

Physics 210: Unix / Linux

Please report all errors/typos. etc to choptuik@phas.ubc.ca

Last updated September 29, 2009.

Index

- Introduction and Motivation
 - Interaction with the Shell: Keyboard and Mouse Features
 - Linux Desktop Environments
 - Files and Directories
 - Absolute and Relative Pathnames, Working Directory
 - Home directories
 - "Dot" and "Dot-Dot"
 - Filenames
 - Commands Overview
 - General Structure
 - Executables and Paths
 - Control Characters
 - **bash** Startup Files
 - Hidden Files
 - Shell Aliases
 - Default 210 Startup Files
 - Shell Options
 - Basic Commands
 - Getting Help or Information
 - **man**
 - Communicating with Other Machines
 - **ssh**
 - **Mail**
 - Exiting **bash** / Logging out
 - **exit**
 - **logout**
 - Changing Your Password
 - **passwd**
 - Creating, Manipulating and Viewing Files
 - **Text editors: kate, gedit, vi/vim (gvim) or emacs (xemacs)**
 - **more**
 - **lpr**
 - **cd** and **pwd**
 - **ls**
 - **mkdir**
 - **cp**
 - **mv**
 - **rm**
 - **chmod**
 - **scp**
 - More about **bash**
 - Local Variables
 - Environment Variables
 - Using **bash** Pattern Matching
 - The **bash** History
 - Standard Input, Standard Output and Standard Error
 - Input and Output Redirection
 - Pipes
 - Regular Expressions and **grep**
 - Using Quotes: (' , " , and `)
 - Forward quotes: ' '
 - Double quotes: " "
 - Backward quotes: ` `
 - Job Control
 - Basic Shell Programming
-

INTRODUCTION AND MOTIVATION

These notes aim to get you familiar with the *interactive* use of Unix for day-to-day organizational and programming tasks, as well as to introduce you to the art of *writing shell scripts*. A natural question that you might have is, "Why

should I learn Unix?" Entire essays could be devoted to this topic, but in the context of computational physics, let us first observe that Unix is *the* operating system of choice for many computational physicists, especially those who do significant development of new programs, as well as those who use very large and powerful computers (i.e. those involved in High Performance Computing (HPC)). Unix is powerful, extensible, and in the case of Linux, the code for the entire operating system is so-called *open source*, meaning it is non-proprietary and can be modified by any user as he or she wishes. Additionally, Linux is generally available free of charge, which is important for those on a budget, especially if they operate a large number of computers. Of course, this fact doesn't constitute a particularly compelling argument from a scientific or operational point of view, but in the "real world", and certainly in academia, affordability often *does* make a huge difference. I could go on, but perhaps it is sufficient to conclude by stating that in the current era, most physicists would consider at least a basic knowledge of Unix to be an essential part of the training of undergraduate physics students.

Having provided some rationale for our study, let us first note that Unix is an *operating system* (OS), which we can loosely define as a collection of *programs* (often called *processes*) that manage the resources of a computer for one or more users. These resources include the CPU (Central Processing Unit), network facilities, terminals, file systems, disk drives and other mass-storage devices, printers, and many more. During the course, one common way you will be encouraged to use Unix is through a command-line interface; you will type commands to create and manipulate files and directories, start up applications such as text-editors or plotting packages, communicate with other machines etc. etc. As some of you may be aware, various Unix vendors have written GUIs (Graphical User Interfaces) for their particular versions of Unix. As with similar systems on Macs and PCs, these GUIs largely eliminate the need to issue commands by providing intuitive visual metaphors for most common tasks. However, the command-line approach is still well worth mastering for a variety of reasons, including:

- *Portability*: All Unix systems support the command-line approach, and by sticking with standard features, what you learn on the Linux machines that you will be using in the course will be applicable on virtually all Unix systems.
- *Power*: Commands can be extended and combined in a straightforward way. These new commands include not only those supplied by the operating system, but also new commands (programs) that you can create using shell scripting and programming features, or with specific programming languages such as C, Fortran, Java, Python and many others.
- *Speed*: Command-line interfaces minimize the amount of information that needs to be passed from machine to machine when working remotely. If you just want to accomplish a few quick tasks, few things are more annoying than a sluggish GUI. As available network speeds get faster and faster, this becomes less of a concern, but one still encounters situations where the command-line is more effective than a GUI.

The versions of Unix implemented by specific vendors (or programming teams) typically have specific names. In particular, on the Physics & Astronomy machine, **hyper** (full internet address **hyper.phas.ubc.ca**: note that **phas** and **physics** are synonymous in the context of this type of domain name, so you can use either interchangeably), you will be using the **Linux** variant of Unix, originally coded in large part by **Linus Torvalds** for PCs (and building on a huge amount of previous and continuing work by the **GNU** project), and now widely distributed (typically at no cost!) by many different companies and organizations. Further, the specific flavour of Linux that is installed on **hyper** is **Ubuntu**. Again, however, what is described in the following should work largely as-is on other Linux flavours, as well as on other vendor-specific versions of Unix. You should also note that the TAs and I will be happy to help you install Linux on your laptop and/or PC, although we will recommend that you install **Mandriva** on your machines, primarily since that is the distribution of Linux with which we are most familiar.

When you type commands in Unix, you are actually interacting with the OS through a special program called a *shell*, which provides a more user-friendly command-line interface than that defined by the basic Unix commands themselves. In this course, we will focus on the use of **bash** ("bourne again shell") which over the years has become the most commonly used shell in the Linux community. However, you should be aware that there are other shells available for your use, and that you can, in fact, change your default shell using the **chsh** command. In particular, **tcsh** is still in widespread use and if you are interested in learning a bit about its features, you can start with a [version of these notes](#) that discusses it in some detail. Unfortunately, at this time there is an unresolved issue with the use of **tcsh** when using the workstations in the computer lab, so for this course it is best that you adopt/keep **bash** as your default shell.

In the notes that follow, commands that you type to the shell, as well as the output from the commands and the shell prompt (usually denoted "% ") will appear in typewriter font. Here's an example

```
% pwd
/home/choptuik
% date
Mon Aug 31 13:58:54 PDT 2009
%
```

Note that the appearance of the prompt is the shell's way of telling you that is waiting for you to enter a command.

If you are going through these notes online (and if you aren't familiar with Unix, then you *should!*), then you should have at least one active shell running in which to type sample commands. I will often refer to a window in which a shell is executing as the *terminal*.

Interacting with the Shell: Keyboard and Mouse Features

One very useful feature of **bash** is the ability to recall previously executed commands, and to edit them via the "arrow" keys (as well as "Delete" and "Backspace"). After you have typed a few commands, hit the "up arrow" key a few times and note how you scroll back through the commands you have previously issued. Use the "down arrow" the same number of times to return to the basic prompt.

There is another handy feature of **bash** and other shells (which should be enabled by default on your **hyper** accounts), which is generically known as *completion*. The basic idea is that you can type the first few characters of a command name (i.e. the first word typed after the prompt), or a filename (i.e. essentially any word that follows a command name), and then type the TAB key. If there is a unique command name (filename) that begins with the characters that you have typed, the shell will automatically complete the command name (filename). Especially for long names this can save a considerable amount of typing. If the initial few characters that you have typed do *not* uniquely identify a command or filename, then if you type a second TAB (with no other intervening characters), the shell will display a listing of all possible matches, then allow you to continue typing. As you enter additional characters you can again use TAB at any point to attempt a completion (or two consecutive TABS to see all possible completions).

You should also become familiar with the use of the mouse (or equivalent mouse device) to select, cut and paste text within a shell, as well as within many other Unix applications. Unix systems tend to use a *three button* mouse, and the following instructions assume that you are using one: if your mouse (device) only has two buttons, then third-button-actions can often be emulated by depressing the two buttons simultaneously. Here, then, are the basic text manipulation actions that can be achieved using the mouse:

- **Select (highlight) arbitrary text:** Depress and hold down the left mouse button while sweeping over the desired selection. The selected text will be highlighted.
- **Select a single whitespace-delimited string (word):** Double click on the word using the left button. The selected word will be highlighted. (Note: whitespace = spaces/blanks or TAB characters)
- **Select a single line of text:** Triple click with the left button anywhere on the desired line. The selected line will be highlighted.
- **Paste selected (highlighted) text:** Position the cursor at the desired insertion point using a single click of the left mouse or via the arrow keys. Notice that this action will clear the highlighting of the selected text, but that in all of the above cases, the selection process has automatically copied the selected text to a global buffer (what you might know as a clipboard). Depress the middle button, and the previously selected text will be inserted *before* the cursor. The buffer/clipboard contents remain intact until the next selection action.
- **Cut text:** Select text as above and then depress either the **delete** or **backspace** key. The selected text will be deleted and stored in the buffer/clipboard so you can, for example, subsequently paste it per the previous instruction.

Note that you can use these techniques to transfer selected text between different windows; i.e. between different shells, a shell and a text editor window, text displayed in some window, and the URL type-in of your browser etc. etc. Also observe that in many applications, depressing the right mouse button (which is *not* used for the actions described above) will bring down a menu that will typically have selections such as copy, paste, cut, undo ..., and it's a good idea to become familiar with that mechanism. In addition, the mapping of mouse buttons tends to be configurable in Linux distributions, so you may sometimes encounter a system where the right button acts like the left normally does and vice versa. Finally, if you intend to use a Mac to do some of the course work, I assume that you already know how to accomplish text selection, cut and paste and the like.

Linux Desktop Environments

As mentioned above, modern implementations of Unix, including Linux, typically come with GUIs, more specifically known as desktop environments (DEs), through which users interact. These environments are quite similar to what you are no doubt used to from your experience with PCs running **Windows**, or Macs running **Mac OS**. Even though the notes below focus on command-line Unix / Linux, when you login to one of the workstations in the computer lab, you will be fundamentally interfacing with the operating system through a DE. In particular, you will actually have to start up (launch) a terminal application (window) within the DE in order to perform the type of command-line work detailed below.

Although many desktop environments are available for use with Linux, the two most popular are

- **KDE**
- **GNOME**

Both of these are available on **hyper**, and you should be able to use either for doing your coursework. However, unless you are already using **GNOME**, I suggest that you use **KDE**, since it is somewhat easier to configure and customize, and because the TAs and I have more experience with it. Also, each of these DEs tends to come with a similar set of applications (e.g. terminal (shell) windows, text editors, web browsers, calendar programs etc.), and the applications written for **KDE** will generally work under **GNOME** and vice versa.

In our first lab session, the TAs and I will demonstrate how to choose the DE at login time, as well as how to do some basic configuration and customization of the **KDE** desktop

FILES AND DIRECTORIES

It is important that you be familiar with the notion of a [hierarchical organization](#) (tree structure) of files and directories that most modern operating systems employ. If you are not, refer to one of the Unix references or on-line tutorials that I have suggested, or ask myself or one of the TAs for help. There are essentially only two types of files in Unix:

- *Plain files*: that contain specific information such as plain text, MATLAB code, executable code, PDF code, a Maple worksheet etc.
- *Directories*: special files that serve as containers for other files (including other directories). In the Windows/Mac worlds, directories are known as *folders*, and although the 'directory' terminology is used exclusively in these notes, I fully expect that many of you will only find it natural to use 'folder'

Absolute and relative pathnames, working directory: All Unix filesystems are rooted in the special directory called */* (*forward slash*). All files within the filesystem have *absolute pathnames* that begin with */* and that describe the path down the file tree to the file in question. Thus

```
/home/choptuik/junk
```

refers to a file named **junk** that resides in a directory with absolute pathname

```
/home/choptuik
```

that itself lives in directory

```
/home
```

that is contained in the root directory

```
/
```

In addition to specifying the absolute pathname, files may be uniquely specified using *relative* pathnames. The shell maintains a notion of your current location in the directory hierarchy, known, appropriately enough, as the *working directory* (hereafter abbreviated WD). The name of the working directory may be printed using the **pwd** command:

```
% pwd
/home/choptuik
```

If you refer to a filename such as

```
foo
```

or a pathname such as

```
dir1/dir2/foo
```

so that the reference *does not* begin with a */*, the reference is identical to an absolute pathname constructed by prepending the WD, followed by a */*, to the relative reference. Thus, assuming that my working directory is

```
/home/matt
```

the two previous relative pathnames are identical to the absolute pathnames

```
/home/matt/foo
/home/matt/dir1/dir2/foo
```

Note that although these files have the same filename **foo**, they have different absolute pathnames, and hence are distinct files.

