Lecture 2: The Unix Shell I

The Unix Shell

The shell is a program that provides a text-based interface between the user and operating system. It interprets commands that are typed by the user and performs operations that manipulate files in the filesystem and processes. While there are several different kinds of shells, we will cover commands that apply to the bash shell, or Bourne-Again shell, which is the shell supplied by default when you are given a Linux account.

This tutorial is an abbreviated version of the Shells and Utilities section of The Linux Tutorial, by James Mohr, which can be found at http://www.linux-tutorial.info/cgi-bin/display.pl?15&0&0&0&3, Sections The Shell, Directory Paths, Shell Variables, Permissions, Regular Expressions and Metacharacters, Quotes, Pipes and Redirections.

Ways to access a shell

The beauty of the shell is that users can obtain access to an operating system either at the machine or remotely. Both means of using the shell are identical to the user, the only difference being that the remote shell might be slower if the internet connection is slow. For our class we will actually never have direct access to the computer upon which we are using the shell. This is because the Linux machine(s) we will be using is in the Netlab, and you will not have direct access to the machine(s). Instead, you will be telnetting remotely either from the Sunlab or from any windows machine.

Access to the Linux machine from the Sunlab or any UNIX machine on campus

To access a shell on the Linux machine from a computer in the sunlab, open up a terminal (which, incidentally, is a shell on the sun machine) and at the prompt type

telnet 172.16.5.10

You will be presented with a prompt that reads

Connected to 172.16.5.10
Escape character is '^]'.
Red Hat Linux release 7.3 (Valhalla)
Kernel 2.4.18-3 on an i686
login:
Your login name will be the first letter of your first name followed by your surname. For example, my login name would be ofringer. After you type your login name, your password will be your username followed by @uwc.ac.za. Upon login, you will be presented with a prompt, which should look something like:

```
Trying 172.16.5.10
Connected to 172.16.5.10
Escape character is '^]'.
Red Hat Linux release 7.3 (Valhalla)
Kernel 2.4.18-3 on an i686
login: ofringer
Password:
Last login: Sat Mar 29 06:47:01 on :0
[ofringer@localhost ofringer]$ 
```

This is the shell prompt, and the shell is up and running and ready to interpret your commands. The first thing you should do is change your password by typing the `passwd` command.

### Access to the Linux machine from a windows machine

The beauty of the shell is that you can also access the machine from any windows machine from the DOS prompt. On any windows machine on campus, in the Start→Programs→Utilities pulldown menu, select the Microsoft DOS prompt. A window will appear with the C: DOS prompt. At this prompt, you can follow the same instructions as on a sun machine to access the Linux server. That is, at the DOS prompt, you type `telnet 172.16.45.201`, and this will connect you to the Linux server. Everything should be the same, except key bindings may differ so you may have to use different keystrokes to type certain special characters, such as `~` or backspace.

### A note about security

It should be noted that Telnet is not the most secure means of accessing a machine. In fact, one should never telnet to a machine unless that is the only means to access it. Telnet is insecure because the password you type at the prompt is transmitted unencrypted to the server. When available, you should always use `ssh` (Secure Shell) instead of `telnet`. Currently, `ssh` is not installed on the sun machines so we are forced to use `telnet`. If `ssh` becomes available to us, `telnet` access to the Linux machine will be forbidden and `ssh` access will be enforced.

### A note about tampering with the Class Linux machine

All access to the class Linux machine will be closely logged and monitored. Please be aware that any improper use of the machine can cause unnecessary downtime on the server or
can increase the load on the machine and make it unpleasant for others who are trying to use it. Your account on the machine is strictly for use only with assignments associated with COS315, and cannot be used for personal means. Any improper downloading, viewing, access, or modification of non-user or restricted files or directories will result in immediate removal of access privileges.

