UNIX/Linux Tutorial

CSC a57

This document is meant to be a crash-course in using the UNIX operating system, with Linux being an example. It is not meant to be an in-depth reference. Beginners are taken through the system concepts and commands. If you're an experienced user, this is probably not for you, though you may find *some* new tricks.

For further information on the Linux operating system, I highly recommend the following:

- *Running Linux* by Matt Welsh and Lar Kaufman. This book takes you through the setup, management and use of Linux.
- *Linux in a Nutshell* by Jessica Perry Hekman. This is a good quick reference for commands and environment settings.

Now for some conventions. The `terminal` type face will be used to indicate things you type in at the system prompt, or for things the computer types to the screen, such as a directory listing. When it comes time to type a command, things in `< >` represent buttons to press. For example `<Ctrl-C>` means pressing down the Control button and the C button at the same time. Things between `[ ]` indicate information you must supply, such as `delete [filename]` The **bold** type face will be used to indicate terms which are jargon, and are therefore used in the literature (and the rest of this manuscript) without explanation. Now that that's out of the way, let's dive in.

Multi-user vs. Single-user

The UNIX family of operating systems are *multi-user* environments. This is in contrast to *single-user* systems, like DOS and Windows. When you run DOS or Windows on your PC, all files are readable and writable (i.e., deletable) by the user. There is nothing (except a warning message) to stop you from executing

```
C:> FORMAT C:
```

and wiping out your hard disk. In the UNIX world, however, there are many users on the same system. Consequently, each file or directory has an ownership attached to it. Your home directory, for instance, will be owned by you, as well as any files you create. Each file and directory also has permissions associated with it. For example, as a regular user, you can read and execute certain programs owned by the system, but you cannot write over them. You won't be able to read files in someone else's mailbox, either (which only makes sense). You can always change the permissions of any file you own. We will see this later on.

Now, if the system has multiple users, we have to have some way of telling it who we are. We also have to make sure someone can't impersonate someone else and get access to their files. This is accomplished through the **login** procedure.

Logging In

The system administrator will have given you a **user name** and *initial* password. Different systems have different ways of presenting the login prompt: it could simply be text on the screen or a very posh graphical interface. Nevertheless, you usually enter your user name, followed by your password. Usually, the characters you type in for your password will not show up on the screen, for added security. If, for some reason, you can't log in, consult the system administrator. Also keep in mind that passwords are case sensitive.

At this point, depending on how the system administrator has set things up, you may be presented with a windows-like (usually X-windows) environment or a simple terminal. Either way, you should now have a screen where you can type in commands. This is your **shell**. Your shell is your interface to the operating system (in DOS, your shell was called `COMMAND.COM`). Since UNIX is a multi-tasking as well as multi-user environment, you can have many shells running at the same time. To further complicate things, there are several different shells you can use. The default shell on Linux is `bash` (stands for Bourne Again SHell). Others include `csh, tcsh, ksh and sh`. They all serve the same purpose, but each has different features and syntaxes. As far as this tutorial is concerned, they more or less work the same, however I will assume you are using `bash`.

Important: When you are all done, remember to **log out** of the system. This is usually accomplished by typing `logout` at the prompt. If you are using X-Windows, there will be a large button at the bottom of the screen labeled 'Logout'. You will know you have logged out completely when you see the login prompt. If you forget to log out, someone else could come along and mess with your files, or send email from your account!

Executing commands

Now it's time to execute our first command. You logged in with an initial password, so the first thing to do is change it. It's a good idea to change your password on a regular basis, for increased security. A password should be something hard to guess and should consist of both uppercase and lowercase letters and numbers. To change your password, execute the `passwd` command. The system will then prompt you for the old password and a new password.

Now that everything is nice and secure, let's have a look at what UNIX and `bash` have to offer us as an environment. When you log in to the system, you are usually deposited in your **home directory**. As we said above, this is a directory that you own and which you can write to. To find out where this directory is on the system, try using the `pwd` command. This will output the directory you are currently sitting in. Here's an example for my account on a computer called `mandragona`:

```
mandragona ~ $ pwd
/u/burns
mandragona ~ $
```

Let's have a look close look at this. The first few items on the command line, `mandragona ~ $` are supplied by the system. This is my system **prompt**. It's the computer's way of telling me it's ready to receive commands. Here, I've customized my prompt to indicate what machine I'm currently logged on (very useful when you have many login sessions on many computers on the same screen!), as well as my current directory (the ~ indicates my home directory). The second line is what the computer spits out. The third line is again my system prompt.