Home directories: Each user of a Unix system typically has a *single* directory called his/her *home directory* that serves as the base of his/her personal files. The command **cd** (change [working] directory) with no arguments will always take you to your home directory. On **hyper** you should see something like this

```
% cd
% pwd
/home2/phys210t
```

When using **bash**, you may refer to your home directory using a tilde (~). Thus, assuming my home directory is

```
/home/choptuik
```

then

```
% cd ~
```

and

```
% cd ~/dir1/dir2
```

are identical to

```
% cd /home/choptuik
```

and

```
% cd /home/choptuik/dir1/dir2
```

respectively. (Note that `cd dirname` cause the shell to change the working directory to *dirname*, assuming that *dirname* is a directory.) **bash** will also let you abbreviate other users' home directories by prepending a tilde to the user name. Thus, provided I have [permission](#) to change to **phys210t**'s home directory,

```
% cd ~phys210t
```

will take me there.

"Dot" and "Dot-Dot": Unix uses a single period (.) and two periods (..) to refer to the working directory and the parent of the working directory, respectively:

```
% cd ~phys210t/dir1
% pwd
/home2/phys210t/dir1
% cd ..
% pwd
/home2/phys210t
% cd .
% pwd
/home2/phys210t
```

Note that

```
% cd .
```

does nothing---the working directory remains the same. However, the . notation is often used when [copying](#) or [moving](#) files into the working directory.

Filenames: There are relatively few restrictions on filenames in Unix. On most systems (including Linux systems), the length of a filename cannot exceed 255 characters. Any character except slash (/) (for obvious reasons) and "null" may be used. However, you should avoid using characters that are special to the shell (such as () * ? \$!) as well as blanks (spaces). In fact, it is probably a good idea to stick to the set:

```
a-z A-Z 0-9 _ . -
```

As with other operating systems, the period is often used to separate the "body" of a filename from an "extension" as in:

```
program.c (extension .c)
paper.tex (extension .tex)
the.longextension (extension .longextension)
noextension (no extension)
```

Note that in contrast to some other operating systems, extensions are *not* required, and are not restricted to some fixed length (often 3 on other systems). In general, extensions are meaningful only to specific applications, or classes of applications, not to *all* applications. The underscore and minus sign are often used to create more "human readable" filenames such as:

```
this_is_a_long_file_name
this-is-another-long-file-name
```

You *can* embed blanks in Unix filenames, but it is not recommended.

Unix generally makes it difficult for you to create a filename that starts with a minus. It is also non-trivial to get rid of such a file, so be careful. If you accidentally create a file with a name containing characters special to the shell (such as * or ?), the best thing to do is [remove](#) or [rename \(move\)](#) the file immediately by enclosing its name in single quotes to prevent shell evaluation:

```
% rm -i 'file_name_with_an_embedded_*_asterisk'  
% mv 'file_name_with_an_embedded_*_asterisk' sane_name
```

Note that the single quotes in this example are [forward-quotes](#) (' '). [Backward quotes](#) (` `). have a completely different meaning to the shell.

COMMANDS OVERVIEW

General Structure: The general structure of Unix commands is given schematically by

```
command_name [options] [arguments]
```

where square brackets ([...]) denote optional quantities. *Options* to Unix commands are frequently single alphanumeric characters preceded by a minus sign as in:

```
% ls -l  
% cp -R ...  
% man -k ...
```

On Linux systems, many commands also accept options that are longer than a single character; by convention, these options are preceded by *two* minus signs as in:

```
% ls --color=auto -CF
```

Arguments are typically names of files or directories or other text strings that *do not* start with - (or --). Individual arguments are separated by whitespace (one or more spaces or tabs):

```
% cp file1 file2  
% grep 'a string' file1
```

There are two arguments in both of the above examples; note the use of single quotes to supply the **grep** command with an argument that contains a space. The command

```
% grep a string file1
```

which has three arguments has a completely different meaning.

Executables and Paths: In Unix, a command such as **ls** or **cp** is usually the name of a file that is known to the system to be executable (see the discussion of [chmod](#) below). To invoke the command, you must either type the absolute pathname of the executable file or ensure that the file can be found in one of the directories specified by your *path*. In **bash**, the current list of directories that constitute your path is maintained in the [environment variable](#), **PATH** (note that case *is* significant for **bash** variables). To display the contents of this variable, type:

```
% echo $PATH
```

(Observe that the **\$** mechanism is the standard way of *evaluating local variables* and *environment (global) variables* alike, and that the **echo** command simply "echoes" its arguments). On **hyper**, the resulting output should look something like

```
./:/home2/phys210t/bin:/home/phys210/bin:/etc:/usr/etc:/usr/ucb:/bin:/usr/bin:/usr/local
```

Note that the directories in the path are separated by a colon (:) *and no whitespace* and that the **.** in the output indicates that the *working directory is in your path*. The order in which path-components appear in the path (first **.** (dot), then **/home2/phys210t/bin**, then **/home/phys210/bin**, etc.) is important. When you invoke a command without using an absolute pathname as in

```
% ls
```

the system looks in each directory in your path---and in the specified order---until it finds a file with the appropriate name. If no such file is found, an error message is printed:

```
% helpme  
-bash: helpme: command not found
```

The path variable is often set for you in a special system file each time a shell starts up, and it is conventional to modify the default setting by setting the **PATH** environment variable in your **~/.bashrc** file. For an example, view the contents of the [course default ~/.bashrc](#) below.

Control Characters: The following control characters typically have the following special meanings or uses within **bash**. (If they don't, then your keyboard bindings are "non-standard" and you may wish to contact the system

administrator about it.) You should familiarize yourself with the action and typical usage of each. I will use a caret (^) to denote the Control (Ctrl) key. Then

```
% ^Z
```

for example, means depress the z-key (upper or lower case) *while simultaneously holding down the Control key*.

- **^D: End-of-file (EOF).** Type ^D to signal end of input when interacting with a program (such as **Mail**) that is reading input from the terminal. Here's an example using **Mail** on **hyper** machines:

```
% Mail -s "test message" choptuik@physics.ubc.ca
This is a one line message.
^D
Cc:
%
```

If you try the above exercise, you will notice that the shell does not "echo" the ^D. This is typical of control characters---you must know when and where to type them and what sort of behaviour to expect. In this case, **Mail** prompts for an optional list of addresses to which the message is to be carbon-copied, but other commands, such as **cat**, will not echo anything. In almost all cases, however, you should be presented with a command prompt, once you have typed ^D. Also, by default, **bash** exits when it encounters EOF, so if you type ^D at a shell prompt, you may find that you are logged out from the terminal session. If you don't like this behaviour (I don't), include the following line in your **~/.bashrc** (it is included in the course default **~/.bashrc**):

```
set -o ignoreeof
```

Note that **set** is a **bash** builtin command (i.e. a sub-command of the **bash** interpreter) that controls many features of the operation of **bash**, and is discussed in slightly more detail in the [shell options](#) section below.

- **^C: Interrupt.** Type ^C to kill (stop in a non-restartable fashion) commands (processes) that you have started from the command-line. This is particularly useful for commands that are taking much longer to execute or producing much more output to the terminal than you had anticipated.
- **^Z: Suspend.** Type ^Z to suspend (stop in a restartable fashion) commands that you have started from the shell. It is often convenient to temporarily halt execution of a command as will be discussed in [job control](#) below.
- **^V: Escape Special Characters:** Type ^V in order to "escape" (protect from shell evaluation) certain other control characters and special keys.

Example 1: Search for [TAB] characters in file **foo** using **grep**:

```
% grep '^V[TAB]' foo
```

Example 2: Removing "Carriage returns" (^M) using **vi**

```
:%s/^V^M//g
```

bash Startup Files: You can customize the environment that results whenever a new **bash** starts by creating and/or modifying certain *startup files* that reside in your home directory. Before proceeding, however, we must note that **bash** makes a distinction between *login shells* and *purely interactive shells*, and executes a different startup file (assuming that it exists) in each case. You will start a **bash** login shell if, for example, you connect to **hyper** from a remote machine such as your home computer. In this case you will have to go through the login procedure of typing your user (account) name and your password. On the other hand, shells that you start from the Linux GUI on one of the workstations in the computer lab, for example, will be purely interactive. In this instance the computer already "knows" who you are and that you are logged in, and does not ask you for your login name or password. Given this, the two most important startup files (there are more, but we don't have to discuss them here, and you can get the full details from the **bash** man page) are as follows

- **~/.profile**
Commands in this file are executed each time a login **bash** is started.
- **~/.bashrc**
Commands in this file are executed each time a purely interactive **bash** is started.

For the purposes of this course (and for most users, in my opinion) it is a bit of a nuisance to have two separate startup files, each of which is executed only when a particular type of **bash** starts, since one will generally want the same customization commands executed in both cases. Fortunately, there is a relatively easy fix to this nuisance which is to

1. Do all of your customizations in the **~/.bashrc** file,
2. Keep the **~/.profile** file as simple as possible
3. As the last command in the **~/.profile** file, execute the commands in the **~/.bashrc** file, using for example the

lines

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

Note that the first and third lines of the above constitute a test for the existence of the `~/.bashrc` file, while the `source ~/.bashrc` command then executes the contents of the `~/.bashrc` file, provided that it exists (i.e. the `source` command tells the shell to execute the commands in the file that is supplied as an argument to it). When sourcing or otherwise manipulating files in startup files, you should *always* perform this type of existence check. Otherwise an error message is apt to be generated, and this can sometimes cause problems with the overall startup process.

IMPORTANT!! Whenever you modify `~/.profile` and/or `~/.bashrc` on **hyper** (or any other Unix system, for that matter) you should abide by the following procedure:

1. **ALWAYS MAKE A BACKUP COPY** of the startup file, using for example

```
% cp ~/.bashrc ~/.bashrc.0
```

2. During the process of modifying one of the startup files, always keep at least one terminal window open to the machine until you have tested (via `ssh` to the machine, for example, see below for information on `ssh`) that you can still login. The reason that this is so vital is that it is possible that your modifications will introduce one or more bugs into the startup files which can make it impossible for you to login. However, as long as the terminal window to the machine remains open, you can kill the failed login process (e.g. the `ssh` command) as necessary using `^C`, try to correct the bugs, and repeat the test login procedure. Finally, if you can't get your desired modifications to work, you can then restore the original contents of the startup file(s) from the saved copy using, e.g.

```
% cp ~/.bashrc.0 ~/.bashrc
```

Hidden Files: Note that files whose name begins with a period (.) are called *hidden files* since they do not normally show up in the listing produced by the `ls` command. Use

```
% cd; ls -a
```

for example, to print the names of *all* files in your home directory. Note that I have introduced another piece of shell syntax in the above example; the ability to type multiple commands separated by semicolons (;) on a single line. There is no guaranteed way to list *only* the hidden files in a directory, however

```
% ls -d .??*
```

will usually come close. At this point it may not be clear to you why this works; if it isn't, you may want try to figure it out after you have gone through these notes and possibly looked at the **man** page for `ls`.

Shell Aliases: As you will discover, the syntax of many Unix commands is quite complicated and, furthermore, the "bare-bones" version of some commands is less than ideal for interactive use, particularly by novices. `bash` provides a simple mechanism called *aliasing* that allows you to easily remedy these deficiencies in many cases. The basic syntax for aliasing is

```
% alias name=string
```

where *name* is the name (use the same considerations for choosing an alias name as for filenames; i.e. avoid special characters) of the alias and *string* is a text string that is substituted for *name* when *name* is used as if it were a command. The following examples should illustrate the basic idea, (see the **bash** documentation (**man bash**) for a few more details, should you wish):

```
% alias ls='ls -FC'
```

provides an alias for the `ls` command that uses the `-F` and `-C` options (these options are described in the discussion of the `ls` command below). Note that the single quotes in the alias definition are essential if the definition contains special characters, including whitespace (recall that whitespace = spaces/blanks and or TAB characters); it is good defensive programming to always include them.

