Contents

Chapter 2

The Shell

The shell is the program you will be using to interact with the system. While there are very nice looking GUIs available to perform some of the administration tasks we will be covering, the objective of this course is not to teach you how to use GUIs. In fact, after you learn the power of the shell, even if you have the option to use a GUI, you will find that doing work from the shell will be far more efficient.

2.1 The Terminal, Shell, and Command Prompt

You will often hear the terms **terminal**, **shell**, and **command prompt** used interchangeably to refer to a text based method of administering a UNIX system. While they are all involved in this type of administration, they are in fact separate entities.

2.1.1 The Terminal

The **terminal** is what is used for entering data into, and displaying data from, the computer you are connected to. In the beginning, this was a physical device connected to a computer or a mainframe, consisting of a keyboard and a simple monitor. You would type data in and it and it and it would give you the results line by line. If there was more lines of data than lines on the display, the data would be buffered and you would press a key to scroll through it.

Later, so called **intelligent terminals** such as the **VT100** where created. Unlike their predecessors, these terminals had the ability to interpret escape sequences to position the cursor and control the display.

Today, few still use a physical terminal to connect to a system. Instead, software programs called **terminal emulators** are used. These programs usually emulate the vt100 intelligent terminal (the de facto standard for terminal emulators). Terminal emulators exist for basically every operating system out there. UNIX has basic terminal emulators that start with the OS and many ter-

minal emulators for the the **X Graphics System**, including **xterm**, **Eterm**, **GNOME Terminal**, and **Konsole**. Mac OS X ships with a terminal emulator, appropriately named **Terminal**. Windows has several 3rd party emulators that can be used for remote access, including **SSH Secure Shell** and **PuTTY**.

2.1.2 The Shell & Command Prompt

The **shell** is the program that you use to interact with the operating system. After you log in through the terminal, a shell is automatically started, and remains until you log out. Unlike the terminal, which is only the device (or program) used to connect the machine, the shell is the program that you actually interact with. You give it commands via the **Command Prompt** and it processes them, passes them to the kernel for execution, then gives you the results. However, while the shell is used interactively by you as you work on the system, the shell program can also be used non-interactively as a scripting language (which will be discussed later).

It is also important to note that there isn't just *one* shell program for you to use. There are many different shells out there, each with their own feature sets. These shells generally fall into one of two families: the **C Shell** family, and the **Bourne Shell** family.

The **Bourne shell** , or **sh**, is the standard UNIX shell and most of the shell scripts written for the operating system are written in **sh**. However, the Bourne shell's derivatives, like the **Korn shell** (**ksh**), the **Z shell** (**zsh**), and the **Bourne-again shell** (**bash**) contain many more features and are often used in favor **sh**.

The **C shell** (**csh**) and its derivative, the **Tenix C Shell** (**tcsh**), have a more "C like" programming syntax that many find easier to learn at first. Unfortunately, this programming syntax, along with the syntax for many of the features that will be discussed later, is incompatible with **Bourne** syntax. Since these incompatibilities can't be resolved, you will have to choose a side in the **Shell Wars**: **Bourne** or **C**.

Which shell you use is entirely up to you. However, the commands given from now on will assume you are using the **bash** shell (or similar). A reference shell for the commands must set, and the choice to use **bash** is based on the idea that **bash** is considered a powerful programming shell, while scripting in **csh** is rumored to be hazardous to your health.