The command `pwd` and `passwd` are relatively simple commands in that we didn't supply any arguments. Most commands in UNIX need arguments to function. Keeping track of what arguments a command requires takes a very good memory, especially for seldom-used commands or features of a command. That's why UNIX has an on-line summary of each command, called **man pages**. To access the man page for a particular command, just execute: `man [command]`. For example:

```
mandragona ~ $ man pwd
```

produces:

```
PWD(1L)                                                  PWD(1L)
```

```
NAME
pwd - print name of current/working directory

SYNOPSIS
pwd
pwd {--help,--version}

DESCRIPTION
This  manual  page  documents the GNU version of pwd.  pwd
prints the fully resolved name of the  current  directory.
That is, all components of the printed name will be actual
directory names -- none will be symbolic links.

Note that most Unix shells provide a built-in pwd  command
with  similar  functionality so the unadorned, interactive
pwd command will usually execute the built-in version  and
not this one.

OPTIONS
--help Print  a  usage message on standard output and exit
      successfully.

--version
      Print version information on standard  output  then
      exit successfully.

FSF                    GNU Shell Utilities                     1
```

While very helpful, these man pages use a lot of jargon and at first you might feel they confuse more than illuminate. However, the more you learn about UNIX, the more they will make sense. If you feel very ambitious, try

```
 $ man bash
```

which gives a very lengthy and technical guide to the `bash` shell.

The Filesystem

Like DOS/Windows, UNIX can have several filesystems on one or more disks. In DOS/Windows, different **partitions** are labeled as `A: B: C: D: E:`etc. They might be hard disks, floppy disks, CD-ROMs, etc. Under each partition, there is a **directory tree** , consisting of many files and directories. Directories are specified using the backslash (\). So, for example, a file on a CD-ROM can be accessed by specifying:

```
D:\SUBDIR1\SUBDIR2\FILE.EXT
```

Things in UNIX are a bit different. We still have directory trees under partitions. However, in UNIX, everything is case-sensitive and file and directory names can have more than an 8-character name and 3-character extension. We are now free to have as many extensions as we like, each consisting of as many characters as we like. Another difference is that subdirectories are separated by forward slashes (/). Finally, different partitions look like subdirectories of the **root partition**. In our example above, we would specify a file on the CD-ROM as:

```
/cdrom/subdir1/subdir2/file.ext
```

Here, the CD-ROM is **mounted** on the subdirectory `/cdrom` . We access it just as if it were a regular file on our hard disk. The same would be true for other hard disks, floppy disks, NFS filesystems, etc. Use the `mount` command to see what partition is mounted under what subdirectory.

We said above that each file and directory has an ownership and permissions attached to it. To see what they are, you use the `ls` command. `ls` provides the same information as `dir` in DOS. However, to make `ls` show us more than just the file names, we need to give it a `-l` **switch**. A switch is simply an argument which modifies the behaviour of a command. Switches usually consist of a dash (-), followed by one or more characters. If you execute the following statement:

```
$ ls -l .bash_profile
```

you should get something like:

```
-rw-r--r--   burns   users   512 .bash_profile
```

The first field describes the permissions of the file. The second tells us that `burns` owns the file. The third says that the group `users` has access to the file (more on groups in a bit). The fourth and fifth fields are the size of the file in bytes and the name of the file. Note that using `ls` by itself will list all files in the directory *except* those which begin with a period (.). These are hidden files. Use the `-a` switch to make `ls` list all files, including hidden ones.

Let's take a look at the permissions. This field consists of 10 characters. The first is reserved for special file attributes, such as "d" to indicate the file is a directory. The next three characters are the permissions associated with the owner of the file. In the above example, `rw-` means the owner of the file can read, write, but not execute the file. If we wanted the owner to be able to execute the file, we could use the `chmod` command (more about this later). If we used this command, the permissions of the owner would then be `rwx`.