Shell basics

When you log into the shell you will be given a prompt that looks like

\[\text{ofringer@localhost ofringer}\]$ \]

From this point onward we will be using \(\text{prompt}\)$ to indicate the prompt. Your home directory is /home/ofringer, which, in shorthand, is also given by ~. You can see what your home directory is by using the echo command, which prints its argument to the screen, as in

\[\text{prompt}\]$ echo ~
/home/ofringer

At any point in time you can also see what your current working directory is with the pwd command, which stands for print working directory and which will give you the same result

\[\text{prompt}\]$ pwd
/home/ofringer

All UNIX commands are executed in the form

\[\text{command [OPTIONS...]} \ [\text{ARGUMENTS...}]\]

and most can take several arguments and options on the command line. For example, to create a directory in your current working directory, you would use the mkdir command. Let’s say you want to make a directory to store your assignments in it, as in

\[\text{prompt}\]$ mkdir assignments

To view the contents of your current directory, you use the ls command

\[\text{prompt}\]$ ls
assignments

Now let’s say you would like to create another directory that has your tutorials in it and you would also like to create another directory within that directory at the same time called tutorial1. This is done with the -p option to the mkdir command, which indicates that you would like to create the parent directory as well

\[\text{prompt}\]$ mkdir -p tutorials/tutorial1

so that now the contents of your current working directory is given by
$ ls
assignments tutorials

You can also see the contents of a particular directory by using that directory as an argument to $ls$

$ ls tutorials
tutorial1

Another convenient method of creating multiple directories is just to add multiple arguments to the $mkdir$ command

$ mkdir assignments/assignment1 assignments/assignment2
so that the contents of the assignments directory is given by

$ ls assignments
assignment1 assignment2

You can also supply multiple arguments to the $ls$ command as in

$ ls assignments tutorials
assignments:
assignment1 assignment2
tutorials:
tutorial1

You can supply each UNIX command with additional options as well. For example, to view the hidden and visible files with $ls$, you would use the $-a$ option, which stands for all,

$ ls -a
.. assignments tutorials

The . directory corresponds to the current directory, and the .. directory corresponds to the parent directory. You can use these when moving around the filesystem with the $cd$ command, for example, if you want to move to the assignments directory, you would type

$ cd assignments
$ ls -a
.. assignment1 assignment2

To move to the parent directory, you would changed directory to .. with

$ cd ..
$ ls -a
.. assignments tutorials
You can also supply multiple options to the `ls` command. For example, if you would like to see all of the files as well as a long listing of them, which shows their permissions and more information on them, then you would use

```bash
[prompt]$ ls -a -l
```

```
total 4
drwxr-xr-x 4 ofringer user 1024 Feb 26 13:44 .
drwxr-xr-x 3 ofringer user 1024 Feb 26 13:44 ..
drwxr-xr-x 4 ofringer user 1024 Feb 26 13:44 assignments
drwxr-xr-x 3 ofringer user 1024 Feb 26 13:44 tutorials
```

We will go into what this all means in a few sections. You can also use both options together as `ls -al`.

### Obtaining help with commands

You can always obtain more detail about a command in UNIX with the `man` command, which stands for manual. For example, to obtain more information about `ls`, you can type `man ls`. If you are not sure about which command you are looking for, you can type `man -k list`, which will give you a list of commands that have the word `list` in their short description. And, as always, you can use the reliable www.google.com to find the answer to just about anything.

### Search paths and the environment

When you type a command, such as `date`, you will get something like

```bash
[prompt]$ date
Wed Feb 26 13:59:32 EAT 2003
```

The `date` command is, like every command on the system, an actual file located somewhere that is being executed by the shell when you type it. UNIX knows where this command is because it searches your search path which contains directories that are used to search for commands. Your search path is defined by the environmental variable `PATH`, which is predefined when you log in. The variables that are set when you log in determine your environment, such as your login name `LOGNAME`, your home directory `HOME`, and the `HOSTNAME`. You can see a list of all of your environmental variables by typing `env`. To find out the value of a particular variable, such as `PATH`, you can type

```bash
[prompt]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/ofringer/bin
```

This may change according to the system you’re on, but what is important is that it contains a list of directories that the shell searches for when you type a command. To see exactly where the `date` command is located, you can use the `whereis` command,
[prompt]$ whereis date

date: /bin/date /usr/share/man/man1/date.1.gz

which tells you that date is located in the /bin directory and that there is an entry for the manual page in /usr/share/man/man1. You can execute the date command either with its full path as /bin/date or, since its path is already located in your search path, then you can simply type date. Unlike DOS, UNIX does not automatically look in the current working directory to executed commands. That is, if you have a command called list and its full path is /home/ofringer/list, you cannot assume that if your working directory is /home/ofringer that UNIX will find your command. Rather, you have to specify the path with ./list, which tells the shell that the command you’re trying to run is located in the current directory.