The following lines define aliases for `rm`, `cp` and `mv` (see below) that will not clobber (destroy/overwrite) files without first asking you for explicit confirmation. They are highly recommended for novices and experts alike.

```
% alias rm='rm -i'
% alias cp='cp -i'
% alias mv='mv -i'
```


The following lines define aliases **RM**, **CP**, and **MV** that act like the "bare" Unix commands **rm**, **cp** and **mv** (i.e. that are *not* cautious). Use them when you are sure you are about to do the correct thing: the presumption being that you have to think a little more to type the upper-case command.

```
% alias RM='/bin/rm'  
% alias CP='/bin/cp'  
% alias MV='/bin/mv'
```

To see a list of all your current aliases, simply type

```
% alias
```

Note that all of the preceding aliases (and a few more) are defined in the file **~phys210/aliases** on **hyper**. If you adopt your **.bashrc** and **.profile** from **~phys210/.bashrc** and **~phys210/.profile**, respectively, as we will ask you to do in an early lab session, and also copy **~phys210/aliases** to your home directory, then the aliases will automatically be available for your use when **bash** starts up, since the lines

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

appear in the template **.bashrc**. (Recall that **source file** tells the shell to execute the commands in the file *file*). Although the use of a separate **~/.alias** file is not a "standardized" approach, I commend it to you as a means of keeping your **~/.bashrc** relatively uncluttered if you define a lot of aliases. However, if you wish, you can simply add alias definitions to your **~/.bashrc**, or define them interactively at the command line at any time.

Note that (in contrast to the **tcsh**, for example) there is no facility for *processing* command arguments when using **bash** aliases: the alias mechanism simply (non-recursively) replaces one piece of text with another. However, should you wish to define your own shell commands that *do* process arguments, this can be readily done using shell scripts or shell functions, both of which are discussed below in the [Basic Shell Programming](#) section.

Default 210 Startup Files: You can view the contents of **~phys210/.bashrc**, **~phys210/.profile** and **~phys210/aliases** by clicking on the links below.

- [.bashrc](#)
- [.profile](#)
- [.aliases](#)

Shell Options: As already mentioned above in the context of the End-of-file control character, ^D, many features of **bash** can be controlled using the **set** builtin command. You can refer to the [set builtin](#) section of the online manual (as well as the [man](#) page for **bash**) for complete details on what can be configured, and how to use **set** to do the configuration.

The local variable **SHELLOPTS** stores a colon-delimited list of the currently defined options, so you can view the current options using

```
% echo $SHELLOPTS  
braceexpand:emacs:hashall:histexpand:history:ignoreeof:interactive-comments:monitor:noc
```

where the output from the **echo** command is typical of what you can expect to see on your **hyper** account.

BASIC COMMANDS

The following list is by no means exhaustive, but rather represents what I consider an essential base set of Unix commands (organized roughly by topic) with which you should familiarize yourself as soon as possible. Refer to the [man](#) pages, or one of the suggested Unix references for additional information.

Getting Help or Information:

man

Use **man** (short for *manual*) to print information about a specific Unix command, or to print a list of commands that have something to do with a specified topic (**-k** option, for keyword). It is difficult to overemphasize how important it is for you to become familiar with this command. Although the level of intelligibility for commands (especially for novices) varies widely, most basic commands are thoroughly described in their **man** pages, with usage examples in many cases. It helps to develop an ability to scan quickly through text looking for specific information you feel will be of use. Examples of **man** invocations include:

```
% man man
```

to get detailed information on the **man** command itself,

```
% man cp
```

for information on **cp** and

```
% man -k compiler
```

to get a list of commands having something to do with the topic 'compiler'. The command **apropos**, found on most Unix systems, is essentially an alias for **man -k**.

Output from **man** will typically look like

```
% man man
```

NAME

man - an interface to the on-line reference manuals

SYNOPSIS

```
man [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L
locale] [-m system[,...]] [-M path] [-S list] [-e extension] [-i|-I]
[--regex|--wildcard] [--names-only] [-a] [-u] [-P pager] [-r prompt]
[-7] [-E encoding] [--no-hyphenation] [-p string] [-t] [-T[device]]
[-H[browser]] [-X[dpi]] [-Z] [[section] page ...] ...
man -k [apropos options] regexp ...
man -f [whatis options] page ...
man -l [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L
locale] [-P pager] [-r prompt] [-7] [-E encoding] [-p string] [-t]
[-T[device]] [-H[browser]] [-X[dpi]] [-Z] file ...
man -w|-W [-C file] [-d] [-D] page ...
man -c [-C file] [-d] [-D] page ...
man [-hV]
```

DESCRIPTION

man is the system's manual pager. Each page argument given to man is normally the name of a program, utility or function. The manual page associated with each of these arguments is then found and displayed. A section, if provided, will direct man to look only in that section of the manual. The default action is to search in all of the available sections, following a pre-defined order and to show only the first page found, even if page exists in several sections.

·
·
·

for a specific command and,

```
% man -k language
```

```
QLocale (3qt) - Converts between numbers and their string representati...
asy (1) - Asymptote: a script-based vector graphics language
awk (1) - pattern scanning and text processing language
bc (1) - An arbitrary precision calculator language
conjure (1) - interprets and executes scripts written in the Magick ...
debconf-getlang (1) - extract a language from a templates file
·
·
·
ruby (1) - Interpreted object-oriented scripting language
ruby1.8 (1) - Interpreted object-oriented scripting language
runantlr (1) - ANOther Tool for Language Recognition
texlua (1) - An extended version of pdfTeX using Lua as an embedded...
texluac (1) - An extended version of pdfTeX using Lua as an embedded...
Text::WrapI18N (3pm) - Line wrapping module with support for multibyte, fullw...
update-language (1) - update various TeX-related configuration files
vmsish (3perl) - Perl pragma to control VMS-specific language features
xasy (1x) - script-based vector graphics language
XtSetLanguageProc (3) - set the language procedure
```

for a keyword-based search. Note that the output from **man -k ...** is a list of commands and brief synopses. You can then get detailed information about any specific command (say **awk** in the current example), with another **man** command:

```
% man awk
```

Also note that the output from **man** is fed ([piped](#)) into the **more** command, so refer to the description of **more** below (or the **man** page for **more!**) for some details that will allow you to page forward and backward, and search for text, in a particular **man** page.

Communicating with Other Machines:

ssh

Use **ssh** to establish a secure (i.e. encrypted) connection from one Unix machine to another. This is the basic mechanism that can be used to (1) start a Unix shell on a remote host and (2) execute one or more Unix commands on such a machine.

Typical usage of **ssh** is

```
bh0% ssh hyper.phas.ubc.ca -l choptuik
```

which will initiate a remote-login for user **choptuik** on the machine **hyper.phas.ubc.ca**. When I enter this command, I will be prompted for my password (for the account **choptuik**) on **hyper**.

```
choptuik@hyper.phas.ubc.ca's password:
```

The following commands are equivalent to the above invocation:

```
% ssh choptuik@hyper.phas.ubc.ca
% slogin hyper.phas.ubc.ca -l choptuik
% slogin choptuik@hyper.phas.ubc.ca
```

The first of the above alternate forms is generally the most convenient to type, and is the one that I use.

If additional arguments are supplied to **ssh**, they are interpreted as commands to be executed remotely. In this case, control immediately returns to the invoking shell after completion (successful, or otherwise) of the command(s), as seen in the following examples, where the password prompts have been suppressed:

```
hyper% ssh matt@bh0.phas.ubc.ca date
Mon Aug 31 18:57:41 PDT 2009
```

```
hyper% ssh matt@bh0.phas.ubc.ca 'pwd; date'
/home2/matt
Mon Aug 31 18:58:18 PDT 2009
```

```
hyper%
```

Enabling X-forwarding with ssh: One very useful option to the **ssh** command is **-X**, which enables **X11 forwarding**. In a nutshell, this means that if you initiate the **ssh** command from a Unix graphical desktop environment (including **Mac OS**, as well as **KDE**, **GNOME** and others on Linux), and then start a graphical application (such as **konsole**, **kate**, **gedit**, **xmapple**, **matlab**, etc.) on the *remote* machine, the application will display on the *local* graphical desktop. Typical usage with this option would be

```
hyper% ssh -X matt@bh0.phas.ubc.ca
```

```
bh0% kate
```

and, assuming that I was logged into **hyper** using one of the Computer Lab workstations, the **kate** window running on **bh0** would be displayed on my *workstation* desktop. If I had not enabled X11 forwarding in the **ssh** command, then when I tried starting **kate**, I would get an error message such as the following:

```
bh0% kate
kate: cannot connect to X server
```

Note that **X** is the venerable windowing software---developed at MIT---on which almost all Unix desktop environments are ultimately based. **X11** is the current version of the software, and has actually been current since 1987! See the [Wikipedia entry for X11](#) additional information should you be interested.

Gory Details of ssh: In contrast to many of the other commands described here, the behaviour of **ssh** depends crucially on the current *context* for the command, which, by convention, **ssh** stores as a number of files in the directory **~/.ssh** (i.e. as a number of files in a *directory* named **.ssh**, located in your home directory). If **~/.ssh** does not exist (which nominally means that you have yet to issue the **ssh** command from that specific account), it will automatically be created, and certain files within **~/.ssh** will be created and/or modified.

For example, assume that as **choptuik@hyper.phas.ubc.ca**, I have never used the **ssh** command. However, I *can* and *do* login into **hyper.phas.ubc.ca** (as **choptuik**) using one of the workstations in the computer lab, and start up a command shell. I can now establish a secure connection to my account on **bh0.phas.ubc.ca** via **ssh** as follows:

```
hyper% ssh matt@bh0.phas.ubc.ca
```

```
The authenticity of host 'bh0.phas.ubc.ca (142.103.234.164)' can't be established.  
RSA key fingerprint is 14:82:47:40:80:9f:52:7f:39:1f:17:a1:df:76:b2:54.  
Are you sure you want to continue connecting (yes/no)?
```

This message from **ssh** is a warning that essentially tells me that I have not connected before to the host **bh0.phas.ubc.ca**. It gives me a chance to check the **ssh** invocation to ensure that I've typed everything correctly, to safeguard against security issues that we won't delve into here. Because I'm sure that I want to connect, I enter **yes**. The output from the **ssh** command then continues:

```
Warning: Permanently added 'bh0.phas.ubc.ca,142.103.234.164' (RSA) to the list of known  
hosts for matt@bh0.phas.ubc.ca's password:
```

After correctly typing my password for **matt@bh0**, I am left in a shell running on **bh0**, and I can now "work" (i.e. issue Unix commands) within that shell.

When I'm done my work on **bh0**, I can use the **logout** (or **exit**) command

```
bh0% logout  
Connection to bh0.phas.ubc.ca closed.  
hyper%
```

to return to **hyper**.

Assuming I've done the above, I now see that the directory **~/.ssh** has been created, and contains the file **known_hosts**:

```
hyper% cd ~/.ssh  
hyper% ls  
known_hosts
```

The purpose of the **known_hosts** file is to maintain identification information for hosts to which I've previously **ssh**'ed. In particular, the next time I **ssh** from **hyper** to **bh0**, the message 'The authenticity of host ...' will not appear, and **ssh** will "automatically" connect to **bh0**.

```
hyper% ssh matt@bh0.phas.ubc.ca  
matt@bh0.phas.ubc.ca's password:
```

Refer to the **man** page on **ssh** for full details on this command.

Mail

I believe that it is a safe assumption that all of you are already experts in the sending and receiving of e-mail, using one or more of your favorite mail clients, and, at least initially, I will be using the broadcast facility provided via the Student Service Centre (as well as the course [News page](#)) to communicate electronically with the class as a whole. At least at the current time, all of the students registered in the course have an e-mail address linked into this facility, so, in principle, if I send a broadcast message, and assuming everyone checks their appropriate mailbox, you should all receive it. Note that the SSC setup does *not* allow me to directly see your individual e-mail addresses, so if you want to get personalized e-mail from myself or the TAs, you will need to first send us a message to which we can reply.

In this course, we will *not* require you to regularly use any mail client on **hyper** unless you wish to do so. If you do, however, be aware that there are a large number of clients available, and that the TAs and I may have little, if any, experience with the client that you choose. That said, there is one text-based (i.e. doesn't use a GUI) client that many peoples in the Physics and Astronomy Dept use (especially the older folk), which is called **alpine** (a variant of an older client called **pine**). **alpine** is relatively easy to use, has an extensive on-line help facility, and the TAs will be able to provide you assistance with it as necessary. The home page for **alpine**, which contains links to thorough documentation, is located [here](#). In addition, should you decide to use mail on **hyper** on a regular basis, I recommend that you consult the PHAS Computing Staff's on-line [E-mail FAQ](#).

However, in the spirit of mastering command-line Unix we conclude this section with a brief illustration of the use of a very old, and, especially for your generation, a very primitive, mail client known as **Mail** (we've already encountered this program in our discussion of special characters):

Again, here's a basic example showing how to use **Mail** to send a message:

```
% Mail -s "this is the subject" choptuik@phas.ubc.ca
```

```
This is a one line test message.  
^D  
Cc:  
%
```

Note that multiple recipients can be specified on the command line. Another form involves [redirection](#) from a file.

```
% Mail -s "sending a file as a message" matt@laplace.phas.ubc.ca < message
```

sends the contents of file **message** with the subject field of the e-mail set to 'sending a file as a message'.