The next three characters in the permissions field work the same as for the first three, but they apply to users who are in the group specified by the `ls -l` command. In the case above, this would be the group `users`. It is possible for one person to be a member of many groups. To find out which group(s) you belong to, simply invoke the `groups` command. Having groups as well as simple users can be very useful. Many users may be working on the same bit of code, for example, so they all need read/write access to the source code while making sure no one else can touch it.

The last three permission characters work the same as the owner and group ones, but these apply to *every* user on the system. Consider the command `ls`. If only one person on the system could execute this command, it would make things very difficult for other users. Also, we wouldn't want anyone to mistakenly delete the `ls` command (in UNIX, there is generally no undelete!). Therefore we can expect that the executable file `ls` will have permissions as follows:

```
-rwxr-xr-x   root   bin   45984 /bin/ls
```

Here, only one user, `root`, can write (i.e., modify or delete) the file. Everyone else can read and execute it. The special user `root` has ownership of most of the system files and is the account used by the system administrator. With a properly maintained system, it is therefore impossible for a regular user to inadvertently destroy important system files.

User Commands and Environment Variables

Now that we have seen what kind of strange land we've been deposited in, let's get to know the language. Your shell is your interface between you and the system. It takes what you type in, tries to figure out what you want to do and then sends messages to the system (or error messages to you, telling you how stupid you are). As stated previously, commands consist of the command name, followed by arguments. The arguments can either be information you provide or switches the command recognizes. Based on this information, the command goes about doing its stuff. However, there is another way a command can get information on how to behave, apart from the command line: environment variables.

Environment variables are just like variables in the sense of programs. In `bash`, you set an environment variable using the following syntax:

```
$ [VARIABLE]=[value]
$ export [VARIABLE]
```

The first line sets the variable. The variable name can be anything you like (just be careful you're not clobbering a previously defined one). The value can be any string. If it contains spaces, quote the whole thing. If we left it at that, we could access this variable as `$[VARIABLE]` later on. For example:

```
$ X=.bash_profile      ;#  No spaces before or after the =
$ echo $X
.bash_profile
$ ls -l $X
 -rw-r--r--   burns   users   512  .bash_profile
```

But we can only access this variable in our current shell. Think of our shell as a subroutine and the environment variable as a local variable. If we run a program, that program will not be able to access our "local" variable. To make it global, we use the `export` command. Now any program we run will be able to access `$X`.

As an example of how environment variables and commands work together, consider the `lpr` command. This sends a file to a printer. Since we are most likely set up on a network, there can be many printers to choose from. We select the appropriate printer using a switch:

```
 $ lpr -P[printer name]
```

Note that the switch is a *capital* P and there is no space between the switch and the printer name. It might get tedious to type out the printer name over and over again, especially if you usually only use one printer. Another way to do it is to use the `PRINTER` environment variable. `lpr` checks this environment variable and uses the printer name stored in it unless it is overridden by the `-P` switch. To see if you already have a `PRINTER` variable set, use the `echo` command: `echo $PRINTER` . Another way to check environment variables is to execute the `env` command. This will list all environment variables in your current shell.

If you want an environment variable to be set every time you log in automatically, you need only edit your .bash_profile file and put the variable assignment statement somewhere in the file on a line by itself. .bash_profile is like `AUTOEXEC.BAT` in DOS. All the lines in .bash_profile, except those beginning with a '#', are executed at login time.

Now let's look at the most common user commands and what they do. The easiest way to do this is to give you a table containing the name, function and possible DOS equivalents. It is up to you to look at the man pages for the commands to see what more they can do for you.

Command	**Function**	**DOS Equivalent**
`bash`	Run the Bourne Again Shell.	COMMAND.COM
`cc`	C compiler. This is available on most systems. There are many switches to modify the behaviour of cc. See the man pages for more details.	`GCC`
`cd [directory]`	Change to directory [directory].	`CD`
`chfn`	Change User Information	
`chmod [args] [file]`	Change permissions of [file]. [args] consists of 3 characters. The first is u, g or `o (user, group or other)`. The second is + or -. The third is `r, w` or `x` (read, write or execute).	`ATTRIB`

