you have finished. What you really want to do is connect the output of the who command directly to the input of the sort command. This is exactly what pipes do. The symbol for a pipe is the vertical bar | which, on a US keyboard, is above Enter on the right, with the backslash. For example, typing

```
% who | sort
```

will give the same result as above, but quicker and cleaner. To find out how many users are logged on, use wc (word count) with the option -l (ell) for number of lines only:

```
% who | wc -l
```

## Exercise 3B

Using pipes, print all lines of list1 and list2 containing the letter 'p', sort the result, and print to a file sorted_plist.txt. Hint: from `grep --help` find an option to print only the line, omitting the file name.

▶ Solution

### Summary

| Command | Operation |
|---|---|
| `command >` file | redirect standard output to a file |
| `command >>` file | append standard output to a file |
| `command <` file | redirect standard input from a file |

| Command | Operation |
|---|---|
| `command1` | command2 |
| `cat file1 file2 >` file0 | concatenate file1 and file2 to file0 |
| `sort` | sort data |
| `who` | list users currently logged in |

PREVIOUS
Working with Files

NEXT
Wildcards and File Access

Last updated on Apr 29, 2019

Search...

# Wildcards and File Access

## Wildcards

### The Characters `*` and `?`

The character `*` is called a wildcard, and will match against none or more character(s) in a file (or directory) name. For example, in your `unixstuff` directory, type

```
% ls list*
```

This will list all files in the current directory starting with list. Try typing

```
% ls *list
```

This will list all files in the current directory ending with list. The character ? will match exactly one character. So `ls` ?ouse will match files like house and mouse, but not grouse. Try typing

```
% ls ?list
```

## Filename Conventions

Unix regards a directory as merely a special type of file. So the rules and conventions for naming files apply also to directories. In naming files, characters with special meanings, such as `/` `*` & `%`, should be avoided. Also avoid using spaces within names. The safest way to name a file is to use only alphanumeric characters, that is, letters and numbers, together with `_` (underscore) and `.` (dot). File names conventionally start with a lower-case letter, and may end with a dot followed by a group of letters indicating the contents of the file. For example, all files consisting of C code may be named with the ending `.c`, for example, `prog1.c`. Then in order to list all files containing C code in your home directory, you need only type `ls *.c` in that directory.

Beware: some applications give the same name to all the output files they generate. For example, some compilers, unless given the appropriate option, produce compiled files named a.out. Should you forget to use that option, you are advised to rename the compiled file immediately, otherwise the next such file will overwrite it and it will be lost.

The Unix system itself ignores file suffixes. You could name your C++ program `mycode.py` and Linux will not care. However, many applications do care about file extensions. A C++ compiler will expect the file to end in `.cpp` or `.cxx`. It will not recognize a file ending in `.py`, but a Python interpreter would.

## Filesystem Security (Access Rights)

In your unixstuff directory, type

```
% ls -l (l for long listing!)
```

You will see that you now get lots of details about the contents of your directory, similar to the example below.

```
-rw------- 1 mst3k users       340 Jan 24 09:41 contour.py
-rw-r--r-- 1 mst3k users       322 May 29  2020 omparea.c
-rw-r--r-- 1 mst3k users       336 May 29  2020 omparea.f90
-rwxr-xr-x 1 mst3k users    105104 Jan 30 09:25 ompheatedplate
-rw-r--r-- 1 mst3k users      3144 Jan 24 11:49 ompheatedplate.cxx
-rw-r--r-- 1 mst3k users      2147 Jan 25 07:31 ompheatedplate.f90
-rw-r--r-- 1 mst3k users       302 Jan 30 13:39 omp.slurm
-rw-r--r-- 1 mst3k users 197321360 Jan 30 13:22 plate5
drwxr-sr-x 2 mst3k users      2560 May 29  2020 pythonMP
-rw-r--r-- 1 mst3k users       270 Jan 30 13:39 serial.slurm
-rw-r--r-- 1 mst3k users       403 Jan 30 15:52 slurm-46361608.out
```

Each file (and directory) has associated access rights, which may be found by typing ls -l.

