How to use Unix

Unix is a *family* of operating systems that has more than forty years of history. They share similar designs to one another, and while they're certainly not the only family of operating systems out there (Microsoft Windows is another major family), you'll need to be familiar with Unix as a programming environment to succeed in the computing industry.

Linux and OS X are the most popular members of the Unix family. The labs in building 314 use Red Hat Enterprise Linux, which is one particular *distribution* of Linux, while Ubuntu and Debian are others. Android and iOS are also based on Unix, but they're rarely used with a *command-line interface*, or CLI, which is the way you'll need to interact with Unix in this unit.

Where can I get help?

If there's a `help` command, it would usually only help you with the shell itself. `man` and `info` run a program called a *pager*, which allows you to scroll through documents that are longer than your terminal is tall. Pagers use a variety of *key bindings*, but most pagers support only some of these:

- Move down a document with <`j`>, <down>, <`page down`>, <`enter`>, or <space>
- Move up a document with <k>, <up>, or <`page up`>
- Quit the pager with <q>

Command	**Purpose**
`man` *`some_command`*	Read the manual page for *some_command*
`info` *`some_command`*	Read a more detailed manual (mostly only for GNU programs)
`apropos` '*`search term`*'	Find a command whose `man` page has a matching summary
`whatis` *`some_command`*	Print the summary from the `man` page for *some_command*
`whereis` *`some_command`*	Print the path to *some_command* and other paths significant to it

It's 2016… why would anyone type commands to control a computer?

I mean, just look at some of them. `ps -ef`. `tar cvpzf`. `grep -Einsv a.+b`. Unix commands are often cryptic and difficult to learn, but they have their benefits over graphical user interfaces. With some practice, there are many tasks that you can complete much more efficiently with a command-line interface, especially if you're working with many files or other input values.

This command creates ten directories, `prac01` to `prac10`. Try doing *that* in a graphical window!

```
$ mkdir prac{01..10}
```

Note	When shown at the start of a command, `$` is only there to represent the shell's prompt. Type everything that follows the symbol, but not the symbol itself.

Terminals and shells

A long time ago, computers were much more expensive than they are now, so an organisation might only have one or two of them. Users would *log in* to them by using a *terminal*, which was essentially just a monitor and a keyboard, and computers could have many terminals attached to them.

Now that computers are cheap and the labs in building 314 have dozens of them, terminals are much rarer, at least in the original sense of the word. Each lab machine has one monitor, one keyboard, and one mouse, and you can use these *peripherals* directly.

Instead of terminals, each lab machine has several *virtual consoles*, which you can reach with `Ctrl+Alt+F1` through `Ctrl+Alt+F12`, and a program called a *terminal emulator*, which you can open as a window from your *desktop environment* (a graphical user interface). These aren't exactly the same as a terminal, but they all provide a command-line interface, and they work almost identically.

Switching virtual consoles, opening a terminal emulator, and even using `ssh` to log in to a computer over the Internet can all be referred to as "connecting to" or "opening" a terminal.

When you connect to a terminal, the first program to run is usually a *shell*, which allows you to run other programs by entering Unix commands. The shell will *print* a *prompt* that may look like this:

```
[17065012@box tmp]$
```

In this example, `17065012` is your user name, `box` is the name of your lab machine (host name), and `tmp` is the name of your current *working directory*. The trailing `$` simply means that you're looking at a prompt, but some shells use `#`, `%`, `>`, or another character. Shells aren't the only programs that have prompts — any program that has a CLI will usually have a prompt, such as `ftp` and `python`.

The file system

Programs are files that contain instructions for a computer. These instructions are often *machine code*, which has been *compiled* and will then be *executed* by the CPU directly, while other programs, like *shell scripts*, may be *interpreted* by another program. *Processes* are instances of running programs.

Like most operating systems, Unix has *file systems*, which are *hierarchies* of files that are arranged in a structure that's mostly like a tree. When a process opens a file for reading or writing, it's said to have a *file descriptor* or *handle* for the file.

Files in Unix can have any number of names, or even no names, and these names are called *links*, *hard links*, or *paths*. Deleting a file simply removes one of its names, so it's sometimes called *unlinking* a file. A file ceases to exist when it no longer has any names *and* no processes have a file descriptor for it.

Not all files are *regular files*, which represent a sequence of bytes on a file system. One kind of *special file* is a *directory*, which is a file that contains other files as its *children*.