`chown [user].[group] [file]`	Change ownership of [file] to the user [user] and group [group].	
`clear`	Clears the terminal	`CLS`
`cp [file1] [file2]`	Copy [file1] to [file2]	`COPY`
`df`	Prints out the amount of free disk space on each partition currently mounted	`chkdsk`
`diff [file1] [file2]`	Print out the differences between [file1] and [file2]	`COMP`
`echo`	Prints out its arguments, doing variable substitution.	`ECHO`
`env`	Prints out environment variables	`SET`
`expr [formula]`	Performs numerical computation, using numbers, environment variables and operators. See man pages for syntax.	
`find [path] [conditions]`	Tool to locate files below the [path] subdirectory tree. There are many [conditions], including file names, dates, sizes, etc.	
`finger [user]`	Display information about [user]. Can be a user on the current machine, or on another machine, using the user@hostname syntax	
`grep [regexp] [file(s)]`	grep searches the names [files] for occurrences of [regexp], printing each line containing it.	
`gzip [file]`	Compress [file], renaming it [file].gz.	`PKZIP`
`less [filename]`	Print out [filename] one page at a time, prompting the user at the end of each page.	`MORE`
`lpr`	Send files to a printer	`PRINT`
`ls [options]`	List the contents of a directory. Lots of switches to control information printed out.	`DIR`
`logout`	Log out of the current shell. Usually `exit` will do the same thing.	`EXIT`
`mail`	Interactive mail tool. See man pages	
`man [command]`	Print out man pages on specified [command].	`HELP`
`mkdir [name]`	Make the subdirectory [name]	`MD`
`passwd`	Change your password.	
`pwd`	Show current directory.	
`rlogin [host]`	Log into a remote [host]	`TELNET`
`rm [file]`	Remove (delete) a [file]. CAUTION: Once deleted, a file is gone for good (unless it is backed up somewhere). There is no undelete command.	`ERASE`
`rmdir [directory]`	Remove (delete) a [directory].	`RMDIR`
`sh`	The Bourne Shell. This is the standard shell, of which `bash` is a derivative.	`COMMAND.COM`
`sort`	A very useful utility for sorting files. See man pages for all the options.	
`tcsh`	A derivative of the `csh` shell	`COMMAND.COM`
`telnet [host]`	TELNET to a remote [host]. Similar to `rlogin`.	`TELNET`
`top`	Displays the most CPU-intensive jobs currently running. Type 'q' to exit.	

`vi`	The Visual Editor. This is the standard editor in UNIX (i.e., it is guaranteed to be available). Emacs is the other commonly used editor. See next section on using vi.	`EDIT`
`w`	Displays information about the system, including all currently logged in users.	
`wc [file]`	Word count. Prints the number of characters, words and lines in [file].	
`write [user]`	Initiate a `write` session to [user]. If [user] is logged in and is accepting messages, whatever you write to your terminal after this command will appear on [user]'s terminal. Type **Ctrl-D** to terminate the session.	

There are many other commands besides the ones mentioned above. If you need to get something done that seems non-standard, before you go and write a C program to do it, poke around and make sure there isn't a command that will do the same thing!

Job Control

In order for UNIX to be multi-user, it must also be multi-tasking (otherwise people would have to take turns). Multi-tasking simply means that many processes, or **jobs** are running on the system at the same time. Each process is getting part of the processor's time. Jobs that you run on the system can either be run in the foreground or the background. When you run something in the foreground, control of the terminal will be restored only when the job is finished. As an example, try running the following command:

```
find / -name "ls" -print -xdev > stdout.txt 2> stderr.txt
```

This command starts searching for a file with the name "ls" in the root directory (/) and all its subdirectories, printing the result to the terminal. The `-xdev` switch prevents the command from searching on other partitions except the one it starts on. The extra bit on the end (`> output.txt 2> errors.txt`) tells the shell to **redirect** the regular output and error messages to a couple of files instead of sending them to the terminal (we'll see more about redirection later). As you can imagine, this command could take quite a long time to execute. You might wish to enter more commands while you wait. You can do this by running the job in the **background**. To do this, simply append an ampersand (&) after the command. Using our previous example:

```
$ find / -name "ls" -print -xdev > /dev/null 2> /dev/null &
[1] 4564
$
```

Control is returned to you almost immediately. In addition, the shell returns two numbers, one in square brackets. The one in square brackets is the **job number** and the other one is the **process id** number. The shell will notify you when a job has ended. To see what jobs you are currently running in the background, simply use the command `jobs`. If you wish to stop a job that is running in the background, you must send it a **signal**. This is done using the `kill` command. To send a request to stop, first try `kill -1 [process]`. You specify the process by its process id number. The `-1` switch indicates the signal you wish to send (in this case `SIGHUP`, which is a rather gentle kick). Some jobs may be a bit stubborn about quitting, in which case you can try `kill -9 [process]` (which will send the more forceful `SIGKILL` signal).