In the left-hand column is a 10-symbol string consisting of the symbols `d`, `r`, `w`, `x`, `-`, and, occasionally, `s` or `S`. If d is present, it will be at the left hand end of the string and indicates a directory; otherwise - will be the starting symbol of the string. The nine remaining symbols indicate the permissions, or access rights, and are taken as three groups of three.

- the leftmost group of 3 gives the file permissions for the user that owns the file (or directory) (mst3k in the above example);
- the middle group gives the permissions for the group of people to whom the file (or directory) belongs (users in the above example);
- the rightmost group gives the permissions for all others. The symbols r, w, etc., have slightly different meanings depending on whether they refer to a simple file or to a directory.

### Access Rights on Files

- `r` (or `-`), indicates read permission, that is, the presence or absence of permission to read and copy the file
- `w` (or `-`), indicates write permission, that is, the permission (or otherwise) to change a file
- `x` (or `-`), indicates execution permission, that is, the permission to execute a file, where appropriate

### Access Rights on Directories

- `r` allows users to list files in the directory
- `w` means that users may delete files from the directory or move files into it
- `x` means the right to access files in the directory or to cd into it.

So, in order to read a file, you must have execute permission on the directory containing that file, and hence on any directory containing that directory as a subdirectory, and so on, up the tree.

#### Some Examples

| Permissions | Meaning |
|---|---|
| `-rwxrwxrwx` | a file that everyone can read, write and execute (and delete) |
| `-rw-------` | a file that only the owner can read and write: no one else can read or write and no one has execution rights (e.g., yourmailbox file) |

## Changing Access Rights

### `chmod` (changing a file mode)

Only the owner of a file can use chmod to change the permissions of a file. The options of chmod are as follows:

| Symbol | Meaning |
|---|---|
| `u` | user |
| `g` | group |

| Symbol | Meaning |
|---|---|
| `o` | other |
| `a` | all |
| `r` | read |
| `w` | write (and delete) |
| `x` | execute (and access directory) |
| `+` | add permission |
| `-` | take away permission |

For example, to remove read write and execute permissions on the file biglist for the group and others, type

```
% chmod go-rwx biglist
```

This will leave the other permissions unaffected. To give read and write permissions on the file biglist to all,

```
% chmod a+rw biglist
```

### Exercise 4A

Try changing access permissions on the file science.txt and on the directory backups. Use ls -l to check that the permissions have changed.

## Shared Systems

Some multiuser systems, such as high-performance clusters, may not permit `chmod` on certain directories, or may automatically revert to the default set of permissions (called the *umask*). This is for data protection and privacy for users.

## Summary

| Command | Operation |
|---|---|
| * | match any number of characters |
| ? | match one character |
| chmod | change file or directory permissions |

PREVIOUS
More About Files

NEXT
Processes and Jobs

Last updated on Apr 29, 2019

Search...

# Processes and Jobs

A *process* is an executing program identified by a unique PID (process identifier). To see information about your processes, with their associated PID and status, type

```
% ps -u mst3k
```

A process may be in the foreground, in the background, or be suspended. In general the shell does not return the Unix prompt until the current process has finished executing. Some processes take a long time to run and hold up the terminal. Backgrounding a long process has the effect that the Unix prompt is returned immediately, and other tasks can be carried out while the original process continues executing.

## Running Background Processes

To background a process, type an & at the end of the command line. For example, the command sleep waits a given number of seconds before continuing. Type

```
% sleep 10
```

This will wait 10 seconds before returning the command prompt %. Until the command prompt is returned, you can do nothing in that terminal except wait. To run sleep in the background, type

```
% sleep 10 &

[1] 6259
```

The `&` runs the process in the background and returns the prompt immediately, allowing you to run other programs while waiting for that one to finish. The first line in the above example is typed in by the user; the next line, indicating *job number* and PID, is returned by the machine. The user is be notified of a job number (numbered from 1) enclosed in square brackets, together with a PID and is notified when a background process is finished. Backgrounding is useful for jobs which will take a long time to complete.

