

# Unix/Linux Primer

Taras V. Pogorelov and Mike Hallock  
School of Chemical Sciences, University of Illinois

August 25, 2017

This primer is designed to introduce basic UNIX/Linux concepts and commands. No prior knowledge of Unix/Linux is required.

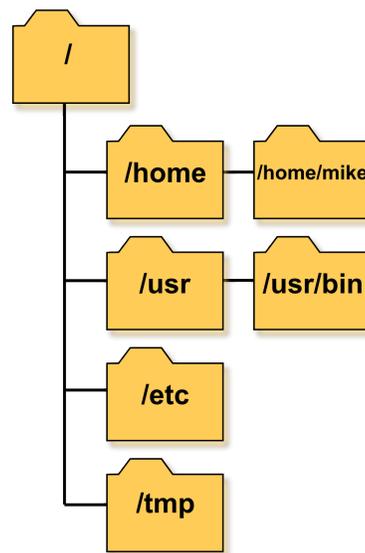
## 1 Filesystem Basics

In Unix, the first central concept is that of a filesystem. It is the hierarchical, tree-like structure that provides a unified namespace for everything on the system. The tree is made out of directories that in turn hold files and other directories. There is a special directory, called the **root** directory, that represents the very top of the filesystem, and all files and directories are decedents of it. It is written as the forward-slash character (/). A **path** is a sequence of directories one must travel to reach a target file or directory. An **absolute path** is one that begins at the root, and therefore begins with a forward-slash. A path is constructed by joining the directory names travelled to reach a file together with forward-slashes. Consider the following absolute path:

```
/path/to/my/file.txt
```

The file, `file.txt` is located in the directory `my`, that is located in the directory `to`, that is in `path` at the root.

As you work, you will be constantly navigating inside the filesystem. At all times, you will have a **current working directory** that represents the directory you are “in” and that will be used as the starting point for a **relative path**. A relative path is one that does not start at the root of the filesystem, but implicitly begins in your working directory. There are also two specially-named directories that you can use to construct paths, the single-dot and double-dot. Single-dot (`.`) represents the current directory, and double-dot (`..`) represents the parent of the current directory. So consider



**Figure 1:** Filesystem hierarchy. The topmost directory is called the root directory.

our working directory to be `/path/to`, the following are all equivalent relative paths:

```
my/file.txt
./my/file.txt
../to/my/file.txt
```

When you log in to a computer, the program that is started for you is called the **shell**. It is the program you interact with that is interpreting all the commands you are typing in. We can use shell commands to find out where we are in the filesystem, to print out directory contents, and to move between directories. When the shell is ready to take input from you, it will have printed a **prompt** at the beginning of the line. The prompt may contain your username, the machine you are on, the directory you are in, or a whole assortment of other information. The default prompts will vary from system to system, and are configurable. For the purposes of this document, we will use a dollar-sign (`$`) to denote the prompt, and to signify that the line is a command that you can type in. Please note that you will not actually type the dollar-sign (`$`).

The first command to try is to find out where we are. Use `pwd` (print working directory) to display the absolute path for the current working directory:

```
$ pwd
```

If you've just logged in, then you are probably in your **home directory**, which is the space on the system that you can keep your files in. We can use the `ls` command to get a listing of files and directories, and the `cd` (change directory) command to move to a new working directory. However, we may not have anything interesting to look at quite yet, so lets make a new directory:

```
$ mkdir unix-tutorial
```

And now list our current directory:

```
$ ls
```

And go in to our newly-created tutorial directory:

```
$ cd unix-tutorial
```

Note that I used a *relative* path to get to my new directory, where I could have also used an absolute path. If my home directory happens to be `/home/mike`, I could have also typed the *absolute path*:

```
$ cd /home/mike/unix-tutorial
```

Tip: `cd` with no arguments will always return you to your home directory. The tilde (`~`) is also a shortcut for your home directory, so `cd ~` will return you home as well.