If you don't remember what id number your process has, simply use the `ps` command, which gives you information about your running jobs, including their process id numbers (PID). `top` is also a useful command: it lists the most CPU-intensive jobs currently running. Hit 'q' to exit from `top`.

If you've started a job without using the ampersand (&), you can still put it in the background. To do this, first you **suspend** the job by hitting < CTRL-Z >. This suspends the job and returns control to you. You then type the `bg` command, which will let the job resume in the background.

If you have written a lengthy bit of code (which you **will** be doing later on) that will take a long time to run, you can run it in the background and then logout of the system. Your job will continue to run without you. However, if it is a particularly intensive job, you will likely inconvenience other users, since the system will be more sluggish. To get around this, you use the `nice` command. This tells the system that your job should have less priority. Simply type `nice -#` followed by your command. `#` indicates the priority level you wish to assign to your job. On Linux, `#` ranges from 0 (highest priority) to 19 (lowest priority).

I/O Redirection

Most programs and commands produce information that is written to the terminal. They can also issue warnings or error messages as well, which are also written to the terminal. But suppose you would like to save this information to a file instead of reading it as it is printed. This could be useful if the command produces more than one screenfull of output. Your shell gives you a way of **redirecting** the output of a command to a file. Further, you might not wish to save the error messages to the same file (or any file at all). This is also possible.

The first thing to realize is that although the standard output and error messages are both being written to the screen, they are not the same output **stream**; therefore it is possible to differentiate them. We've seen a glimpse of this already with the `find` command:

```
find / -name "ls" -print -xdev > stdout.txt 2> stderr.txt
```

As you can see, to redirect the standard output of a command to a file, we use the `>` symbol. To redirect the standard error stream, we use the `2>` symbol. If you want to get rid of a stream completely (i.e., you don't want it printed to the terminal or put in a file), simply redirect the stream to the special file `/dev/null`, which is like a black hole for data.

While output redirection is very nice, UNIX also allows for **input redirection** as well. Consider the command `wc`. If we create a simple file (here we get to use output redirection):

```
$ echo "This is a simple text file" > simple.txt
```

we can now count how many characters, words and lines this file has in two different ways:

```
$ wc simple.txt
      1       6      27 simple.txt
$ wc < simple.txt
      1       6      27
```

For the first command, I gave the name of the file explicitly as an argument to `wc`. In the second command I redirected the contents of the file to `wc`'s **standard input**. The result is the same, except that in the first case, `wc` knows the name of the file (and gives it as output), whereas in the second case, it does not. Many commands will allow you to do this (e.g., `sort`).

'So what?', you might ask. By itself, this does not seem like such a great thing. However, it gives us considerable power when you want to do complicated things. Suppose (for some reason) I wanted to know how many files there are in the directory `/usr/bin` which contain the string 'ls' in their names. I *could* go and count them by hand. Or I could use the commands which are already available to me. However, there is no one program which will do this. But if I could somehow redirect the output of one command to the input of another, maybe several small programs can do the job together. The command `ls` will list all files. The command `grep [string]` will only print out lines which contain `[string]` and the command `wc` will count up lines. So, we can do the following:

```
$ ls /usr/bin | grep ls | wc
      6       6     106
```

Here, the pipe symbol (|) takes the output of the precedeing command and sends it to the input of the next command. We have executed 3 commands to handle one problem, which UNIX people call making a **pipeline**. Real UNIX gurus delight in making *huge* pipelines with complicated output redirection to perform amazing tasks.