To the system, a “job” represents a group of processes (often just one) to which *signals* should be sent. Signals include backgrounding, forgrounding, and cancelling. This meaning of “job” must be distinguished from “jobs” submitted to a queueing system such as Slurm. To a resource manager (queueing system) a “job” is a set of instructions to be executed.

### Backgrounding a Current Foreground Process

At the prompt, type

```
% sleep 100
```

You can suspend the process running in the foreground by holding down the `[CTRL]` key and typing [`z`] (written as `^Z`) Then to put it in the background, type

```
% bg
```

Note: do not background programs that require user interaction.

### Listing Suspended and Background Processes

When a process is running, backgrounded or suspended, it will be entered onto a list along with a job number. To examine this list, type

```
% jobs
```

An example of a job list could be

1. Suspended sleep 100
2. Running firefox
3. Running vi

To restart (foreground) a suspended processes, type

```
% fg %jobnumber
```

For example, to restart sleep 100, type

```
% fg %1
```

Typing `fg` with no job number foregrounds the last suspended process.

## Killing a Process

### `kill` (terminate or signal a process)

It is sometimes necessary to kill a process (for example, when an executing program is in an infinite loop). To kill a job running in the foreground, type ^C ([CTRL] + [c]). For example, run

```
% sleep 100 ^C
```

To kill a suspended or background process, type

```
% kill %jobnumber
```

For example, run

```
% sleep 100 &

% jobs
```

If it is job number 4, type

```
% kill %4
```

To check whether this has worked, examine the job list again to see if the process has been removed.

### `ps` (process status)

Alternatively, processes can be killed by finding their process numbers (PIDs) and using `kill PID_number`:

```
% sleep 100 &
% ps

PID   TTY       TIME CMD
20077 pts/5 S   0:05 sleep 100
21563 pts/5 T   0:00 firefox
21873 pts/5 S   0:25 vi
```

To kill off the process `sleep 100`, type

```
% kill 20077
```

and then type ps again to see if it has been removed from the list. If a process refuses to be killed, uses the `-9` option, i.e., type

```
% kill -9 20077
```

Linux systems support a command `killall` which takes the name of the process rather than its PID.

```
killall -9 sleep
```

Note: It is not possible to kill off other users' processes! Unless, of course, it is a system you control and for which you have *root* privileges. The *root* account is the system administrator, and on Linux is all-powerful.

## Summary

| Command | Operation |
|---|---|
| `ls -lag` | list access rights for all files |
| `chmod [options] file` | change access rights for named file |
| `command &` | run command in background |
| `^C` | kill the job running in the foreground |
| `^Z` | suspend the job running in the foreground |
| `bg` | background the suspended job |
| `jobs` | list current jobs |
| `fg %1` | foreground job number `1` |
| `kill %1` | kill job number `1` |
| `ps` | list current processes |
| `kill 26152` | kill process number `26152` |
| `killall name` | kill process `name` |

**PREVIOUS**
Wildcards and File Access

**NEXT**
Other Useful Commands

Last updated on Apr 29, 2019

Search...

# Other Useful Commands

## `exit`

Exit the current shell. If it is the login shell, this command logs the user off.

## `which`

The `which` command indicates the path to the executable specified.

```
% which myexec
```

The which command returns the location of the executable according to the rules used to search paths. The shell always searches from left to right in the list contained in the PATH environment variable; the first executable of the specified name is the one that is used.