2.2 Command Structure

All commands in UNIX have the same three common elements: the command name, options, and arguments. The command name is required for all commands (obviously, or else how would the shell know what to do?). Some commands also require arguments to be specified in order for the command to make sense. For example, the **cp** command, which is used to copy files, wouldn't know what to do if you didn't also tell it what file you wanted copied and where

you wanted it copied to. Options, as the name implies, are optional. Commands have a default behavior that they will use if no options are specified, however you can specify options if you want to alter the program's behavior. This command structure is represented using the following syntax:

```
command [OPTION] ARGUMENT
```

The command is the first value on the command line, followed by required arguments (if any). Optional flags or values are specified in square braces [ ]. For example, lets analyze the syntax for the **cp** command:

```
cp [OPTION]... SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
```

The first value is the name of the command (**cp**). Note that by convention, all commands are lowercase. This is also a good time to remind you that everything in UNIX is case sensitive (for example, the files '**syllabus.txt**' and '**Syllabus.txt**' are completely different files). After the command we see, `[OPTION]...`, which tells you this is where you would enter one or more options. After options, the syntax lists two required arguments: `SOURCE`, which specifies the file you want copied, and `DEST`, which specifies what you want it copied to. The syntax for **cp** is relatively simple, so lets consider its slightly more complicated relative **scp**, which is used for coping between machines:

```
scp [-1246BCpqrv] [-c cipher] [-F ssh_config] [-i identity_file]
    [-l limit] [-o ssh_option] [-P port] [-S program]
    [[user@]host1:]file1 [...] [[user@]host2:]file2
```

The fact that this listing has 3 lines is only because there are so many options. The command is still entered on one line. Lets really break do the **scp** command syntax.

2.2.1 Command

In order to run, you must either specify the **absolute** or **relative path**, or the command must be in your **PATH**. All the commands you will need for now will be in your **PATH**, so don't worry too much about this at the moment. We will revisit this when we talk about the file system structure. In the **scp** example, the command is simply `scp`.

2.2.2 Options

By convention, all options are specified with a '-' (hyphen). Options can also be specified in any order. Additionally, options that don't take arguments can be combined into a single block of flags. For example, if you wanted to specify the `2`, `B`, and `r` options, the following are equivalent:

```
-2 -B -r  ==  -B -2 -r == -2Br == -2rB == -2B -r == -r2 -B == ...
```

The **scp** syntax shows that there are two types of options: those with required arguments, and those without (often called **flags**). If you specify an option that has an argument, you must specify the argument, or the command will fail. For example, if you wanted to change the port that **scp** connects on, you would use the `-P` option, but you must also specify the `port` argument. When you specify `-P 222`, you are telling **scp** "connect to the remote server on port 222", but if you were to only specify `-P`, you would be telling **scp** "connect using port...", then stopping your sentence. **scp** won't know what to do and will give you an error.

You may have also noticed something interesting preceding the arguments. Lets look at the destination of the file `[[user@]host2:]file2`. We see that `file2` is not in square brackets and is therefore required. However, it is preceded by optional values (actually one optional value nested in another). By default, the destination will be the server you are currently logged on to and the user you are currently logged in as. Lets see how this would work. If I wanted to just copy `syllabus` to `syllabus.old`, I would enter:

```
scp syllabus syllabus.old
```

If I wanted to copy it to the server named `conquest`, i would enter:

```
scp syllabus conquest:syllabus.old
```

and if I wanted to copy it to the server named `conquest` as the user `aoaks`, i would enter:

```
scp syllabus aoaks@conquest:syllabus.old
```

Also note how the `user` option is nested in the `server` option. Don't try to specify the `user` option without specifying the `host` option, or the command won't work. (well, it would, but then you would have a file named `aoaks@syllabus.old`, which probably isn't what you wanted)

2.2.3 Arguments

The arguments to the **scp** are `file1` and `file2`, and as mentioned before, are required. If you don't specify them, you will get an error. Also note that order of the arguments matter. The syntax says that the first non-option value (i.e. the first value that doesn't start with '-' or follow an option with a required argument) will be `file1` and the second non-option value will be `file2`. Thus, unlike options which can be specified anywhere and in any order, arguments must be specified in the correct order according to the syntax of the command.

2.2.4 Help With Commands

As you might expect, nobody will ever remember all of the options to every command on the system. That's just a waste of time. You will end up remembering the options you use all of the time and you will need to look up options

you don't use very often. To find information on the commands on your system, consult the `man` pages (manual pages). One of the commands available to you is `man`, which takes one argument: the name of the command you want to look up.