If you are interested, you can consult the **man** page for **Mail** for additional information on its use, but primarily you should note that although it may seem like obsolete technology, the type of usage illustrated above (which, I should mention, can also be achieved using **alpine**), can be quite useful. For example, you might write a script that takes a long time to accomplish some task. It can then be convenient to have the script send you a message when it has completed. This is cumbersome, if not impossible, to accomplish using fancy GUI-based mail clients, but is essentially trivial with **Mail**.

*Exiting **bash** / Logging out:*

exit

Type **exit** to leave both login and purely interactive shells.

If there are suspended jobs (see [job control](#) below), you will get a warning message, *and you will not be logged out*.

```
% exit  
There are stopped jobs
```

If you then type **exit** a second time (with no intervening command), the system assumes you have decided you don't care about the suspended jobs, and will log you out. Alternatively, you can deal with the suspended jobs, and then **exit**.

logout

logout has the same effect as **exit**, but can only be used in login shells. If you enter **logout** in a purely interactive shell, you will receive the message

```
% logout  
bash: logout: not login shell: use `exit`
```

Changing your passwd

passwd

If you decide that you want to change your password on the physics machines, you will have to login to the main physics server, **warp.physics.ubc.ca**---which can also be referenced as simply **physics.ubc.ca**---and use the **passwd** command to make the change. When you execute **passwd**, you will first be prompted for your current password, and then be asked to type your new password *twice*. The new password should be 8 characters in length---if it's longer, the extra characters are ignored---and must include a mix of upper and lower case alphabetic characters, numbers and special characters other than '&'. The program will tell you if it doesn't like the new password that you have chosen, and will ask you to make up a new one.

If the change is successful, you should see something like this, where I first login to **warp.physics.ubc.ca** using **ssh** from a terminal session on **hyper**, and where none of the passwords I type (new or old) are echoed to the terminal:

```
hyper% ssh choptuik@warp.physics.ubc.ca
```

```
Last login: Sun Sep 13 13:31:36 2009 from hyper.phas.ubc.  
*****  
* 08-11-04 warp had a disk failure. System was restarted from an  
* alternate boot disk. Please report problems to sysadmins.  
*****
```

To print one-sided use "lpr -Zsimplex file".
Undergraduate students will need to arrange for a quota.

```
-----  
type "more /etc/motd" on physics to re-read this message.  
type "more /etc/motd.full" on physics to read the complete message file  
=====
```

Authorized uses only. All activity may be monitored and reported.

```
warp% passwd
Old password:
Password:
Reenter password:
Checking, please wait ...
Password okay. Changing password ...
The command completed successfully.
```

```
warp% exit
Connection to warp.physics.ubc.ca closed.
```

hyper%

A successful password change on **warp** will eventually be propagated to the other physics Unix machines, including **hyper**, but the propagation may not be immediate. You may thus find that you have to use your old password for some time to access your **hyper** account before the new one becomes active.

Unless you intend to use **Mathematica**, which will *not* be covered in 210, and which is not installed on **hyper**, you should not have to use **physics.ubc.ca** for any other purpose during this course.

Creating, Manipulating and Viewing Files (including Directories):

Text editors: kate, gedit, vi / vim (gvim) or emacs (xemacs)

Although you may find it somewhat painful (especially if you've developed a serious relationship with Microsoft Word), I consider it an absolutely key goal for everyone in this course to become reasonably proficient in at least one of the text editors: **kate**, **gedit**, **vi** or **emacs**. Note that a text editor, although similar in spirit to a word processor, really has a different fundamental purpose. As the name suggests, this is to create and manipulate files that contain *plain text* (i.e. files for which many of the features of modern word processors, such as the ability to create documents that use different fonts, font-sizes, styles, colors, include figures etc. etc. are completely irrelevant). Plain text files are central to the use of most programming languages and programming environments in Unix, as well as to the configuration and customization of the operating system itself. During the course, many of the homework assignments, as well as many of your term projects, will require the creation of this type of file.

For many, if not most of you, **kate** and **gedit** are probably the most straightforward choices (they are part of the **KDE** and **GNOME** graphical desktop environments, respectively, but both can run within either **KDE** or **GNOME**). Of all of the editors listed above, they provide the most intuitive interface, namely GUIs which, though not as snazzy looking, closely resemble those of word processors with which you will no doubt be familiar. I expect that most of you will be able to start using either, and to readily master whichever you choose, with little, if any, help. If you do need assistance, the applications provide extensive help facilities. Note that both editors implement the [mouse-text-manipulation features](#) discussed in the introductory section, although they also have GUI menus and buttons for those actions. Also observe that **kate** and/or **gedit** may not be available on some non-Linux Unix systems that you may need to work on, but given the introductory nature of this course, I don't consider that sufficient reason to dissuade you from their use.

vi and **emacs** are the two major "traditional" text-editors that are found on most Unix implementations (certainly **vi** should be!) Both are themselves text-based; that is, they do not provide a GUI, and for the most part, do not allow for manipulation of the text being edited with the mouse. Moreover, **vi** developed a reputation for being suitable mainly for "hardcore" users, who didn't mind dealing with its rather unique, simplistic, and not entirely intuitive (to put it mildly!) user interface. **emacs**, on the other hand, was viewed as a much more elegant, powerful and full-featured editor, to the point that with a suitable configuration, you could get **emacs** to do just about everything but make coffee. Personally, I use **vi**, since that's the editor I first encountered on Unix, and although it is a good idea for *any* Unix user to know a bit about **vi** (if only because some of its syntax appears in many other standard Unix commands), I would strongly recommend that any of you who know neither **vi** or **emacs**, and who wish to learn one of them, seriously consider learning **emacs**, at least to start. In addition, if you intend to become a *serious* Unix user, then you really *should* learn how to use one of these text-based editors---probably **vi** due to its ubiquity and relative simplicity---since you will almost certainly encounter situations where you need to edit files using a basic terminal session that will not support the use of a GUI.

Over the years, **vi** on Linux systems has evolved to become **vim** (for Vi IMproved), so that, for example, if you execute **vi** on **hyper**, it is actually **vim** that starts up. This is a minor point, but something to keep in mind should you be looking online for information concerning **vi** (i.e. you should probably search for information on **vim**).

The good news is that there are now GUI-based versions of both of these editors, **gvim**, for **vi / vim** and **xemacs** for **emacs** and you are more than welcome to use these rather than their text-based antecedents for course work. Note, however, that these GUIs are not as user friendly as the word processing software that you are probably accustomed to (or **kate / gedit** for that matter) but they will, for example, allow you to use the mouse to position the

cursor as well as to [highlight text and cut](#). Again, it is up to you which text editor you choose to use, but we *really* want you to learn to use at least one that isn't a Microsoft or Mac/Apple product!

Not least because all of these text editors are complex pieces of software, it is impossible to give any sort of an overview of their use in these introductory notes. You are therefore directed to the help facilities supplied by the editors themselves, as well as to the online resources listed in the course [Online Resources](#) page, for links to web sites and specific documents that should aid in you in your task of mastering your editor of choice. *This in one job that we expect you to do largely on your own*, but the TAs and I, will of course, be more than happy to help out as we can, and, I hope, some of your classmates will be able to be of assistance as well.

Finally, if you know of, or find, another text editor on **hyper** that you would prefer to use, please send me an e-mail message and I will let you know whether I approve of its use, or whether I would rather you make another choice.

more

Use **more** to view the contents of one or more files, one page at a time. For example:

```
% more /usr/share/dict/words
A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
Aaronic
Aaronical
Aaronite
--More-- (0%)
```

In this case I have executed the **more** command in a shell window containing only a few lines (i.e. my pages are short). The

```
--More-- (0%)
```

message is actually a prompt: hit the space bar to see the next page, type **b** to backup a page, and type **q** to quit viewing the file. You can also search for a string in the output by typing a '/' (forward slash) followed by the text to be located:

```
/misspell
...skipping
misspeak
misspeech
misspell
misspelling
misspend
misspender
misstate
misstatement
misstater
misstay
misstep
missuade
--More-- (49%)
```

Refer to the **man** page for additional features of the command. We have already noted that output from **man** is typically [piped](#) through **more**.

lpr

Use **lpr** to print files. If no options are passed to **lpr**, files are sent to the system-default printer, or to the printer specified by your **PRINTER** [environment variable](#), if it is set. Typical usage is

```
lpr file_to_be_printed
```

The default printer on **hyper** machines is the HP LaserJet 4520dn in Hennings 205 and, by default, printing will be two-sided (duplex). Should you need to make one-sided hard copy, print the file using the **-o sides=one-sided** option:

```
hyper% lpr -o sides=one-sided file_to_be_printed
```

Note that this last form of the **lpr** command is specific to **hyper**.

cd and pwd

Use **cd** and **pwd** to change (set) and print, respectively, your working directory. We have already seen examples of these commands above. Here's a summary of typical usages (again note the use of semi-colons to separate distinct Unix commands issued on the same line):

```
% cd
% pwd
/home/choptuik
% cd ~; pwd
/home/choptuik
% cd /tmp: pwd
/tmp
% cd ~phys210; pwd
/home/phys210
% cd ..; pwd
/home
% cd phys210; pwd
/home/phys210
```

Recall that **..** refers to the parent directory of the working directory so that

```
% cd ..
```

takes you up one level (closer to the root) in the file system hierarchy.

ls

Use **ls** to list the contents of one or more directories. On Linux systems, I advocate the use of the alias

```
% alias ls='ls --color=auto -FC'
```

which will cause **ls** to

1. Append special characters (notably ***** for executables, **/** for directories and **@** for symbolic links) to the names of certain files (the **-F** option),
2. List in columns (the **-C** option).
3. Color-code the output, again according to the type of the file.

Example (with color coding suppressed)

```
% cd ~phys210t
% ls
cmd*  dir1/  dir2/
%
```

Note that the file **cmd** is marked executable while **dir1** and **dir2** are directories. To see hidden files and directories, use the **-a** option:

```
% cd ~phys210t; ls -a
./  .aliases  .bashrc  dir1/  .profile  .Xauthority
../  .bash_history  cmd*    dir2/  .viminfo
```

and to view the files in "long" format, use **-l**:

```
% cd ~phys210t; ls -l
-rwxr-xr-x 1 phys210t ugrad  53 2009-08-31 20:12 cmd*
drwxr-xr-x 4 phys210t ugrad 4096 2009-08-31 20:15 dir1/
drwxr-xr-x 2 phys210t ugrad 4096 2009-08-31 20:16 dir2/
```

The output in this case is worthy of a bit of explanation. First observe that **ls** produces one line of output per file/directory listed. The first field in each listing line consists of 10 characters that are further subdivided as follows:

- first character: file type: **-** for regular file, **d** for directory.
- next nine characters: 3 groups of 3 characters each specifying read (r), write (w), and execute (x) permissions for the user (owner of the file), user's in the owner's group and all other users. A **-** in a permission field indicates that the particular permission is denied.

Thus, in the above example, **cmd** is a regular file, with read, write and execute permissions enabled for the owner (user **phys210t**) and read and execute permissions enabled for members of group **ugrad** and all other users. **dir1** and **dir2** are seen to be directories with the same permissions. Note that you must have execute permission for a directory in order to be able to **cd** to it, and read permission in order to access any of the files it contains (including getting a listing of those files via **ls**). See **chmod** below for more information on setting file permissions. Continuing to decipher the long file listing, the next column lists the number of links to this file (advanced topic) then comes the name of the user who owns the file and the owner's group. Next comes the size of the file in bytes, then the date and time the file was last modified, and finally the name of the file.