Sometimes there are different commands on the same filesystem that have the same name but perform different functions. UNIX prioritizes which commands it will executed by executing the first instance of the command found in the search path. So, given the path above, if we have a command called date in /home/ofringer/bin, then when we type date we can be sure that /bin/date will executed because it’s the first instance of date in our path. We can check to see which version will execute with the which command

[prompt]$ which date
/bin/date

Note that the difference between which and whereis is that which only searches your path while whereis looks for the executable in a standard list, which is given by

/{bin,sbin,etc}
/usr/{lib,bin,old,new,local,games,include,etc,src,man,sbin,X386,TeX,g++-include}
/usr/local/{X386,TeX,X11,include,lib,man,etc,bin,games,emacs}

Some commands are built-in to the shell, so they will not appear when you look for them with the which or whereis commands, such as the cd command.

Permissions and file information

When you type ls -al, you get a long listing of all of the files and directories in your current directory, such as (yours may be a bit more lengthy)

[prompt]$ ls -al

total 18
  drwx------  5 ofringer  cos315  1024 Feb 26 14:48  .
  drwxr-xr-x  7 root        root  1024 Feb 26 14:02  ..
  drwxrwxr-x  4 ofringer  cos315  1024 Feb 26 14:48  assignments
-rw-r--r--  1 ofringer  cos315   124 Feb 26 14:30  .bashrc
  drwxrwxr-x  3 ofringer  cos315  1024 Feb 26 14:48  tutorials
The first column contains a 10-character string that contains as its first character the type of file and as the next 9 the permission bits on the file. The following table describes the other columns:

<table>
<thead>
<tr>
<th>type</th>
<th>permissions</th>
<th>hard links</th>
<th>owner</th>
<th>group</th>
<th>size in bytes</th>
<th>modification time</th>
<th>filename</th>
</tr>
</thead>
</table>

The first character in the first column determines the type of file as follows:

- regular file
- character device
- block device
- directory
- pipe
- symbolic link

The device files can be found in the `/dev` directory, for example,

```
[prompt]$ ls -l /dev/tty1 /dev/hda1
brw-rw---- 1 root disk 3, 1 Aug 31 02:31 /dev/hda1
crw------- 1 root root 4, 1 Mar 30 2003 /dev/tty1
```

which shows that `/dev/hda1` is a block special file that represents a hard disk, while `/dev/tty1` is a character special file that represents a terminal.

The next 9 characters correspond to the permissions on the file. The first three correspond to the user or owner of the file, the next three correspond to the group, and the last correspond to others, and `r` corresponds to read, `w` corresponds to write, while `x` corresponds to executable. For example, since the permissions on the `.bashrc` file are `-rw-r--r--`, this implies that the user has read and write access to the file, members of the group `cos315` only have read access, and others also only have read access. You can change the permissions on a file with the `chmod` command, which can be used in one of many ways. The first way you set the desired permissions explicitly, such as

```
[prompt]$ chmod u=rwx,g=rx,o= .bashrc
[prompt]$ ls -l .bashrc
-rwx------ 1 ofringer cos315 124 Feb 26 14:30 .bashrc
```

so we have changed the permissions of `.bashrc` so that others cannot read it but that members of the group can read or execute it, while the user can read, write, and execute it (even though the file itself may not be an executable file anyway). Note that if we leave off a permission, it is not set, such as

```
[prompt]$ chmod g= .bashrc
[prompt]$ ls -l .bashrc
-rwx------ 1 ofringer cos315 124 Feb 26 14:30 .bashrc
```

We can also set the permissions additively, in that we can add or subtract certain permissions from the set, such as
$ chmod u-wx .bashrc
$ ls -l .bashrc
-r-------- 1 ofringer cos315 124 Feb 26 14:30 .bashrc

in which we have stripped write and execute permissions from the file. To add them back on, as well as to the group and others, we can use

$ chmod u+wx,g+rw,o+r .bashrc
$ ls -l .bashrc
-rwxrw-r-- 1 ofringer cos315 124 Feb 26 14:30 .bashrc

The other way we can set the permissions is by explicitly turning on and off the specific bits that make up the permissions for the file on whichever medium it is stored. That is, either setting its value to 0 (off) or 1 (for on). Consider the first three bits which correspond to the permission bits for the user (the next three correspond to those for the group, and so on...). Using the binary number system, there are only 8 possible numbers we can represent with three bits, namely 000,001,010,011,100,101,110,111, which correspond to 0,1,2,3,4,5,6,7 in the decimal number system. If the first location corresponds to the r bit, and the second and third correspond to the w and x bits, respectively, then each of the decimal numbers corresponds to 1 of 8 possible permissions, as shown in the following table:

<table>
<thead>
<tr>
<th>Permissions</th>
<th>Binary representation</th>
<th>Decimal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>--x</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>-w-</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>-wx</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>r--</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>r-x</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>rw-</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>rwx</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

You don’t have to know binary arithmetic to figure all of this out. In fact, because of the nature of binary arithmetic, all you need to remember is the following table:

<table>
<thead>
<tr>
<th>value</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>r</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>

From this table we can see that a value of 4 in place of the user permission bit implies read permission, while a value of 2 implies write permission, and 1 implies execute permission. We can use the sums of these numbers to also imply more than one permission, for example, to get read and write permission, this would be $4+2=6$. Now you can see why you only need an octal number to specify the permissions of a file, since there are only three bits, which only allows for a total storage of 8 numbers, or 8 types of permissions! You can then use the values in this table to specify the permissions for the user, group and other in a file, such as

$ chmod 755 .bashrc
$ ls -l .bashrc
-rwxr-xr-x 1 ofringer cos315 124 Feb 26 14:30 .bashrc
or

[prompt]$ chmod 600 .bashrc
[prompt]$ ls -l .bashrc
-rw------- 1 ofringer cos315 124 Feb 26 14:30 .bashrc

Permissions on a directory

Since everything in UNIX is regarded as a file, a directory also has permissions on it just as a file does. You can check the permissions on a directory without having to check the permissions on every file in that directory with the -d flag in the ls command

[prompt]$ ls -dl assignments
drwxrwxr-x 4 ofringer cos315 1024 Feb 26 14:48 assignments

This shows that everyone has full permissions except for others, who do not have write access to the directory. You can change the permissions on directories just as you can for files. The difference is that write access to a directory implies that you can create or delete files in that directory, while read access implies that you can read files in that directory, while execute access to a directory implies that you can enter that directory and list its contents. As an example, change the permissions on a directory as follows

[prompt]$ chmod go-wrx,u-x assignments
[prompt]$ ls -dl assignments
drw------- 4 ofringer cos315 1024 Feb 26 14:48 assignments

Since you do not have execute privileges to the directory, if you try and list its contents, you will be denied

[prompt]$ ls -l assignments
ls: assignments/assignment1: Permission denied
ls: assignments/assignment2: Permission denied
total 0

Now set the permissions on the directory back to what they were with

[prompt]$ chmod 755 assignments

UMASK and file creation permissions

When a file or directory is created, the program that creates it automatically assigns a set of permissions to that file. For example, when you create a new directory with mkdir, the mkdir program creates a directory with permissions of 777. We may not want to allow the creation of directories that are writeable by the group and others, but we don’t care if those directories are executable or readable by others. So we want to make sure that any directories that are created don’t have write permission. The way to do this is with the
umask command in UNIX. This command can be interpreted as the inverse of chmod. That is, rather than specifying the permissions you want to apply, you specify the permissions you do not want to allow. As an example, let’s say you do not want to allow write access by the user or group, but you don’t mind if the program that creates a file or directory makes it readable or executable by you or anyone else, and you don’t mind what kind of permissions it sets for you, the user. This implies that you don’t care about the first bit, but you want to restrict the second two to no more than 5. Therefore, you would set the file permission mask to 022 with

[prompt]$ umask 022

As an example, let’s see what the default permissions are when you create a directory with mkdir. Since we don’t want to mask any of the bits, we first do so with

[prompt]$ umask 000

Then we make a directory and check its permissions with

[prompt]$ mkdir testdir
[prompt]$ ls -dl testdir
drwxrwxrwx  2 ofringer cos315  1024 Feb 26 16:14 testdir

which shows us that the default permissions on the creation of directories by mkdir is 777 (the -d flag implies a directory listing, so that it doesn’t show us the contents of the directory, but rather, shows us the permissions on the directory itself). Let’s say that we do not like these permissions and that we want to prevent the group and others from writing and executing files in the directory (or searching for files in the directory). This implies that we want to disallow the w and x bits for both group and others, implying that the mask should be $0(w+x)(w+x)=0(2+1)(2+1)=033$. First, remove the directory you just made and try again