For example, if you enter `man scp`, you will get the manual pages for `scp` which will tell you what the command is for, the syntax of the command (the syntax above is directly from the `man` pages), and an explanation for every option and argument to the command. In addition to these items, the `man` pages usually also contain additional information like examples. If, by chance, there are no `man` pages for a command, most commands have a `-h` or `--help` option that will give you similar information. In short, if you need to know about a command, READ THE MANUAL.

2.3 Wildcards

The shell offers you a method to simplify what can end up being long strings or repeated values on the command line through the use of **wildcards**. These wildcards are used by the shell for **pattern matching** (which we will discuss in much greater detail in later sections). When the shell received a command with wildcards in it, it expands them and selects all values that match the pattern.

2.3.1 The ? (Question Mark)

The **?** (question mark) matches a single character, whatever it may be (letter, number, symbol, etc). For example, if you wanted to copy all of the chapter notes into the **notes/** directory, rather than typing:

```
mv chap1note.tex chap2note.tex chap3note.tex chapAnote.tex notes/
```

you can type:

```
mv chap?note.tex notes/
```

The second statement will expand the **?** to match all of the files that match the form `chap`, then any single character, then `note.tex`. If instead you had a lot of notes and used the form `chap13note.tex`, you could use:

```
mv chap??note.tex notes/
```

to match all of the notes of that form. Note that this would not match the files from the first example, because this pattern expects two characters between `chap` and `notes.tex`, and the files from the first example only had one. What if you wanted to combine both? That is the topic of the next section.

2.3.2 The * (Asterisk)

Unlike the **?**, which is used to match a single character, the ***** (asterisk) is used to match anything. It matches any number of any characters. Using the previous example of chapter notes, instead of typing:

```
mv chap1notes.tex chap23notes.tex chapAnotes.tex notes/
```

you could type:

```
mv chap*notes.tex notes/
```

this will match any files that start with `chap` and end with `notes.tex`, regardless of what is in between. Since ***** expands to match anything, if you use just the *****, you will match everything in the directory. For example, if you wanted to backup everything in a directory into a backup directory, rather than typing in possibly large number of files, you can simply type the following to copy all files:

```
cp * backup/
```

Be Careful UNIX will always assume that you know what you are doing and will not warn you when you are about to do something disastrous. For example, if you wanted to remove all of the text files in a directory, you would type something like:

```
rm *.txt
```

This will do what you want, but what if you were typing too fast and accidentally typed something like:

```
rm * .txt
```

The shell assumes you know what you are doing and will not warn you about what is about to happen. Since you placed a space between the `*` and the `.txt`, the shell will expand the `*` to match all files in the directory and delete them all. So remember to always be careful when using the `*` wildcard with potential dangerous commands.

2.3.3 Character Classes

While the ***** and **?** wildcards can be useful when you want to work with large batches of files, sometimes you need to be more restrictive in your pattern matching. This can be accomplished through the use of **character classes**, which are specified inside square brackets ([ ]). Each pair of square brackets specifies matches one character, similar to the **?**, except unlike the **?**, the square brackets limits which characters are matched by what is placed inside of them.

To match one of a set of characters, place the list of characters in the square brackets. For example, to match `a`, `g`, `u`, `w`, and `y`, you would use `[aguwy]`. To reverse this, insert a **!** (exclamation point) at the beginning of the class. Following the previous example, using `[!aguwy]` will match any character other than `a`, `g`, `u`, `w`, and `y`. To match one of a range of characters, place the first and last characters of the range around a **-** (hyphen). For example, to match, to match any character between `a` and `r`, you would use `[a-r]`. Again, to reverse this, insert a **!** at the beginning of the class. Following the previous example, using `[!a-r]` would match any character that is not between `a` and `r`.