As it is pretty empty in here, lets use the copy command to bring a file in to this directory. We will grab a list of dictionary words to play with later:

```
$ cp /usr/share/dict/words .
```

The copy command takes two arguments, the source file first, and the destination second. Note we used an absolute path to the words file in the filesystem, and used the relative `.` as the destination, to signify we want to copy to the directory we are currently working in. You can now run `ls` to verify the file is in your directory.

Now, make a new directory, and lets make another copy of the words file.

```
$ mkdir testing124
$ cp words testing124
```

Here we used the relative path of the directory we just made as the destination.

We've used `ls` so far to view our current working directory, but we can use it to view other directories too:

```
$ ls testing124
$ ls /usr/share/dict
```

Oops, looks like I made a slight typo. I wanted the directory to be called "testing123". We will use the *move* command `mv` to rename that:

```
$ mv testing124 testing123
```

Like `cp`, `mv` takes two arguments: the source name and the destination name. Do an `ls` to verify the directory name changed. Do `ls` on the renamed directory to verify that the contents of the directory are unmodified.

Now lets do one more copy:

```
$ cd testing123
$ pwd
$ cp ../words words2
```

Here, we used the relative directory symbol `..` to refer to the parent directory to use it as the source of the file we want to copy. What is the parent directory? What directory is the parent of the parent (`../..`)? What do you think would have happened if we used `.` as the destination for that last copy command?

The outcome of using copy and move depends on the source and destination arguments. The behavior changes whether the source is a file or a directory, and if the destination is a file, directory, or does not yet exist.

If we run the command `mv src dst`:

- If `dst` doesn't exist, then `src` is renamed to `dst`.
- If `dst` is a directory, then `src` is moved in to the directory.
- If `src` and `dst` are files, then `src` replaces `dst`; meaning `dst` is deleted and `src` is renamed to `dst`.
- If `src` is a directory, and `src` is a file, then this is an error! A directory cannot replace a file.

For `cp`, the rules are similar. If you copy a file to a new filename, you end up with two copies of the file. If you copy a file with a destination of a directory, you get a copy of the file in that directory. However, if the source is a directory, you'll get a puzzling message that the directory you tried to make a copy of has been omitted for the copy. To copy directories requires the use of an *option* to the `cp` command to ask it to recursively make a copy of the whole directory. We will learn about command options in the next section.

## 2 Command Options and Arguments

So far, we've run commands that require arguments (like `cp` and `mv`), and those where the arguments are optional (`cd` and `ls`). Nearly all commands also support a wide array of **options** that modify their behavior in some way. Options come in two flavors: those that require an argument of their own, and those that do not (these are commonly referred to as **flags**). Options are typically written as a dash and a single letter, or two dashes and a word. For options that require an argument, the argument should directly follow the option. Lets look at some examples to make this a bit more clear.

Lets go back to your home directory and use the “long listing” option for `ls`:

```
$ cd
$ ls -l
$ ls -l unix-tutorial
```

Before, `ls` just gave us the list of file and directory names, but now we have seven columns of output. The first column are the *permissions* of the file, the second is the number of links, third is the owner of the file and the fourth is the group. The fifth column is the file size, the sixth is the *modification date* of the file, and finally the last column is the filename.

We'll get to permissions and ownership another time, but one thing to note is if the first character of the permissions is a “d”, then that means it is a directory. Some terminals will colorize the output and directories will show up as a different color, but lacking that, this is the way you can distinguish directories from regular files.