The vi Editor

For those of us who learned UNIX on a more "primitive" system, `vi` was the editor of choice (it was the **only** choice). Now there are hosts of different editors, including `emacs`, `joe` and `textedit`. I've tried all these editors to some extent. They all have their pros and cons. `Emacs` is probably the most popular editor out there. Despite all this, I still prefer to use `vi` (or `vim` on Linux), because it's guaranteed to be available, it's small and you can do everything from your keyboard (no menus which require a mouse, for example). However, it's a bit non-intuitive for those who are used to the DOS/Windows type of editor. For more information on the other editors, refer to their man pages.

Vi has essentially 2 modes. When you enter vi, you are usually in the **command mode**. In this mode, you can enter commands, move around, or enter one of the **input modes**. You can't start typing away, as you might expect, because in command mode, the letters you type are actually commands. For instance, typing `dd` will delete one entire line of text.

To move around in command mode, you can usually use the cursor keys the way you would expect. If this doesn't work, you can fall back on the `h, j, k` and `l` keys. Use h and l to go left and right, respectively. Use k and j to move up and down, respectively. Here are some more movements you can do in command mode:

```
h,j,k,l         Left, up, down, right one character
w, b            Forward, back one word
), (            Go to beginning of next, current sentence
0, $            Go to beginning, end of the current line
^               First non-blank character
Ctrl-F, Ctrl-B  Scroll forward, backward one screen
Ctrl-D, Ctrl-U  Scroll forward, backward half a screen
```

When you reach the spot you want to start making changes, you will then usually enter one of the input modes. By typing 'i', for instance, you enter insert mode. From now on, whatever you type will be inserted at the current cursor position. To exit any of the input modes, hit Esc, and you will be back in command mode. Here are some more of the available input modes:

```
a               Append after the cursor position
A               Append at the end of the line
i               Insert before cursor
I               Insert at beginning of line
o               Open a line below cursor
O               Open a line above cursor
R               Begin overwriting at cursor position.
```

Some commands require arguments, so simple one-key commands won't suffice. Therefore, we have what are called **colon-commands**. They consist of a colon (:), followed by the command and arguments. For example, to read in a file called `dummy` and insert its contents after the cursor, you would type in `:r dummy`. Here are some other commands:

```
:#              Go to line #
:e [file]       Edit [file]
:help           Get help (only in vim)
:r [file]       Read in [file] and insert it at the cursor
:w [file]       Write to [file].  :w saves to the current file
:wq             Save the current file and exit
:q!             Do not save the current file and exit
```

That should give you enough to get going. There are a lot more commands available, such as search and replace.

Putting it all Together

By now you should be proficient at logging in and executing a few commands. Now we're going to look at a specific case which you will no doubt find useful: editing and compiling a C program. Begin by logging in to your account. It's a good idea to have a seperate directory for each of your projects. I usually have a directory called `src` for all my source code, with subdirectories below it for each project. So, make a directory for your source code. For example:

```
~ $ mkdir src
~ $ cd src
~/src $ mkdir first
~/src $ cd first
~/src/first $
```

Now use your favorite text editor to enter the following bit of code.

```
/*  A simple program which prints out a message  */

#include <stdio.h>

main()
{
   printf("Goodbye World!\n");
}
```

Save the file as `first.c` or something similar. If we now list the files in our directory, we should get something like:

```
~/src/first $ ls
first.c
~/src/first $
```

Now, we shall compile this bit of code. To do this, you run the C compiler:

```
 ~/src/first $ cc -o first first.c
```

The `-o` switch tells the compiler to name the executable file `first` instead of the default `a.out`. Now you will have three files in your directory: `first.c, first.o` and `first`. The file `first.o` is the object file which is created and then linked with the standard libraries to produce the executable file `first`. To see if the program works, just type `./first` in the current directory (the `./` forces bash to search for the command `first` in the current directory).

Let's suppose you really like the executable file you've just produced and would like to be able to run the file from no matter what directory you are currently in. All you have to do is make sure that the file is somewhere in your **path**. Your path is a list of directories where the shell will look for executable files. To see what your path is currently set to, just type `echo $PATH`. The elements in the list are seperated by colons (:). You can add a directory to the path by typing:

```
 $ export PATH="$PATH:[new directory]"
```

I like to keep all my finished executable files in a directory called `bin` below my home directory. I then add `~/bin` to my path statement in my `.bash_profile` file.

If everything works, congratulations! Go have a coffee. You're on your way to becoming a UNIX guru.