## `wc`

The `wc` command returns the number of lines, words, and characters in an ASCII file. A word is defined as a non-zero length string surrounded by whitespace.

```
% wc myfile.txt
```

To print only the number of lines, use

```
% wc −l  myfile.txt
```

## `diff`

`diff` shows the differences between two ASCII files on a per-line basis.

```
% diff file1 file2
```

## `find`

`find` is an extremely powerful command with many options. The simplest and most common use of it is to search for a file of a given name or with a name containing a pattern.

```
% find . -name myscript.sh
```

This starts from current directory (`.`) and searches for myscript.sh. The period is optional under Linux (but not under Mac OSX). To search for a name with a pattern it must typically be enclosed in quotes

```
% find . -name "*.sh"
```

See the manpage or examples online for more usage patterns of this command.

## `du`

The `du` command outputs the number of kilobyes used by each subdirectory. Useful if you have gone over quota and you want to find out which directory has the most files. Some options make it more useful; in particular, -s summarizes directories and -h prints it in human-readable format. In your home directory, type

```
% du -s -h *
```

## `gzip` and `zip`

This reduces the size of a file, thus freeing valuable disk space. For example, type

```
% ls -l science.txt
```

and note the size of the file. Then to compress `science.txt`, type

```
% gzip science.txt
```

This will compress the file and place it in a file called science.txt.gz. To see the change in size, type ls -l again. To uncompress the file, use the gunzip command.

```
% gunzip science.txt.gz
```

Most Linux systems also provide the standard `zip` and `unzip` commands.

```
% zip science.txt.zip
% unzip science.txt.zip
```

## tar (tape archive)

The standard archive format in Linux is `tar`. Tar is usually used to bundle directories. (Zip can also be used for this purpose.) Typically the output file is compressed with gzip.

```
tar czf mydir.tar.gz mydir
```

The `c` option creates the tarfile (also called a “tarball”). The `f` option is for file, and this form of the command must be followed by the name of the file to contain the archive. The `z` option gzips the file.

To extract the files

```
tar xf mydir.tar.gz
```

Newer versions of `tar` can detect that the archive is zipped, so a `z` is not necessary for extraction. The `x` option extracts. This will create the directory `mydir` if it does not exist. If it does, the contents will be replaced by the contents of mydir.tar.gz.

## `file`

`file` classifies the named files according to the type of data they contain, for example ASCII (text), pictures, compressed data, etc. To report on all files in your home directory, type

```
% file *
```

## `cut`

The cut command extracts selected portions of a line, based on fields separated by a delimiter

```
% cut?d delim ?fC1,C2,C3
```

Examples:

```
% cut -d ' ' ?f1 /etc/resolve.conf
% cat myfile | cut -c 80
```

## `sort`

This command sorts lines of a text file, based on command-line options. The default is to sort alphabetically, based on lexigraphical ordering (in which e.g. 100 comes before 2).

```
% sort mylist.txt
```

## `uniq`

Removes duplicate lines (file must be sorted first since it only compares lines pairwise).

```
% uniq mylist.txt
```

A frequent pattern is to pipe the output of sort into `uniq`

```
% sort animals | uniq
```

## `history`

The bash shell keeps an ordered list of all the commands that you have entered. Each command is given a number according to the order it was entered.

```
% history (show command history list)
```

If you are using the `bash` or `tcsh` shell, you can use the exclamation character (`!`) to recall commands easily.

```
% !! (recall last command)
% !-3 (recall third most recent command)
% !5 (recall 5th command in list)
% !grep (recall last command starting with grep)
```

You can increase the size of the history buffer by typing

```
% set history=100
```

**PREVIOUS**

Processes and Jobs

**NEXT**

Environment Variables

Last updated on Apr 29, 2019

Search...

# Environment Variables

Variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for particular variables and if they are found will use the values stored. Some are set by the system, others by you, yet others by the shell, or any program that loads another program. Standard Unix variables are split into two categories, *environment variables* and *shell variables*. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions. Environment variables are exported and have a farther reaching significance; those set at login are valid for the duration of the session. By convention, environment variables are written in UPPERCASE while shell variables usually have lowercase names.