The `ls` command has a large array of options that can tell you a lot of information, and also has a collection of options for sorting the output too. Sort the files by *modification date* (`-t`), and in reverse-order (`-r`) so that the newest files are on the bottom of the list:

```
$ ls -l -r -t
```

Options are generally order-independent, and single-letter options may be combined for brevity. The following are equivalent to the above command:

```
$ ls -lrt
```

```
$ ls -ltr
```

Be careful though when mixing options that require arguments with those that do not, as now ordering and grouping matter. Consider the `tar` command (short for Tape ARchive) that creates and extracts *tarfiles*, which is sort of the UNIX equivalent of ZIP files.

```
$ tar -z -x -f file.tar.gz -v
```

This runs `tar` with the options `-z -x -v`, and the `-f` option specifies the file to operate on. When options take arguments, an option that has an argument must be last, although more options may follow. The `-z` flag indicates that the file is zipped (compressed), the `-v` stands for verbose (a common option in unix, it makes programs usually print more information), the `-x` stands for eXtract (as opposed to `-c` for Create), and `-f` gives the file to either extract or create. These are all functionally identical:

```
$ tar -f file.tar.gz -z -x -v
```

```
$ tar -zxvf file.tar.gz
```

```
$ tar -xvf file.tar.gz -z
```

### 3 Finding Help

We've only scratched the surface so far with a couple of commands, and even then we've hardly covered everything they can do. Now is a good time to introduce the online help system called `man`. The `man` command displays **manpages** that describe what each command does, tells you what the required and optional arguments are, and what the options and their arguments are.

```
$ man ls
```

When you run the above command, the manpage is loaded in to a **pager**. A pager is a program that allows you to interactively scroll through a long text document on the screen. Since we don't use the mouse for navigation,

we have no scrollbars to click on. The default pager is the `less` command. When viewing a document you can use the following keys:

- **down-arrow**, **enter**, or **j** to scroll down one line
- **up-arrow** or **k** to scroll up one line
- **spacebar** or **f** to go down a page
- **b** to go up a page
- **g** to go to the top of the document
- **G** to go to the bottom of the document
- **q** to quit and exit the pager
- **/** to search
- **h** to get help on all these keys and more

Go ahead and scroll through the manpage to get an idea of everything that is in it until you get to the bottom. Now go back to the top with **g**. Try and figure out what options to use with `ls` in order to figure out the largest file in the directory `/usr/bin` is, and report the size in a human-readable format (e.g., approximately how many kilobytes or megabytes, not the exact number of bytes).

Now, lets use the pager to take a look at our words file:

```
$ cd
$ cd unix-tutorial
$ less words
```

Try searching for the first occurrence of “chemistry”. Type `/chemistry` and press enter. You can search for the next occurrence from your current position by just hitting `/` and pressing enter.

## 4 Searching and Manipulating Text

It may not seem obvious at first, but most of the power of unix lies in its ability to easily manipulate text to perform a wide range of tasks. Lets take a quick look at a very common utility, `grep`.

As the manpage will tell you, `grep` searches for patterns of characters in a file. These patterns are called **regular expressions**, which is a special grammar for describing a text string to search for. Many unix utilities will utilize regular expressions, so having a basic understanding is important.

The `grep` command will take its first command-line argument as the pattern to search for, and any subsequent arguments as files to search for the pattern. A regular expression can be as simple as a word. Lets look through the words file again and pull out all the entries that contain the word “chemistry”:

```
$ grep chemistry words
```

The command only prints the lines of the file that matched the pattern we asked for, which in this case, was just a single word. Now, lets say we want to find all the words that start with “chem”.

```
$ grep chem words
```

This returned more than we wanted. We got all words that contained the pattern. We can use the `^` character to anchor the pattern to match at the beginning of the line:

```
$ grep ^chem words
```

or anchor at the end with `$`:

```
$ grep chem$ words
```

The period in a pattern is a wildcard, it will match any character. Try the following:

```
$ grep ch.m$ words
```

In addition to matching characters exactly, you can conditionally match characters. The question-mark will match if the preceding character is present or

not. A plus-sign will match any number of repeats of the preceding character, and an asterisk takes the plus-sign and expands upon it, and will match if the preceding character is present or not.

```
$ grep 'physics\?$' words
```

```
$ grep 'lag\+e' words
```

```
$ grep 'x*hello' words
```