2.3.4 Escaping

As you have seen, the shell gives special meaning to certain characters, most notably the **!**, **?**, **{**, **}**, [,], and **space**, the space character. If you want to use these characters as part of your commands, you will need to remove their special meaning. Removing the special meaning is called **escaping** and is accomplished by preceding the character with a \ (backslash). For example, if you wanted to copy the file '`my data.txt`', you would couldn't type:

```
cp my data.txt backup/
```

because the command would try to copy the files '`my`' and '`data.txt`', neither of which is the file you wanted to copy. To copy the file, you need to escape the space:

```
cp my\ data.txt backup/
```

2.3.5 Summary

Here is a quick summary of the shell escapes and their meanings:

wildcard	matches
`*`	zero of more characters
`?`	single character
`[abcde]`	exactly one character listed
`[a-e]`	exactly one character in given range
`[!abcde]`	any character not listed
`[!a-e]`	any character not in given range

This is by no means a complete list. There are several other wildcards recognized by various shells. You are encouraged to look them up and see how they work.

2.4 Shell Variables

As mentioned earlier, the shell can be used as a scripting language, and like any programming language, the shell allows you to define **shell variables**. You can define any variables want to within your shell, with some exceptions. There are certain shell variables that are used to control the behavior of the operating system. These are examples of **environment variables** and should only be modified if you know what you are doing.

The difference between **shell variables** and **environment variables** is that shell variables are local to a particular instance of the shell program, be it your login shell or a shell script, while environment variables are inherited by any program you start, including another shell. That is, when you start a new process, that new process inherits a copy of all the environment variables for itself.

To set the shell variable `var` to the value `value` (this is for **bash**, the syntax is slightly different in **csh** based shells) you type:

```
var=value
```

or, in the case of an environment variable:

```
export var=value
```

Note that in either case, you do not need to declare the variable before you set a value to it.

These variables can then be expanded anywhere in the command line. To reference the variable, prefix the variable name with a **$** (dollar sign). For example, to print the value of the variable `var`, enter:

```
echo $var
```

2.5 Redirection

Input and output to and from the a program is accomplished through a mechanism called **character streams**. Under normal circumstances every Linux program has three streams opened when it starts: one for input; one for output; and one for error messages. The stream for input is called **standard input** (**stdin**) and is normally connected to the keyboard. The streams for output and error are called **standard output** (**stdout**) and **standard error** (**stderr**) respectively, and are normally connected to your terminal display. Though each stream has a default connection, they can be redirected to/from other sources, such as a file or pipe (more on the pipe later).

2.5.1 Standard Input

The **standard input** (**stdin**) is the stream used for input to the program (those that accept input). Usually, this stream is connected to the keyboard and receives user input interactively. For example, using the `mail` program, you can send an e-mail written from the keyboard:

```
aoaks@tsunami:~$ mail -s Hello aoaks@ocf
Hi,
Haven't heard from you in a while.
Aaron
[ctrl-d]
```

Here, text entered on the keyboard by the user is in bold. The [ctrl-d] is used to indicate that the user pressed control+d, which will send the EOF (end of file) signal. **EOF** tells the program listening on standard input that you have finished typing and are ready for your input to be processed.

This input method may be fine if you are only typing up a few lines in an e-mail, but what if you want to do something with an extensive input file? If

you have input that is hundreds of lines long, typing in the input data would be impossible. How, then, do we send these hundreds of lines of input to the program? The answer is to use redirection. Instead of typing in the data when you run the command, you put the data in a file and redirect standard input to read from that file instead of the keyboard. For example, if the e-mail in the previous example had been written and edited in a file name `long_email.txt`, you would type:

```
aoaks@tsunami:~$ mail -s Hello aoaks@ocf < long_email.txt
```

The < (left chevron) tells the shell that standard input should come from the file `long_email.txt` instead of the keyboard. You can sort of see that intuitively. The < makes it look like the `long_email.txt` file is being directed into the mail command, which it is.