If any of the arguments to **ls** is a directory, then the contents of that directory are listed. Finally, note that the **-R** option will recursively list sub-directories:

```
% cd ~phys210t; pwd
/home2/phys210t

% ls -R
.:
cmd*  dir1/  dir2/

./dir1:
file_1  subdir1/  subdir2/

./dir1/subdir1:
file_s

./dir1/subdir2:
file_3

./dir2:
file_4
```

Note how each sub-listing begins with the relative pathname to the directory followed by a colon. For kicks, you might want to try

```
% cd /
% ls -R
```

which will list essentially all the files on the system which you can read (have read permission for). Type **^C** when you get bored.

mkdir

Use **mkdir** to make (create) one or more directories. Sample usage:

```
% cd ~
% mkdir tempdir
% cd tempdir; pwd
/home/choptuik/tempdir
```

If you need to make a 'deep' directory (i.e. a directory for which one or more parents do not exist) use the **-p** option to automatically create parents as needed:

```
% cd ~
% mkdir -p a/long/way/down
% cd a/long/way/down; pwd
/home/choptuik/a/long/way/down
```

In this case, the **mkdir** command made the directories

```
/home/choptuik/a    /home/choptuik/a/long    /home/choptuik/a/long/way
```

and, finally

```
/home/choptuik/a/long/way/down
```

cp

Use **cp** to (1) make a copy of a file, (2) copy one or more files to a directory, or (3) duplicate an entire directory structure. The simplest usage is the first, as in:

```
% cp foo bar
```

which copies the contents of file **foo** to file **bar** in the working directory. Assuming that **cp** is aliased to **cp -i**, as recommended, you will be prompted to confirm overwrite if **bar** already exists in the current directory; otherwise a new file named **bar** is created. Typical of the second usage is

```
% cp foo bar /tmp
```

which will create (or overwrite) files

```
/tmp/foo /tmp/bar
```

with contents identical to **foo** and **bar** respectively. Finally, use **cp** with the **-r** (recursive) option to copy entire hierarchies:

```
% cd ~phys210t; ls -a
./ .aliases .bashrc dir1/ .profile .Xauthority
../ .bash_history cmd* dir2/ .viminfo
```

```
% cd ../; pwd
/home2
```

```
% cp -r phys210t /tmp
cp: cannot open `phys210t/.bash_history' for reading: Permission denied
cp: cannot open `phys210t/.viminfo' for reading: Permission denied
cp: cannot open `phys210t/.Xauthority' for reading: Permission denied
```

```
% cd /tmp/phys210t; ls -a
./ ../ .aliases .bashrc cmd* dir1/ dir2/ .profile
```

Study the above example carefully to make sure you understand what happened when the command

```
% cp -r ~phys210t /tmp
```

was issued. In brief, the directory **/tmp/phys210t** was created and all contents of **/home2/phys210t** (including hidden files) *for which I had read permission* were recursively copied into that new directory: sub-directories of **/tmp/phys210t** were automatically created as required.

mv

Use **mv** to rename files, or to move files from one directory to another. Again, I assume that **mv** is aliased to **mv -i** so that you will be prompted if an existing file will be clobbered by the command. Here's a "rename" example

```
% ls
thisfile
% mv thisfile thatfile
% ls
thatfile
```

while the following sequence illustrates how several files might be moved up one level in the directory hierarchy:

```
% pwd
/tmp/lev1
% ls
lev2/
% cd lev2
% ls
file1 file2 file3 file4
% mv file1 file2 file3 ..
% ls
file4
% cd ..
% ls
file1 file2 file3 lev2/
```

rm

Use **rm** to remove (delete) files or directory hierarchies. The use of the alias **rm -i** for cautious removal is *highly recommended*. Once you've removed a file in Unix there is essentially nothing you can do to recover it other than restoring a copy from backup media (assuming the system *is* regularly backed up), and if the file was created since the last backup, you're really out of luck! Examples include:

```
% rm thisfile
```

to remove a single file,

```
% rm file1 file2 file3
```

to remove several files at once, and

```
% rm -r thisdir
```

to remove *all* contents of directory **thisdir**, including the directory itself. Be particularly careful with this form of the command and note that

```
% rm thisdir
```

will not work. Unix will complain that **thisdir** is a directory.

chmod

Use **chmod** to change the permissions on a file. See the discussion of **ls** above for a brief introduction to file-permissions and check the **man** pages for **ls** and **chmod** for additional information. Basically, file permissions control who can do what with your files. *Who* includes yourself (the user **u**), users in your group (**g**) and the rest of the world (the others **o**). *What* includes reading (**r**), writing (**w**, which itself includes removing/renaming) and executing (**x**, and recall that execution permission on a directory is required in order to **cd** to it). When you create a new file, the system sets the permissions (or mode) of a file to default values that you can modify using the **umask** command. (See the discussion of **umask** in the man page for **bash** for more information).

On **hyper**, your defaults should be such that you can do anything you want to a file you've created, while the rest of the world (including fellow group members) normally has read and, where appropriate, execute permission. As the **man** page will tell you, you can either specify permissions in numeric (octal) form or symbolically. I prefer the latter. Some examples that should be useful to you include:

```
% chmod go-rwx file_or_directory_to_hide
```

which removes all permissions from 'group' and 'others', effectively hiding the file/directory,

```
% chmod a+x executable_file
```

to make a file executable by *everyone* (**a** stands for all and is the union of user, group and other) and

```
% chmod a-w file_to_write_protect
```

to remove *everyone's* write permission to a file, including yours (i.e. the user's), which prevents accidental modification of particularly valuable information. Note that permissions are added with a **+** and removed with a **-**. You can also set permissions *absolutely* using an **=**, for example

```
% chmod a=r file_for_all_to_read
```

scp

Use **scp** (whose syntax is an extension of **cp**) to copy files or hierarchies from one Unix system to another. **scp** is part of the **ssh** distribution, so will prompt you for a password for access to the remote account.

For example, assume I am logged into **hyper** and that I want to copy my **~/ .bashrc** file to a file named **~/ .bashrc-hyper** in my home directory on **matt@bh0.phas.ubc.ca**. The following will do the trick

```
hyper% scp ~/ .bashrc matt@bh0.phas.ubc.ca:~/ .bashrc-hyper
matt@bh0.phas.ubc.ca's password:
. bashrc                                100% 1813      1.8KB/s   00:00
```

The last line in the above output is a status report that lists, in order, the name of file that was transferred, the percentage of the file transmitted (for large files, or on slow connections, you will see this number being updated in "real time"), the number of bytes transferred, the approximate speed of the transfer, and the elapsed time for the copy. If you wish to suppress this output use the **-q** (for quiet) option to the command:

```
hyper% scp -q ~/ .bashrc matt@bh0.phas.ubc.ca:~/ .bashrc-hyper
```

The above example copies a file *from* the local host *to* a remote host. You can use **scp** to go the other way as well: i.e. the command can be used bi-directionally between hosts. Thus, for example, the following invocation will copy my **~/ .aliases** file on **bh0** to the file **~/ .aliases-bh0** on my account on **hyper**:

```
hyper% scp -q matt@bh0.phas.ubc.ca:~/ .aliases ~/ .aliases-bh0
```

WARNING!! Be very careful using **scp**, particularly since there is no **-i** (cautious) option to warn you if existing files

will be overwritten (there actually is a `-i` option, but it serves a completely different purpose!). Also note that there *is* a `-r` option for remote-copying entire hierarchies.

MORE ABOUT `bash`

Local Variables: `bash` allows for the definition of *variables*, which are used to store various pieces of information in the form of text strings. Indeed this basic notion of "variable" should be familiar to you if you have experience programming in a language such as **C**, **Maple**, **Mathematica**, **python**, **MATLAB** etc. Further, `bash` distinguishes between two types of variables: *local variables*, whose values are available only in the current shell, and *environment* or global variables, whose values are inherited (accessible) by processes (including other shells) that are started within the current shell.

The syntax for defining a new local variable (or for changing its value) is simple:

```
% varname=value
```

As with file names and aliases, you should avoid names for shell variables (of either type) that contain special characters. Also, a variable name *cannot* begin with a number. To access the value of a variable (or, synonymously, to *evaluate* the variable) we simply prefix the variable name with a **\$** (dollar sign). We can then use the **echo** command, which, as already mentioned in the section on [executables and paths](#), simply "echoes" its arguments (see **man echo** for full details), to display the value. Here are some examples:

```
% var1=val1
% echo $var1
val1

% var1='val1'
% echo $var1
val1

% var2='val1 val2 val3'
% echo $var2
val1 val2 val3
```

You should observe that the use of the prefix **\$** to evaluate a shell variable is quite different from the evaluation mechanism found in many other programming languages (e.g. **Maple**, **C**, **MATLAB**), where use of the variable name itself generally results in evaluation (except when the name appears on the left hand side of an assignment). When writing scripts that use variables it is a common mistake to forget to use the **\$** when needed, so be extra vigilant about this point.

Also, note the usage of the single quote characters (`'`) to *delimit* the assignment value in the second and third examples: this is completely analogous to their use in our previous discussion of [alias](#) definitions. In the third example, the quotes are necessary since the value being assigned contains whitespace. In the second example, the quotes aren't needed, but they don't hurt either (i.e. we can view their inclusion as a bit of defensive programming). It is also important to understand that the quotes themselves do *not* become part of the assigned value. Finally, the double quote character (`"`) can also be used as a delimiter in variable assignment, but as we will discuss in the section on [using quotes](#), depending on the string that is enclosed in quotes, the ultimate value that is assigned to the variable can differ from that which would be obtained by using single quotes.

For us, a major use of local variables will be in the context of [bash programming](#) (writing **bash** scripts), as we will see below.

Environment Variables: As mentioned above, `bash` uses another type of variable---called an *environment variable*---which is often used for communication *between* the shell and other processes (possibly another shell, which does not necessarily have to be **bash**). To see a list of all currently defined environment variables, use the **env** command:

```
% env

TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=dd7960f4980905b6f1adf1bb4a82e093-1251778554.439092-1315938975
SSH_CLIENT=174.6.42.200 44784 22
SSH_TTY=/dev/pts/22
USER=choptuik
MAIL=/var/mail/choptuik
.
.
.
```

or the **printenv** command:

```
% printenv
```

```
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=dd7960f4980905b6f1adf1bb4a82e093-1251778554.439092-1315938975
SSH_CLIENT=174.6.42.200 44784 22
SSH_TTY=/dev/pts/22
USER=choptuik
MAIL=/var/mail/choptuik
:
:
```

and to display the values of one or more specific environment variables, supply the variable name(s) to **printenv**, or use the **echo** command in conjunction with the **\$** evaluation mechanism:

```
% printenv PWD
/home/choptuik
```

```
% printenv HOME PATH
/home/choptuik
./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

```
% echo $LOGNAME
choptuik
```

```
% echo $SHELL $USER
/bin/bash choptuik
```

Many environment variables are automatically defined whenever **bash** starts; a list of a few of these is given below. If you want to define your own environment variable, **MYENV**, say, and set its (initial) value, use the syntax

```
% export MYENV='value'
% printenv MYENV
value
```

The **export** keyword tells the shell that you are defining an environment variable, and not a local variable, but otherwise the assignment syntax is identical to that of local variables. It is conventional, but not mandatory, to use all-uppercase names for environment variables. Also, once an environment variable is defined, and you wish to change its value, it is not necessary to use the **export** keyword again:

```
% MYENV='newvalue'; printenv MYENV
newvalue
```

Note that a key aspect of the global nature of environment variables is that they (and their values) are "inherited" by any shell that is executed within a running **bash** (which includes the shells that are always started whenever a **bash** script is executed). Thus, if after having executed the assignments above, I now start a new shell, the environment variable **MYENV** will be defined with its expected value:

```
% bash
A new shell starts ...
% printenv MYENV
newvalue
```

Following is a list of a few of the environment variables that are generally predefined and/or redefined as necessary by **bash**:

- **PATH**: Stores the current list of directories that are searched for commands (executables).
- **HOME**: Stores the user's home directory;

```
% cd $HOME/dir1
```

is equivalent to

```
% cd ~/dir1
```

- **PWD**: Stores the current working directory.
- **USER**: Stores the name of the user account (login name)
- **LOGNAME**: Same as **USER**, but more generically available, so preferred in scripts.

Note that some environment variables, such as **PATH**, are often modified by the user (typically in a startup file).

Others, such as **HOME** and **PWD**, should not be altered from their system-defined values, for reasons that will hopefully be clear to you.

Using bash Pattern Matching: **bash** provides facilities that allow you to concisely refer to one or more files whose names match a given *pattern*. The process of translating patterns to actual filenames is known as *filename expansion* or *globbing*. Patterns are constructed using plain text strings and the following constructs, known as *wildcards*

- ? Matches any single character
- * Matches any string of characters (including no characters)
- [a-z] (Example) Matches any single character contained in the specified range (the match set)---in this case lower-case 'a' through lower-case 'z'
- [^a-z] (Example) Matches any single character *not* contained in the specified range

(*Aside:* Note that in **bash**, what characters are included in a range can depend on the setting of certain environment variables. In particular, on **hyper** things are set up so that while the range **[a-c]** is equivalent to **[abc]**, the range **[A-C]** is equivalent to **[aAbBcC]**. The same settings (environment variables) that controls this behaviour also controls the sorting order of filenames when commands such as **ls** are invoked (so that the files appear in an order which is case-insensitive on **hyper**. If you want the sort order to be case-sensitive, you can do so by putting the following line in your **~/.bashrc**

```
% export LC_COLLATE=C
```

end of aside)

Continuing, match sets may also be specified explicitly, as in

```
[02468]
```

Examples:

```
ls ??
```

lists all regular (not hidden) files and directories whose names contain precisely two characters.