The “match all” period character is commonly used with the asterisk, which means zero or more of the preceding character. The two together (`.*`) can match any amount of text. This can be helpful. Try the following:

```
$ grep '^super.*man$' words
```

These last few commands have all had the patterns enclosed in single quotes. The reason for that is to prevent the shell from interpreting some of these characters that have special meaning. For instance, asterisk is also a wildcard character, where it can match multiple filenames. We also have to use backslashes before some of the special characters to get all flavors of `grep` to recognize them.

Another helpful tool is `awk`. It takes text one line at a time and breaks the text into individual words. There are many features to `awk`, but to get started we will use it to simply select which columns of text we want, and only print those.

Type the command `date`. It tells you the current date and time. Lets use `awk` to print only the day, month, and year. When `awk` reads a line of text, it splits it into words, and saves each word into a variable, starting with `$1` for the first word, `$2` for the second, and so on. We can then use the `print` command to output these variables back to the screen. Type the following:

```
$ date | awk '{print $3,$2,$6}'
```

The vertical bar character following `date` is called a pipe, and is the subject of the next section.

## 5 Redirection and Piping

While having a collection of commands to perform text manipulation is helpful, the real power is how you can use them together. We've been using the commands so far to read from a file, but we can also **pipe** the output of one command to be the input for another to make multiple manipulations at once:

```
$ grep super words | grep man
$ grep super words | grep man | sed -e 's/super/hulk/'
```

The pipe operator of the shell is connecting the first command's **standard out** with **standard in** of the next command. Standard in and out are the technical terms for the default place that a command reads from or writes to, respectively. Typically, when you run a command, standard out is your screen, and standard in is the keyboard. However, you can conceptually think of them as a streams of text, and we use the pipe operator to divert the stream of text leaving one command to be the stream of text that enters another. See how we didn't specify a file to read from for the second **grep**? Without a file name, it reads from standard in by default. We also used **sed** here to perform a substitution. It used the output of the second **grep** command as its input.

Not all commands work line-by-line. For example, **wc** counts words and lines:

```
$ wc -l words
$ grep cat words | wc -l
```

**Redirection** refers to the action of taking the output from any command and saving it as a file, or taking a file and using it as the input for a command. Where piping allowed us to attach commands together, redirection allows us to attach a file to the input or output of a command.

To redirect output from a command and save it as a file, use the greater-than sign:

```
$ ls -l > file_list.txt
```

Not as frequently used, but standard input can be pulled from a file using the less-than sign:

```
$ grep unix < file_list.txt
```

## 6 Practice

Download the 1QD5 PDB file using the `curl` command:

```
$ curl https://files.rcsb.org/download/1qd5.pdb.gz > 1qd5.pdb.gz
```

The downloaded file is compressed, use `gunzip` to expand it:

```
$ gunzip 1qd5.pdb.gz
```

Note that `gunzip` will have stripped the `.gz` suffix off the filename. Use `ls` to verify that.

What is this protein? Hint: search the file for “TITLE”.

Use `less` to take a quick look at the file and see its structure.

The PDB file format is made up of different types of records, with the first word on each line describing the type of data found on that line. You’ll see that individual atoms that make up the protein are listed on lines that start with the word `ATOM`. Use `grep` and `wc` to determine how many atoms are in the structure.

How many residues are in the structure? Column 6 of the `ATOM` lines is the *residue id*, which is a number assigned to each amino acid in sequence order. Use the command `awk '{print $6}'` in a series of pipes to extract and print the sixth column of the lines that begin with `ATOM`. Why does each id show up multiple times? Use `uniq` command to remove consecutive, repeated lines. What change to the `awk` command could you do in order to get the residue type (the three letter code in column 4) printed along the residue number? Finally, count the total number of residues, which tells us how long this protein is.