[prompt]$ rmdir testdir
[prompt]$ umask 033
[prompt]$ mkdir testdir
[prompt]$ ls -dl testdir
drwxr--r--  2 ofringer cos315  1024 Feb 26 16:20 testdir/

The easiest way to figure out what the permissions will be on a file or directory created by a program will be to subtract the mask from the permissions created from the file, and this will be the permissions on the file. In this example, since mkdir creates a directory with permissions 777, if the mask is set to 033, then the resulting permissions on the file will be $(7-0)(7-3)(7-3)=744$. As another example, if the mask is set to 022, then the permission on the directory will be 755, or rwxr-xr-x, which is the most common mask to set. You will probably not need to change the mask to anything other than 022 because usually you can just change the permissions manually once a file or directory is created. You can now remove the test directory with

[prompt]$ rmdir testdir
Metacharacters

When you are looking for a file you do not need to type in its entire name to list its contents, but rather, you can use metacharacters that represent certain parts of file or directory names that the shell will fill in for you. For example, in our home directories, by now we have the assignments and tutorials directories. Let’s make some more directories and files that we can use as examples. Create the following directories

[prompt]$ mkdir -p test/dirs{1,2,3,4,5,a,b,c,aa,bb,cc}/
/testdir{A,B,C,AA,BB,CC}

This will create a total of 78 directories with the test directory as the parent (the \ implies don’t evaluate the return character so that you can add on more on the next line). You can use the * metacharacter to list all directories in test that end with an a with

[prompt]$ ls -d test/*a
test/dira  test/dirsaa

and you can also list all of those that end with an A in testdir with

[prompt]$ ls -d test/*a/*A
/testdirsa/testdirA  test/dirsaa/testdirAA
/testdirsta/testdirA  test/dirsA/testdirAA

You can also use the ? metacharacter to specify only one character to replace, rather than a string of characters that replace the * metacharacter, as in

[prompt]$ ls -d test/dirs?
test/dirs1  test/dirs3  test/dirs5  test/dirsb
/testdirs2  test/dirs4  test/dirsA  test/dirsC

which only shows the directories with one character after the dirs. Alternatively, you can only show those directories that have two characters after the dirs, as in

[prompt]$ ls -d test/dirs??
test/dirsaa  test/dirsbb  test/dirscc

and you can use the ? and * metacharacters together as well

[prompt]$ ls -d test/*rs?/testdir?
test/dirs1/testdirA  test/dirs3/testdirA  test/dirs5/testdirA  test/dirsb/testdirA
test/dirs1/testdirB  test/dirs3/testdirB  test/dirs5/testdirB  test/dirsb/testdirB
test/dirs1/testdirC  test/dirs3/testdirC  test/dirs5/testdirC  test/dirsb/testdirC
test/dirs2/testdirA  test/dirs4/testdirA  test/dirsA/testdirA  test/dirsC/testdirA
test/dirs2/testdirB  test/dirs4/testdirB  test/dirsA/testdirB  test/dirsC/testdirB
test/dirs2/testdirC  test/dirs4/testdirC  test/dirsA/testdirC  test/dirsC/testdirC

You can also use the bracket ([ ]) metacharacters to imply a range upon which you would like to list. For example, to list the directories dirs1,dirs2,...,dirs5, you can use
[prompt]$ ls -d test/dirs[1-5]
test/dirs1 test/dirs2 test/dirs3 test/dirs4 test/dirs5

which also works for characters, as in

[prompt]$ ls -d test/dirs[a-c]
test/dirsa test/dirsb test/dirsc

Another metacharacter is $, which indicates a variable. We already saw the use of $PATH to specify that PATH is an environmental variable. But we can also specify our own variables with, for example,

[prompt]$ string=dirs

And we can check the value of the variable with

[prompt]$ echo $string
dirs

We can then use this variable in the list commands that we performed before:

[prompt]$ ls -d test/$string?
test/dirs1 test/dirs3 test/dirs5 test/dirsb
test/dirs2 test/dirs4 test/dirsa test/dirsc

Quotes

There are three different kind of quotes that you might use in the UNIX shell. They are the back-quotes, ‘(usually on the same key as the ~), single-quotes ’, and double-quotes ‘‘. The back quote is used to enclose a command and substitutes whatever is in the back-quotes as the output of the command itself. For example, a commonly used command is wc, which counts the number of words, lines, and characters in a string that it is fed, and outputs the result. For example, if the ls command is piped into the wc command, the result is