2.5.2 Standard Output

The **standard output** (**stdout**) is the stream used for output from a program (from those that output information). Usually this stream is connected to the terminal display. For example, using the `who` command, you can find out which users are logged into the system:

```
aoaks@tsunami:~$ who
```

```
jameson  pts/8        Jan 17 01:06 (c-71-204-186-58:S.0)
aoaks    pts/23       Feb  6 02:58 (conquest.ocf.berkeley.edu)
sle      pts/7        Jan 27 12:24 (lightning.ocf.berkeley.edu)
```

Since standard output has not been redirected, the output data (i.e. info on the users) is sent to the terminal display. If, say, you wanted to write this data to a file for use later, you would simply redirect the output stream. For example:

```
aoaks@tsunami:~$ who > users.txt
aoaks@tsunami:~$ cat users.txt
```

```
jameson  pts/8        Jan 17 01:06 (c-71-204-186-58:S.0)
aoaks    pts/23       Feb  6 02:58 (conquest.ocf.berkeley.edu)
sle      pts/7        Jan 27 12:24 (lightning.ocf.berkeley.edu)
```

The > (right chevron) tells the shell that standard output should be sent to the file `users.txt` instead of the terminal display. Again, the > can be seen as an arrow directing the output from the `who` command to the `users.txt` file. As you can see, the first command did not print any data to the screen. The `cat` command simply displays the contents of the file and is used here to show that the file now contains the output of the `who` command.

2.5.3 Standard Error

The **standard error** (**stderr**) is the stream used for error output from a program (from those that output errors). Usually this stream is connected to the terminal display. For example, if you try to `cat` a file that doesn't exist, `cat` will output an error to **stderr**:

```
aoaks@tsunami:~$ cat missing.txt
cat:  missing.txt:  No such file or directory
```

Since this is printed to **stderr** and not **stdout**, if you redirect standard output, nothing will happen (well, the `error.log` file will be created, but since this outputs no data to **stdout** the file will be blank):

```
aoaks@tsunami:~$ cat missing.txt > error.log
cat:  missing.txt:  No such file or directory
```

To solve this, you will need to use **file descriptor** for **stderr**:

```
aoaks@tsunami:~$ cat missing.txt 2> error.log
aoaks@tsunami:~$ cat error.log
cat:  missing.txt:  No such file or directory
```

Here we see the correct results. The errors from the `cat` command have been redirected into the `error.log` file. Note that the redirection for **stdin** and **stdout** could have been accomplished using **file descriptors** (0< instead of < and 1> instead of >), but these are usually left off for simplicity.

2.5.4 Overwrite vs. Append

When you use the > redirection operator, the output file you choose will be **overwritten**. That is, when the shell is preparing the file to receive data from the command, if the file doesn't exist, a new file will created, and if the file does exist, its contents will be deleted.

If you want to **append** data to a file, you need to use the >> (two right chevrons) redirection operator. When you use >>, if the file already exists, rather than deleting the original contents, it will append the data to the end of the original contents.

2.5.5 Combined Redirection

The redirections used for **stdin**, **stdout**, and **stderr** can be used in any combination and in any order. For example, the following is completely valid:

```
bc < expressions.txt > results.txt 2>> error.log
```

This would direct a bunch of math expressions from `expressions.txt` into `bc` (a basic calculator program), output the results to `results.txt` and output

any errors to `error.log`.

Be Careful Do not use the same file for both input and output. As you will learn later, the character streams used for redirection are prepared before the command is actually run. In the case of output streams, if your output stream is set for overwrite, the shell will clear the file before opening the stream. At this point, the data in the file has been erased, so when the command goes to read the file, it will be empty. Unfortunately, at this point the data in the file can't be recovered, so always be careful not to do input and output using the same file.