Paths are names that you can use to refer to files. The directory names along a path are separated by slashes (`/`). Any *paths* that you use in a command will be *relative* to your current working directory, unless you start the path with a slash. Every directory has at least two children — one that refers to the directory itself (`.`) and one that refers to the directory's parent (`..`).

Commands

The instructions that you use to control the shell are called *commands*, and they consist of one or more *arguments*. The first argument names the program to be run, while any subsequent arguments are, as far as Unix is concerned, *optional* inputs to the program. In the following command:

```
$ ls -la /usr/bin /var/log
```

- `ls` is the name of the program (thanks to `$PATH`, the fact that it's in `/bin` won't be a problem)
- `-la` are the *options* (options are *usually* single letters, and they may *usually* be combined)
- `/usr/bin` and `/var/log` are the other arguments

Some commands don't care about the order of your arguments, while others do. *Some* operating systems require you to *pass* all of your options to a command before any other arguments.

Working with the file system

Command	**Purpose**
`cd` *`path`*	Change your working directory to *`path`*
`ls [-Ralt]` *`path`*	List the children of the directory at *`path`*
`cp [-Rfip]` *`source dest`*	Copy the file at *`source`* to *`destination`*
`mv [-fi]` *`source dest`*	Move the file at *`source`* to *`destination`*
`rm [-Rfi]` *`path`*	Remove the file at *`path`*
`mkdir [-mp]` *`path`*	Create a new directory at *`path`*
`chmod [-R]` *`mode path`*	Change the permissions of *`path`* to *mode* (e.g. `755` or `+x` or `o-w`)
`.` *`path`*	Execute *`path`*, as if you had typed its contents into the current shell

See also: `pwd`, `find`, `rmdir`, `chown`, `chgrp`, `dd`, `df`, `du`, and `quota`.

Creating and managing tarballs (`tar` archives)

Command (after `tar`)	**Purpose**
`-c[vz]f` *`ball path...`*	Create a tarball at *`ball`* containing copies of the files at *`path...`*
`-t[vz]f` *`ball`* `[`*`path...`*`]`	Print the contents of *`ball`*, optionally filtering by *`path...`*
`-u[vz]f` *`ball path...`*	Update *`ball`* with only the files in *`path...`* that have changed
`-x[pz]f` *`ball`* `[`*`path...`*`]`	Extract (copy) the contents of *`ball`*, optionally filtering by *`path...`*

Working with processes

Command	**Purpose**
`ps [-eflu]`	Print a list of running processes and some details about them
`top [-dnpu]`	Monitor a list of running processes and some details about them
`kill [-`*`signal`*`]` *`pid`*	Send *`signal`* (e.g. `HUP`, `INT`, `TERM`, `KILL`) to a process with the ID *`pid`*
`pgrep [-fguv]` *`pattern`*	Print the IDs of any processes that match the given *`pattern`*
`fg [[%]`*`n`*`]`	Continue a recently stopped process/job *n* in the *foreground*
`bg [[%]`*`n`*`]`	Continue a recently stopped process/job *n* in the *background*

See also: `pkill`, `disown`, `nohup`, and `xargs`.

Working with text and other data

Command	**Purpose**
`vi` *`file`*	Edit the given *`file`* using `vi`, the recommended text editor
`echo` *`words...`*	Print the given *`words...`* to *standard output* (usually your terminal)
`cat` *`path...`*	Concatenate (join) the given files, while printing their contents
`less` *`path`*	Display the contents of the given *`path`* in a pager (alternative: `more`)
`grep [-cinv]` *`pattern`*	Print only the lines of the input that match the given *`pattern`*
`sed s/`*`from`*`/`*`to`*`/[`*`g`*`]`	Print the input, where any text matching *`from`* is replaced with *`to`*
`diff` *`foo bar`*	Print only the lines that are different between *`foo`* and *`bar`*

See also: `printf`, `tee`, `cut`, `paste`, `head`, `tail`, `od`, `more`, `egrep`, `fgrep`, `sort`, `uniq`, and `wc`.

Combining commands with one another

Command	**Purpose**
`foo` `|` *`bar`*	Take the output of *`foo`*, and pass it to *`bar`* as input
`foo` `>` *`path`*	Take the output of *`foo`*, and send it to *`path`* instead of the terminal
`foo` `<` *`path`*	Execute *`foo`*, but take input from *`path`* instead of the terminal

See also: ComSSA's revision material for the Unix and C Programming (COMP1000) exam <https://docs.google.com/presentation/d/1V0daPBXxOrxb4Ckrfhodf2QO-6Ag067y8am479oIjB0>