```
cp a* /tmp
```

copies all files whose name begins with 'a' to the temporary directory **/tmp**.

```
mv *.f ../newdir
```

moves all files whose names end with '.f' to directory **../newdir**. Note that the command

```
mv *.f *.for
```

will *not* rename all files ending with '.f' to files with the same prefixes, but ending in '.for', as is the case on some other operating systems. This is easily understood by noting that expansion occurs *before* the final argument list is passed along to the **mv** command. If there aren't any '.for' files in the working directory, ***.for** will expand to *nothing* and the last command will be identical to

```
mv *.f
```

which is not at all what was intended.

The bash History

Important note: This section has been extensively edited since I distributed hard copy of the Sep 10 version of these notes. The old version was based on a discussion of the history and event handling mechanisms in the **tcs**h (the shell I previously used in my computational physics courses), and I incorrectly thought that the mechanisms were (nearly) identical in the two shells. They are *not*, so some of the information in the old version is, ahem, *wrong*. In addition, for various reasons, not least since **bash** allows you to scroll back and forth through the command history, and to easily edit the command line, much of the material that was previously here is largely irrelevant for the purposes of this course.

The key thing to note is that **bash** maintains a numbered *history* of previously entered command lines. Type

```
% history
```

(which I personally alias as **hi**) to view your command-line history. The number of commands maintained in the history list can be controlled using the environment variable, **HISTFILESIZE**, which is set to 1000 in the default course **.bashrc** file, and which you can easily change as needed. Also note that your history generally persists from login to login---commands are stored in a hidden file **~/.bash_history**---but because you can have multiple terminals executing simultaneously, the precise way that **~/.bash_history** gets updated must be considered an advanced topic. You can get some sense for this by noting that if you type different sets of commands concurrently in two or more terminals, and then type **history** in each, you will see that the output varies from terminal to terminal (as it should!) And yet there is at any given time a *unique* **~/.bash_history** file. Enough said!!

Standard Input, Standard Output and Standard Error: A typical Unix command (process, program, job, task, application) reads some input, performs some operations on, or depending on, the input, then produces some output. It proves to be extremely powerful to be able to write programs (including **bash** scripts) that read and write their input and output from "standard" locations. Thus, Unix defines the notions of

- *standard input*: default source of input
- *standard output*: default destination of output
- *standard error*: default destination for error messages and diagnostics

Many Unix commands are designed so that, unless specified otherwise, input is taken from standard input (or *stdin*), and output is written on standard output (or *stdout*). Normally, both *stdin* and *stdout* are attached to the terminal. The **cat** command with no arguments provides a canonical example (see **man cat** if you can't understand the example):

```
% cat
foo
foo
bar
bar
^D
```

Here, **cat** reads lines from *stdin* (the terminal) and writes those lines to *stdout* (also the terminal) so that every line you type is "echoed" by the command. A command that reads from *stdin* and writes to *stdout* is known as a *filter*.

Input and Output Redirection: The power and flexibility of the *stdin/stdout* mechanism becomes apparent when we consider the operations of input and output redirection that are implemented in **bash** (and many other shells). As the name suggests, redirection means that *stdin* and/or *stdout* are associated with some source/sink other than the terminal.

Input Redirection is accomplished using the **<** (less-than) character, followed by the name of a file from which the input is to be extracted. Thus the command-line

```
% cat < input_to_cat
```

causes the contents of the file **input_to_cat** to be used as input to the **cat** command. In this case, the effect is exactly the same as if

```
% cat input_to_cat
```

had been entered

Output Redirection is accomplished using the **>** (greater than) character, again followed by the name of a file into which the (standard) output of the command is to be directed. Thus

```
% cat > output_from_cat
```

will cause **cat** to read lines from the terminal (*stdin is not* redirected in this case) and copy them into the file **output_from_cat**. Care must be exercised in using output redirection since one of the first things that will happen in the above example is that the file **output_from_cat** will be clobbered. If the shell option **noclobber** has been set using

```
set -o noclobber
```

which is highly recommended for novices (and included in **~/.bashrc**), then output will not be allowed to be redirected to an existing file. Thus, in the above example, and assuming that **noclobber** is set, if **output_from_cat** already existed, the shell would respond as follows:

```
% cat > output_from_cat
output_from_cat: File exists
```

and the command would be aborted.

The standard output from a command can also be *appended* to a file using the two-character sequence >> (no intervening spaces). Thus

```
% cat >> existing_file
```

will append lines typed at the terminal to the end of **existing_file**.

From time to time it is convenient to be able to "throw away" the standard output of a command. Unix systems have a special file called **/dev/null** that is ideally suited for this purpose. Output redirection to this file, as in:

```
verbose_command > /dev/null
```

will result in the stdout from the command disappearing without a trace.

Pipes: Part of the "Unix programming philosophy" is to keep input and output to and from commands in "machine-readable" form: this usually means keeping the input and output simple, structured and devoid of extraneous information which, while informative to humans, is likely to be a nuisance for other programs. Thus, rather than writing a command that produces output such as:

```
% pgm_wrong
Time = 0.0 seconds Force = 6.0 Newtons
Time = 1.0 seconds Force = 6.1 Newtons
Time = 2.0 seconds Force = 6.2 Newtons
```

we write one that produces

```
% pgm_right
0.0 6.0
1.0 6.1
2.0 6.2
```

The advantage of this approach is that it is then often possible to combine commands (programs) on the command-line so that the standard output from one command is fed directly into the standard input of another. In this case we say that the output of the first command is *piped* into the input of the second. Here's an example:

```
% ls -1 | wc
10      10      82
```

The **-1** option to **ls** tells **ls** to list regular files and directories one per line. The command **wc** (for word count) when invoked with no arguments, reads stdin until EOF is encountered and then prints three numbers: (1) the total number of lines in the input (2) the total number of words in the input and (3) the total number of characters in the input (in this case, 82). The pipe symbol "|" tells the shell to connect the standard output of **ls** to the standard input of **wc**. The entire **ls -1 | wc** construct is known as a *pipeline*, and in this case, the first number (10) that appears on the standard output is simply the *number* of regular files and directories in the current directory.

Pipelines can be made as long as desired, and once you know a few Unix commands, and have mastered the basics of the **bash** history mechanism, you can easily accomplish some fairly sophisticated tasks by interactively building up multi-stage pipelines.

Regular Expressions and grep: Regular expressions may be formally defined as those character strings that are recognized (accepted) by *finite state automata*. If you haven't studied automata theory, this definition won't be of much use, so for our purposes we will define regular expressions as specifications for rather general *patterns* that we will wish to detect, usually in the contents of files. Although there are similarities in the Unix specification of regular expressions to **bash** wildcards (see above), there are important differences as well, so be careful. We begin with regular expressions that match a single character:

```
a      (Example) Matches 'a', any character other than
       the special characters: . * [ ] \ ^ or $ may be
       used as is

\<*     (Example) Matches the single character '*'.
       Note that \' is the "backslash" character. A
       backslash may be used to "escape" any of the
       special characters listed above
       (including backslash itself)

.      Matches ANY single character.

[abc]  (Example) Matches any one of 'a', 'b' or 'c'.

[^abc] (Example) Matches any character that ISN'T an
       'a', 'b' or 'c'.
```


[a-z] (Example) Matches any character in the inclusive range 'a' through 'z'.
[^a-z] (Example) Matches any character NOT in the inclusive range 'a' through 'z'.
^ Matches the beginning of a line.
\$ Matches the end of a line.

Multiple-character regular expressions may then be built up as follows:

ahgfh (Example) Matches the string 'ahgfh'. Any string of specific characters (including escaped special characters) may be specified in this fashion.
a* (Example) Matches zero or more occurrences of the character 'a'. Any single character expression (except start and end of line) followed by a '*' will match zero or more occurrences of that particular sequence.
. * Matches an arbitrary string of characters.

All of this is may be a bit confusing, so it is best to consider the use of regular expressions in the context of the Unix **grep** command.

grep

grep (which loosely stands for (g)lobal search for (r)egular (e)xpression with (p)rint) has the following general syntax:

```
grep [options] regular_expression [file1 file2 ...]
```

Note that only the *regular_expression* argument is required. Thus

```
% grep the
```

will read lines from stdin (normally the terminal) and echo only those lines that contain the string 'the'. If one or more file arguments are supplied along with the regular expression, then **grep** will search those files for lines matching the regular expression, and print the matching lines to standard output (again, normally the terminal). Thus

```
% grep the *
```

will print all the lines of all the regular files in the working directory that contain the string 'the'.

Some of the options to **grep** are worth mentioning here. The first is **-i** which tells grep to ignore case when pattern-matching. Thus

```
% grep -i the text
```

will print all lines of the file **text** that contain 'the' or 'The' or 'tHe' etc. Second, the **-v** option instructs grep to print all lines that *do not* match the pattern; thus

```
% grep -v the text
```

will print all lines of **text** that *do not* contain the string 'the'. Finally, the **-n** option tells grep to include a line number at the beginning of each line printed. Thus

```
% grep -in the text
```

will print, with line numbers, all lines of the file **text** that contain the string 'the', 'The', 'tHe' etc. Note that multiple options can be specified with a *single -* followed by a string of option letters with no intervening blanks. Most Unix commands allow this syntax for providing several options.

Here are a few slightly more complicated examples. Note that when supplying a regular expression that contains characters such as '*', '?', '[', '!' ..., that are special to the shell, the regular expression should be surrounded by single quotes to prevent shell interpretation of the shell characters. In fact, you won't go wrong by *always* enclosing the regular expression in single quotes.

```
% grep '^.....$' file1
```

prints all lines of **file1** that contain exactly 5 characters (not counting the "newline" at the end of each line):

```
% grep 'a' file1 | grep 'b'
```

prints all lines of **file1** that contain at least one 'a' *and* one 'b'. (Note the use of the pipe to stream the stdout from the first grep into the stdin of the second.)

```
% grep -v '^#' input > output
```

extracts all lines from file **input** that *do not* have a '#' in the first column and writes them to file **output**.

Pattern matching (searching for strings) using regular expressions is a powerful concept, but one that can be made even more useful with certain extensions. Many of these extensions are implemented in a relative of **grep** known as **egrep**. See the man page for **egrep** if you are interested.

Using Quotes (' ', " ", and ` `): Most shells, including **bash**, use the three different types of quotes found on a standard keyboard

```
' ' -> Known as forward quotes, single quotes, quotes
" " -> Known as double quotes
` ` -> Known as backward quotes, back-quotes
```

for distinct purposes.

Forward quotes: ' ' We have already encountered several examples of the use of forward quotes that inhibit shell evaluation of *any and all* special characters and/or constructs. Here's an example:

```
% a=100
% echo $a
100

% b=$a
% echo $b
100

% b='$a'
% echo $b
$a
```

Note how in the final assignment, **b='\$a'**, the **\$a** is protected from evaluation by the single quotes. Single quotes are commonly used to assign a shell variable a value that contains whitespace, or to protect command arguments that contain characters special to the shell (see the discussion of **grep** for an example).

Double quotes: " " Double quotes function in much the same way as forward quotes, except that the shell "looks inside" them and evaluates (1) any references to the values of shell variables, and (2) anything within back-quotes (see below). Example:

```
% a=100
% echo $a
100

% string="The value of a is $a"
% echo $string
The value of a is 100
```

Backward quotes: ` ` The shell uses back-quotes to provide a powerful mechanism for capturing the standard output of a Unix command (or, more generally, a sequence of Unix commands) as a string that can then be assigned to a shell variable or used as an argument to another command. Specifically, when the shell encounters a string enclosed in back-quotes, it attempts to evaluate the string as a Unix command, precisely as if the string had been entered at a shell prompt, and returns the standard output of the command as a string. In effect, the output of the command is substituted for the string and the enclosing back-quotes. Here are a few simple examples:

```
% date
Tue Sep  1 16:50:10 PDT 2009

% thedate=`date`
% echo $thedate
Tue Sep  1 16:50:15 PDT 2009

% which true
/bin/true
```

```
% file `which true`  
/bin/true: ELF 64-bit LSB executable, x86-64, ...
```

```
% file `which true` `which false`  
/bin/true: ELF 64-bit LSB executable, x86-64, ...  
/bin/false: ELF 64-bit LSB executable, x86-64, ...
```

Note that the **file** command attempts to guess what type of contents its arguments (which should be files) contain and **which** reports full path names for commands that are supplied as arguments. Observe that in the last example, multiple back-quoting constructs are used on a single command line.

Finally, here's an example illustrating that back-quote substitution is *enabled* for strings within double quotes, but *disabled* for strings within single quotes:

```
% var1="The current date is `date`"  
% echo $var1  
The current date is Tue Sep 1 16:50:40 PDT 2009
```

```
% var2='The current date is `date`'  
% echo $var2  
The current date is `date`
```

Job Control: Unix is a multi-tasking operating system: at any given time, the system is effectively running many distinct processes (commands) simultaneously (of course, if the machine only has one CPU, only one process can run at a specific time, so this simultaneity is somewhat of an illusion). Even within a single shell, it is possible to run several different commands at the same time. *Job control* refers to the shell facilities for managing how these different processes are run. It should be noted that job control is arguably less important in the current age of windowing systems than it used to be, since one can now simply use multiple shell windows to manage several concurrently running tasks.

Commands issued from the command-line normally run in the *foreground*. This generally means that the command "takes over" standard input and standard output (the terminal), and, in particular, the command must complete before you can type additional commands to the shell. If, however, the command line is terminated with an ampersand: **&**, the job is run in the *background* and you can *immediately* type new commands while the command executes. Example:

```
% grep the huge_file > grep_output &  
[1] 1299
```

In this example, the shell responds with a '[1]' that identifies the task at the shell level, and a '1299' (the process id) that identifies the task at the system level. You can continue to type commands while the **grep** job runs in the background. At some point **grep** will finish, and the next time you type 'Enter' (or 'Return'), the shell will inform you that the job has completed:

```
[1]+ Done grep the huge_file > grep_output
```

The following sequence illustrates another way to run the same job in the background:

```
% grep the huge_file > grep_output  
^Z  
[1]+ Stopped grep the huge_file > grep_output  
% bg  
[1]+ grep the huge_file > grep_output &
```

Here, typing **^Z** while the command is running in the foreground stops (suspends) the job, the shell command **bg** restarts it in the background. You can see which jobs are running or stopped by using the shell **jobs** command.

```
% jobs  
[1] + Stopped grep the huge_file > grep_output  
[2] Running other_command
```

Use

```
% fg %1
```

to have the job labeled '[1]' (that may either be stopped or running in the background), run in the foreground. You can **kill** (terminate) a job using its job number (%1, %2, etc.)

```
% kill %1  
[1] Terminated grep the huge_file > grep_output
```

You can also **kill** a job using its process ID (PID), which you can obtain using the Unix **ps** command. See the **man**

pages for **ps** and **kill** for more details.

On many Unix systems, including Linux, there is a **killall** command, which allows you to kill processes by name. Finally, the shell will complain if you try to **logout** or **exit** the shell when one or more jobs are stopped. Either explicitly kill the jobs (or let them finish up if that's appropriate) or type **logout** or **exit** again to ignore the warning, kill all stopped jobs, and exit.

Another useful, though Linux-specific, command is **ps tree**, which shows processes currently running on the host machine in the form of a tree. If you want to limit the output to your own processes (and not, for example, **root's**), use

```
% pstree -u your_userid
```

BASIC SHELL PROGRAMMING

For the novice user a Unix shell can be viewed primarily as a command interpreter. However, shells are actually fully functional programming languages and it is extremely useful to know at least a little about shell programming, also known as writing *shell scripts*, for the following reasons (not an exhaustive list!):

1. Scripts can be used to customize or extend Unix commands in a more powerful and robust fashion than the aliasing mechanism discussed above.
2. Scripts can be used to automate sequences of Unix commands, with the possibility of changing one or more of the arguments to one or more of the commands. If you find yourself frequently typing the same sequence of commands, it takes very little time to create a script to accomplish the task, after which the execution of a *single* command does the trick. This has the added bonus that the script *per se* provides documentation for the job you are doing.
3. Many tasks that are cumbersome to perform in the context of a general purpose programming language, such as **Java**, **C** or **Fortran**, are easy to accomplish using a script. This particularly applies to issues involving file and directory manipulation, or the processing of output from a number of programs.

Time constraints preclude anything but a basic overview of **bash** programming; if you wish to become a wizard of this particular craft, you might want to consult the classic text, *The UNIX Programming Environment*, by Kernighan and Pike, cited in the following as reference [1]. In addition, there is plenty of information to be found about the subject online (see the representative links at the end of this document, for example, which are also available via the [Course Resources](#) page). Finally, should you find yourself in need of *complex* scripts, you may wish to consider learning/using **perl**, which is an extremely powerful scripting language that has become very popular in the Unix community over the past couple of decades (the **python** language is another good choice in this regard).

We start with a very simple example. Consider the problem of "swapping" the names of two files, which arises more often in practice than one might expect, and which cannot be accomplished with a standard Unix command. Assuming that no file **t** exists in the working directory, the command sequence

```
% mv a t
% mv b a
% mv t b
```

will exchange the names of files **a** and **b**. Building on this sequence, here's a script called **swap** that, naturally enough, "swaps" the names of an *arbitrary* pair of files:

```
#!/bin/bash

# Bare-bones script to swap names of two files

# Usage: swap file1 file2

mv $1 t
mv $2 $1
mv t $2
```

The first line of the script

```
#!/bin/bash
```

is an important bit of Unix magic that tells the shell that when the name of the file containing the script is used as a command, the shell should start up a *new* shell (in this case another **bash**) and execute the remaining contents of the script in the context of that new shell. *Every* **bash** script that you write should start with this incantation.

IMPORTANT!! When the new **bash** associated with the execution of *any* script starts, *it does NOT execute the commands in either `~/.profile` or `~/.bashrc`*. This means, in particular, that none of the aliases that are defined through execution of `~/.bashrc` (including the indirect definitions that may be made via a **source** `~/.aliases` or equivalent) will be active during the execution of the script, unless you redefine them within the script itself. The

same applies to **bash** options such as **noclobber**. So be very careful when using commands such as **rm**, **cp**, **mv** etc. as well as output redirection, since the commands will be the "bare-bones" versions, and will *not* prompt for confirmation in case an existing file will be overwritten. Similarly, output redirection *will* clobber existing files (unless you include **set -o noclobber** somewhere before the first use of output redirection).

*In this regard it is impossible to overstate the importance that you develop the habit of making back-up copies of **any** files that have value to you, **before** you start making significant modifications to them, or feeding them to a script that could potentially and unintentionally change them. This applies not only in the context of writing scripts, but anytime that you are about to modify a file that has taken you non-trivial effort to create!*

Continuing with our dissection of the script, lines that begin with a hash ("number sign") # (excluding the magic first line) such as

```
# Bare-bones script to swap names of two files
```

```
# Usage: swap file1 file2
```

are comments, and are ignored by the shell.

The final three lines of the script

```
mv $1 t
mv $2 $1
mv t $2
```

do all the work. The constructs **\$1** and **\$2** evaluate to the first and second arguments, respectively, which are supplied to the script. In general, one can access the first nine arguments of a script using **\$1**, **\$2**, ..., **\$9**, and, if more than nine arguments need to be parsed (!), using **\${10}**, **\${11}**, etc. If a specific argument is missing, the corresponding construct will evaluate to the null string, i.e. to "nothing".

Having created a file called **swap** containing the above lines, I set execute permission on the file with the **chmod** command

```
% chmod a+x swap
% ls -l swap
-rwxr-xr-x 1 phys210 public 116 2009-09-07 16:32 swap*
```

and the script is ready to use:

```
% ls
f1 f2 swap*
% cat f1
This is the first file.
% cat f2
This is the second file.
% swap f1 f2
% cat f1
This is the second file.
% cat f2
This is the first file.
```

When developing and debugging a shell program, it is often very useful to enable "tracing" of the script. This is done by adding the **-x** option to the header line:

```
#!/bin/bash -x
```

Having made this modification, I now see the following output when I invoke **swap** a second time:

```
% swap f1 f2
+ mv f1 t
+ mv f2 f1
+ mv t f2
```

Note how each command in the script is echoed to standard error (with a **+** prepended) as it is executed. Again, observe that the **mv** command used in this instance is the "bare bones" version since any aliases that I have defined via **~/.bashrc** for an interactive **bash** will *not* be in effect while the script executes.

Although **swap** as coded above is reasonably functional, it is not very robust and can potentially generate undesired "side-effects" if used incorrectly. Observe, for example, what happens when the script is invoked without any arguments (tracing has now been disabled by removing the **-x** option in the header)

```
% swap
```

```
mv: missing file argument
Try `mv --help' for more information.
mv: missing file arguments
Try `mv --help' for more information.
mv: missing file argument
Try `mv --help' for more information.
```

or, worse, with one argument

```
% swap f1
mv: missing file argument
Try `mv --help' for more information.
mv: missing file argument
Try `mv --help' for more information.
% ls
f2 swap* t
```

Here's a second version of **swap** that fixes several of the shortcomings of the naive version, and that also illustrates many additional shell programming features:

```
#!/bin/bash

# Improved version of script to swap names of two files

# Set shell variable 'P' to name of script
P=`basename $0`

# Set shell variable 't' to name of temporary file
t=.swap.tempfile.3141

# Usage function
usage () {
cat << END
usage: $P file1 file2

        Swaps filenames of file1 and file2
END
exit 1
}

# Function that is invoked if temporary file already exists
t_exists () {
cat << END
$P: Temporary file '$t' exists.
$P: Remove it explicitly before executing this script.

/bin/rm -f $t
END
exit 1
}

# Function that checks that its (first) argument is an
# existing file
check_file () {
if [ ! -f $1 ]; then
    echo "$P: File '$1' does not exist"
    error="yes"
fi
}

# Argument parsing---script requires exactly 2 arguments
case $# in
2) file1=$1; file2=$2 ;;
*) usage;;
esac

# Check that the arguments refer to existing files
check_file $1
check_file $2

# Bail out if either or both arguments are invalid
test "X${error}" = X || exit 1
```

```
# Ensure that temporary file doesn't already exist
test -f $t && t_exists

# Do the swap
mv $file1 $t
mv $file2 $file1
mv $t      $file2

# Normal exit, return 'success' exit status
exit 0
```

Let us examine this new version of **swap** in detail.

As the comment indicates, the command

```
# Set shell variable 'P' to name of script
P=`basename $0`
```

sets the shell variable **P** to the filename of the script, i.e. to **swap** in this case. This happens as follows. First, **\$0** is a special shell-script variable that always evaluates to the invocation name of the script---i.e. what the user actually typed in order to execute the script. Second, as **man** tells us, the **basename** command deletes any prefix ending in **/** from its argument and prints the result on the standard output. Third, the backquotes around the **basename** invocation capture the standard output of the command, which is then assigned to the shell variable **P** via the assignment statement.

We use **basename** here so that if someone invokes our script using its full path name, perhaps

```
% /home/phys210/shellpgm/ex2/swap f1 f2
```

the shell variable **P** will still be assigned the value **swap**. The value of **P** is subsequently used in diagnostic messages, to make the origin of the messages clear to the user. Use of this mechanism can save some typing if one is writing a script that prints many such messages. In addition, if the script is subsequently used as a basis for a *new* shell program, a minimum of changes (perhaps none) are necessary in order that the new script output the "correct" diagnostics.

The next assignment sets the shell variable **t** to the name of a temporary file that, under normal circumstances, should never exist in the directory in which **swap** is executed. This isn't the most bullet-proof of strategies, but it's better than using **t** itself for the name for the temporary file!

```
# Set shell variable 't' to name of temporary file
t=.swap.tempfile.3141
```

The next section of code defines a **shell function**, called **usage**, which can be invoked from anywhere in the script. When called, the function will print a message to standard output informing the user of the proper usage of the command, and then exit (stop execution of the script).