2.5.6 /dev/null

Sometimes you just want one of the output streams to disappear. For example, if you are using the `find`, you will get errors every time find gets to a directory it doesn't have access too. This can quickly overwhelm the meaningful output, so often we want to get rid of **stderr**. This can be accomplished by redirecting **stderr** to **/dev/null**. This is a special file on the file system that is like a black hole for data. Any data that is sent there simply disappears. It doesn't accumulate in **/dev/null**, it simply vanishes. For example:

```
find / -name myfile.txt 2> /dev/null
```

this command will redirect all errors to **/dev/null** where they will vanish, so the only output that will be sent to the display is meaningful search results.

2.5.7 Combining Commands

Say, for example, you wanted to find out the number of users currently logged in to your system. The simplest way of doing this would be to use the `who` command to get a list of users on the system, then use the `wc` command, which can be used to could the number of lines in the input. If we only use the redirection abilities we have learned so far, we would do something like:

```
aoaks@tsunami:~$ who > users.txt
aoaks@tsunami:~$ wc -l < users.txt
15
```

While this did accomplish the task, there were some drawbacks. You had to type in two commands and you had to create a temporary file for use only with this command that is now wasting hard drive space and will have to be deleted. This may not seem so bad for this example, but the output from commands can be quite large and often you will be combining multiple commands, requiring a temporary file for each combination.

The solution to this problem is to create a **pipeline** using the | (**pipe**). The | redirects the standard output of the first command to the standard input of the second command, creating a **pipeline** between them. Using the previous example, the command is shorted to:

```
aoaks@tsunami:~$ who | wc -l
15
```

Here, `who` is said to have been **piped** to `wc`. You will find that this ability to combine simple commands to make much more complicated commands to be extremely useful in you work as a system administrator.

2.6 Quoting & Command Substitution

2.6.1 Back Quotes

Back quotes are used to evaluate commands and insert the output of those commands back where the back quotes were located. This is called **command substitution**. These creates a way to evaluate a sub-expression and place its output back in the original expression. For example:

```
aoaks@tsunami:~$ echo The date is ‘date‘
The date is Tue Feb 6 04:50:08 PST 2007
```

Notice that the `date` command in back quotes is evaluated and the results are placed back in the original expression.

2.6.2 Single Quotes

You already learned how to remove the special meaning of certain characters by escaping them with the \ (backslash). However, this can be very inconvenient in cases when you have to escape more than a couple of characters. To completely remove the meaning of all special characters, place them between `'`s (single quotes). For example:

```
aoaks@tsunami:~$ echo '*?[8-9]'
*?[8-9]
```

Notice that none of the wildcards were evaluated.

2.6.3 Double Quotes

A slightly less restrictive quoting method is using `"` (double quotes). Quoting with double quotes will remove the special meaning of all wildcards, but will allow the interpretation of the `$` (variable evaluation), and the ‘ (command substitution). For example:

```
aoaks@tsunami:~$ echo “* There are ‘who | wc -l‘ users currently logged into $HOSTNAME.”
* There are 16 users currently logged into tsunami.
```

Notice that the * wildcard was not evaluated, but the `$HOSTNAME` variable and command substitution were evaluated.

2.7 Treatment of the Command Line

Its important to note the "order of operations" the shell uses to take the command that you enter and transform it into the command it sends to the operating system.

Parsing: The shell first breaks up the command line into words, using spaces and tabs as delimiters, unless quoted. All consecutive occurrences of a space or a tab are replaced here with a single space.

Variable Evaluation: All words preceded by a $ are evaluated as variables unless quoted or escaped.

Command Substitution: Any command surrounded by back quotes is executed by the shell, and its output inserted in he place it was found.

Redirection: The shell then looks for the character >, <, >> to open the files that they pointed to.

Wildcard Interpretation: The shell scans the command line for wildcards and replaces them with a list of filenames that match the pattern.

Path Evaluation: Finally, the shell looks for the PATH variable to determine the sequence of directories it has to search in order to hunt for the command.