## Environment Variables

An example of an environment variable is the `$SHELL` variable. The value of this is the current shell you are using. Type

```
% echo $SHELL
```

More examples of environment variables are

```
$USER (your login name)
$HOME (the path name of your home directory)
$PWD (current working directory)
$DISPLAY (the name of the computer screen to display X windows; only set if X is enabled)
$PATH (the directories the shell should search to find a command)
```

## Using and Setting Variables

Environment variables are set using the `export` command (`bash` or `zsh`) or the `setenv` command (`tcsh` or `csh`), displayed using the `printenv` (`bash`, `tcsh`) or `env` (`bash`, `zsh`) commands, and unset using the `unsetenv` command. To show all values of these variables, type

```
% printenv | more
```

To set a value of an environment variable, type (for `bash`)

```
% export VAR=value
```

## Sourcing

A group of shell commands can be placed into a file and then *sourced*. When a file is sourced, the commands are executed as if they had been typed at the command line in the current shell. For example, if several environment variables needed to be set over and over again, they could be collected into a file such as this simple script called `envs.sh`:

```
export NUM_CPUS=16
export MEM=64

ncpus=${NUM_CPUS}
```

## Exercise 6A

Download the `envs.sh` file to the Unix system you are using. Run the command

```
source envs.sh
```

Print the values of the environment variables in the file. To print the value of the shell variable ncpus, type

```
echo $ncpus
```

## Dotfiles

Each time you log in to a Unix host, the system looks in your home directory for initialization files. Information in these files is used to set up your working environment. The first initialization file sourced is the *login* initialization. It is sourced only in the login shell. Note: modern “desktop” user interfaces tend to “swallow” the login setup file, and it may be difficult to determine what is happening in these cases if there is an error.

At login the bash shell first sources .bash_profile or .profile (if .bash_profile exists .profile will be ignored). Child shells source .bashrc. The `zsh` sources `.zprofile` and child shells source `.zshrc`. Two older shells, csh (C shell) and tcsh, read `.login` for login shells and `.cshrc` or `.tcshrc` for all other shells.

Note that all these file names begin with periods or “dots”; hence they are called *dotfiles*. As we have learned, dotfiles are hidden and will only be visible with `ls -a`.

The `.bash_profile`, `.profile`, or `.login` is to set conditions which will apply to the whole session and/or to perform actions that are relevant only at login. The `.bashrc`, `.zshrc`, or `.tcshrc` file is used to set conditions and perform actions specific to the shell and to each invocation of it. The rc stands for resource; many Unix dotfiles use this convention to set resources.

If you wish for your login shell to source the .bashrc also, add the lines

```
if [ -f ~/.bashrc ];
  then . ~/.bashrc
fi
```

to the `.bash_profile` script.

Warning: NEVER put commands that run graphical displays (e.g. web browsers) in your dotfiles. If you change your `.bashrc` you can force the shell to reread it by using the shell source command.

```
% source ~/.bashrc
```

## Setting the Path

When you type a command, your path (or `$PATH`) variable defines in which directories the shell will look to find the command you typed. If the system returns a message saying “`command: Command not found`”, this indicates that either the command doesn’t exist at all on the system or it is simply not in your path.

For example, suppose you have installed a program called “units” into your home directory in a folder called `units174`. Units is a simple utility that can convert Imperial to metric and vice versa, from SI to cgi, and so forth. This folder contains a `bin` subdirectory in which the executable is located. To run units, you must either directly specify the units path (`~/units174/bin/units`), or you need to have the directory `~/units174/bin` in your path. You can add it to the end of your existing path (the `$PATH` represents this) by issuing the command:

```
% export PATH=$PATH:$HOME/units174/bin
```

If you have `units` installed you can test that this worked by trying to run units in any directory other than where units is actually located.

```
% cd; units
```

Hint: You can run multiple commands on one line by separating them with a semicolon.

To add this path permanently, add the `export` line to your `.bashrc` file after the list of other commands. Make sure that you include the `$PATH` when you reset it, or you will lose access to basic system commands!

**PREVIOUS**

Other Useful Commands

Last updated on Apr 29, 2019