```
# Usage function
usage () {
cat << END
usage: $P file1 file2

        Swaps filenames of file1 and file2
END
exit 1
}
```

The general form of a function definition is

```
routinename () {
    command
    command
    ...
}
```

The parentheses pair after *routinename* tells the shell that a function is being defined, while the braces enclose the body of the function.

Within the **usage** routine appears the construct

```
cat << END
...
END
```

known as a "here document". Here documents can be used anywhere in a script to provide "in-place" input for the standard input of a command. You can refer to the **man** page on **bash** for full details, but the basic idea and mechanics are quite simple. To provide "in-place" input to an arbitrary command, append `<< END` after the command name, any arguments to the command, and any output redirection directives. There can be whitespace before and after the token **END**. Subsequent lines are then the standard input to the command. A line that *exactly* matches the string **END** (i.e. `grep '^END$'` succeeds) signals end-of-file (i.e. the end of the here document), so be sure you have such a line in your script!.

IMPORTANT!! Note that this means that *there can be NO whitespace before or after the second occurrence of END*. You need to be extremely careful about this point since it is very easy to accidentally add a space or two after the end-of-file token, and quite difficult to notice that the extra space is there. If there *is* extraneous white space, **bash** is likely to view everything until the end of the script itself as the standard input to the command being fed with the here document!

Finally, note that the string **END** is arbitrary; you can use essentially any string you wish as long as you use the identical string in both contexts. **END** is simply my convention.

An interesting and useful feature of here-documents is that they are partially interpreted by the shell *before* being fed into their destination command. In particular, shell-variable-evaluations

```
$var
```

are executed, as are

```
`command [arguments]`
```

constructs. Thus, when the **usage** function is executed, the message

```
usage: swap file1 file2
```

```
    Swaps filenames of file1 and file2
```

will appear on standard output, after which the execution of

```
exit 1
```

will return control to the invoking shell. Here, the argument to the **exit** command is an exit code indicating a completion status for the script. Since there is generally only one way for a command to succeed, but often many ways it can fail, a exit status of **0** indicates success in Unix, while any non-zero value (**1** in this case), indicates failure.

All Unix commands return such codes (scripts that terminate without an explicit **exit**, implicitly return success to the invoking shell) and they can be used in the context of shell-control structures such as **if**, **while** and **until** statements.

The function **t_exists** is very similar in construction to **usage**, and is used in the unlikely event that a file named **.swap.tempfile.3141** does exist in the directory in which the script is invoked.

```
# Function that is invoked if temporary file already exists
t_exists () {
cat << END
$P: Temporary file '$t' exists.
$P: Remove it explicitly before executing this script.

/bin/rm -f $t
END
exit 1
}
```

Function **check_file** illustrates the use of function arguments, as well as the shell **if** statement.

```
# Function that checks that its (first) argument is an
# existing file
check_file () {
if [ ! -f $1 ]; then
    echo "$P: File '$1' does not exist"
    error="yes"
fi
}
```

As with arguments to the script itself, function arguments are accessed *positionally*, via **\$1**, **\$2**, Note, then, that the evaluation of **\$1**, for example, depends crucially on context (or scope): within a function, **\$1** evaluates to the first argument to the routine, while outside of any function it evaluates to the first argument to the script.

For our purposes, a suitably general form of the shell **if** statement is

```
if command a; then
  commands 1
elif command b; then
  commands 2
elif command c; then
  commands 3
  ...
else
  commands n
fi
```

All clauses apart from the first are optional, as is apparent from the **if** statement in the **check_file** routine. The evaluation of the **if** statement begins with the execution of *command a*. If this command succeeds (returns exit status 0), then *commands 1* are executed (commands must appear on separate lines, or be separated by semicolons) and control then passes to the command following the end of the **if** statement (i.e. after the **fi** token). Otherwise, *command b* is executed; if it succeeds, *commands 2* are performed, otherwise *command c* is executed, and so on.

The **if** statement in our **check_file** routine

```
if [ ! -f $1 ]; then
  echo "$P: File '$1' does not exist"
  error="yes"
fi
```

uses the Unix **test** command, for which **[** is essentially an alias (the **]** is "syntactic-sugar" and does nothing but make the expression "look right"). Thus an equivalent form is

```
if test ! -f $1; then
  echo "$P: File '$1' does not exist"
  error="yes"
fi
```

test accepts a general expression *expr* as an argument, evaluates *expr* and, if its value is true, sets a zero exit status (success); otherwise, a non-zero exit status (failure) is set. **test** accepts many different options for performing a variety of tests on files and directories, and implements a fairly complete set of logical operations such as negation, or, and, tests for string equality/non-equality, integer equal-to, greater-than, less-than etc.; see **man test** for full details.

In the current case, the **-f \$1** option returns true if the first argument to the routine is an existing regular file (i.e. not a directory or other type of special file). The **!** is the negation operator, so the overall **test** command returns success (true) if the first argument is *not* an existing regular file.

The next section of code introduces the shell **case** statement:

```
# Argument parsing---script requires exactly 2 arguments
case $# in
2) file1=$1; file2=$2 ;;
*) usage;;
esac
```

A general **case** statement looks like

```
case word in
pattern) commands ;;
pattern) commands ;;
...
esac
```

Starting from the top, and using essentially the same pattern-matching rules used for filename matching, the **case** statement compares *word* to each *pattern* in turn, until it finds a match. When a match is found, the corresponding commands (and only those commands) are executed, after which control passes to the statement following the end of the **case** statement (i.e. after the **esac** token). Note that the commands associated with each case must be terminated with a double semi-colon.

In our current example, we match on the built-in shell variable **\$#**, which evaluates to the number of arguments that were supplied to the shell. The first set of actions

```
2) file1=$1; file2=$2 ;;
```

is evaluated if precisely two arguments have been supplied. If the script has been invoked with anything *but* two arguments, `$#` is then matched against `*`, which will *always* succeed; i.e.

```
*) usage ;;
```

serves as a "default" case, and the **usage** function will thus be called if we've used an incorrect number of arguments.

Using the **check_file** function, the script then ensures that each argument names an existing regular file:

```
# Check that the arguments refer to existing files
check_file $file1
check_file $file2
```

Note that if either or both of the checks fail, then the shell variable **error** (all variables are global in a shell script unless explicitly declared **local**, see **man bash** for more information) will be set to **yes**. The calls to **check_file** are followed by a command list that tests whether **error** has been set, and exits the script if it has:

```
# Bail out if either or both arguments are invalid
test "X${error}" = X || exit 1
```

The expression

```
"X${error}" = X
```

illustrates a little shell trick that tests whether a shell variable has been defined. If **error** has been set to **yes** by **check_file**, then **"X\${error}"** evaluates to **Xyes**; otherwise it evaluates to **X**. The binary operator **||** (double pipe) can be used between any two Unix commands (or, more generally, pipelines):

```
command 1 || command 2
```

and has the following semantics: *command 1* is executed, and *if and only if* the command *fails* (returns a non-zero exit status), *command 2* is executed. Thus, the sequence is equivalent to

```
if [ ! command 1 ]; then
    command 2
fi
```

Similarly, the next piece of the script

```
# Ensure that temporary file doesn't already exist
test -f $t && t_exists
```

illustrates the use of the binary operator **&&** (double ampersand), which also can be used between any two commands:

```
command 1 && command 2
```

In this case *command 1* is executed, and *if and only if* the command *succeeds* (returns a 0 exit status), *command 2* is executed. Thus, an equivalent form is

```
if [ command 1 ]; then
    command 2
fi
```

In the current example, if the temporary file **.swap.temp.3141** *does* exist, the function **t_exists** is called to print the diagnostic message and exit.

Finally, if we've made it past all of the error-checking, it's time to actually swap the filenames, and have the script return a "success" exit status to the invoking environment:

```
# Do the swap
mv $file1 $t
mv $file2 $file1
mv $t     $file2

# Normal exit, return 'success' code
exit 0
```

We can now test our improved version of **swap**, exercising in particular all of the error-checking features that have been incorporated. Here again is a contents-listing of the directory containing the script:

```
% ls
f1 f2 swap*
% more f1 f2
:::::::::::::
f1
:::::::::::::
This is the first file.
:::::::::::::
f2
:::::::::::::
This is the second file.
```

We start with a no-argument invocation:

```
% swap
usage: swap file1 file2

        Swaps filenames of file1 and file2
```

followed by single-argument execution:

```
% swap f1
usage: swap file1 file2

        Swaps filenames of file1 and file2
```

In both cases **swap** dutifully prints the usage message to standard output as desired.

We now invoke **swap** in a "normal" fashion, and verify that it is working properly:

```
% swap f1 f2
% more f1 f2
:::::::::::::
f1
:::::::::::::
This is the second file.
:::::::::::::
f2
:::::::::::::
This is the first file.
```

Supplying **swap** with two arguments that are *not* names of files in the working directory results in appropriate error messages:

```
% swap a1 a2
swap: File 'a1' does not exist
swap: File 'a2' does not exist
```

as does an invocation where *one* of the arguments is invalid:

```
% swap a1 f2
swap: File 'a1' does not exist
```

Finally, after (perversely) creating **.swap.tempfile.3141** using the **touch** command (**touch filename** creates the (empty) file *filename* if it does not exist, and changes its time of last modification to the current time if it does),

```
% touch .swap.tempfile.3141
```

execution of **swap** with valid arguments triggers the **t_exists** routine:

```
swap: Temporary file '.swap.tempfile.3141' exists.
swap: Remove it explicitly before executing this script.
```

```
/bin/rm -f .swap.tempfile.3141
```

Removing the file as instructed, the script once again silently performs its job:

```
% /bin/rm -f .swap.tempfile.3141
% swap f1 f2
```

We will conclude our whirlwind tour of shell programming with a description of a few more control structures, some additional niceties concerning shell variable evaluation, and a glimpse at a command useful for writing scripts that

"interact" with the user.

The shell provides three structures for looping. The first is a **for** loop:

```
for var in word list; do
    commands
done
```

Here, for each word (token) in *word list*, the *commands* in the body of the loop are executed, with the shell variable *var* being set to each word in turn. As usual, an example makes the semantics clear:

```
% cat for-example
#! /bin/bash

# Illustrates shell 'for' loop

for i in foo bar 'foo bar '; do
    echo "i -> $i"
done

% for-example
i -> foo
i -> bar
i -> foo bar
```

Note that a "word" can contain whitespace if it has been quoted, as is the case for **'foo bar '**

In many instances, a for loop in a script will loop over all of the arguments supplied to the script. The built-in shell variable **\$*** evaluates to the argument list, so we can write

```
for i in $*; do
    commands
done
```

but the shell also has a shorthand for this particular case, namely:

```
for i; do
    commands
done
```

In addition to **for** iterations, there are also **while** loops:

```
while command; do
    commands
done
```

and **until** loops:

```
until command; do
    commands
done
```

For these iterations, the body of the loop is repetitively executed as long as *command* succeeds or fails, respectively.

The following table summarizes some built-in shell variables that are particularly useful for script writing [1]:

Variable	Evaluates to
\$#	number of arguments
\$*	all arguments
\$?	return value of last command
\$\$	process-id of the script

Also observe that, as intimated above in the discussion of [environment variables](#), a **bash** script inherits *all* of the environment variables (such as **\$HOME**, **\$PATH**, ...) that have been set in the invoking shell.

As shown in the next table [1], we can also use some tricks in the evaluation of shell variables to make writing scripts a little easier at times:

Expression	Evaluates to
<code>\$var</code>	value of <code>var</code> , nothing if <code>var</code> undefined
<code>\${var}</code>	same as above; useful if alphanumeric follow variable name
<code>\${var-thing}</code>	value of <code>var</code> if defined; otherwise thing ; <code>\$var</code> unchanged.
<code>\${var=thing}</code>	value of <code>var</code> if defined; otherwise thing ; if undefined <code>\$var</code> set to thing
<code>\${var+thing}</code>	thing if <code>var</code> defined; otherwise nothing
<code>\${var?message}</code>	if defined, <code>\$var</code> ; otherwise print message and exit shell.

Finally, the `read` command can be used to interactively provide input to a script. Here's an example

```
% cat read-example
#!/bin/bash

echo "Hello there! Please type in your name:"
read name
echo "Pleased to meet you, $name"

% read-example
Hello there! Please type in your name
Matthew Choptuik
Pleased to meet you, Matthew Choptuik
```

References / Additional Information

1. Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice Hall, 1984
2. [Bash Guide for Beginners](#) (Includes sections on writing scripts.)
3. [Bash Reference Manual](#)
4. [Bash Scripting Tutorial](#)
5. [Linux Shell Scripting Tutorial: A Beginner's handbook](#)
6. [Advanced Bash-Scripting Guide](#) (An extensive reference, with lots of examples, and which, despite the name, does not assume prior knowledge of scripting or programming.)