[prompt]$ ls -al | wc
    15  128  912

which indicates that the command output 15 lines, 128 words, and 912 characters. You can see for yourself what it outputs with ps ax without the |wc (we'll deal with the | shortly). If you wanted to store the output of this command into a variable, you would type

[prompt]$ var=`ls -al | wc`

and then you could use that variable later as

[prompt]$ echo $var
15 128 912

The single-quote is different because it puts out exactly what was written between the quotes. For example, if the above command is typed with single quotes, we get
which means that nothing is evaluated within the quotes, including any metacharacter, such as the $. For example, if we typed a command with a $ in single quotes, we would get

```
[prompt]$ echo '$var'
$var
```

This is useful when we want to echo something in which we need the $ to be printed. For example,

```
[prompt]$ echo 'This costs $2.99 per kilo.'
This costs $2.99 per kilo.
```

However, a problem now arises because let’s say we want to include a variable in something we’re printing. This can be accomplished with double-quotes. Double quotes are identical to single quotes except that the metacharacter $ is interpreted within them. For example,

```
[prompt]$ var=2.99 ; echo 'This costs $var per kilo.'
This costs 2.99 per kilo.
```

But now we have run into another problem. How do we include the $ and also evaluate the variable $var? This is done by using double- and single-quotes together, as in

```
[prompt]$ var=2.99 ; echo 'This costs "$var" per kilo.'
This costs $2.99 per kilo.
```

More simply, we can also force printing of the $ with

```
[prompt]$ var=2.99 ; echo 'This costs \$var per kilo.'
This costs $2.99 per kilo.
```

**Pipes and redirection**

When you log in to a UNIX machine, the program that is running and reading your commands is called the shell. The shell has three files associated with it that it opens up immediately when you log in, or when you start another shell. These are the standard output, `stdout`, the standard input, `stdin`, and the standard error, `stderr`. The drivers that manage your keyboard input convert your keystrokes into characters that are written to `stdin` and subsequently read by the shell and passed to the screen. When you hit a return key the shell processes the characters read in from `stdin` and outputs the results to the standard output `stdout` and the errors from the file to the standard error `stderr`. Both of these results are displayed on the screen. Take for example the `ls` command. When you type the letters `l` and `s`, they are read in to the shell via `stdin`, which waits for a return keystroke. When it reads one, it executes the command `ls` and writes the resulting directory listing to `stdout`, which appears on the screen. Since no errors occurred, nothing is written to `stderr`. Now try typing
Several hundred files will fly by the screen, and you won’t be able to read them as they go by. This is because all of stdout is written to the terminal at once. We have two options that we can use to view the results slowly. The first is to redirect the standard output to a file and view the contents of the file, as in

```
$ ls /dev > results.txt
```

The > implies that we redirect the standard output from the `ls` command to the file `results.txt`. We can view the contents of `results.txt` with the `more` command with

```
$ more results.txt
```

This allows us to scroll through the results more slowly (and then we can delete the file with `rm -f results.txt`, where the `-f` flag forces the removal without asking us for verification first). Rather than create a new file and delete it later, we can take advantage of the pipe in UNIX, which is given by the `|` symbol. This is done by piping the standard output from the `ls` command into the standard input of the `more` command using

```
$ ls /dev | more
```

Now let’s say you typed the command

```
$ ls /dev /def > results.txt
```

Since `/def` does not exist, the error message was not written to the file `results.txt` because the `>` command only redirects standard output, not standard error. The redirection command `>` is actually shorthand for `1>`, which says redirect the standard output, which is represented by file descriptor 1. To redirect the standard error, then, since it is represented by file descriptor 2, then we would use `2>`. In the above example, then, to redirect the standard output to `results.txt` and the standard error to `errors.txt`, we would use

```
$ ls /dev /def 1> results.txt 2> errors.txt
```

We can also redirect the standard error into the standard output so that the error and output show up in the same file, as in

```
$ ls /dev /def 1> results.txt 2>&1
```

Which can be done more easily simply with

```
$ ls /dev /def &> results.txt
```

where `&>` indicates that we would like to redirect both the standard output as well as the standard error into the file `results.txt`. We can also employ both redirection and pipes, such as
$ ps ax | grep pts > out.txt

where the `ps ax` command lists the processes, and the output of that is piped into the `grep` command, which prints out instances of the string `pts` that it finds. This output is then redirected into the file `out.txt`. As a final example, we can employ multiple pipes, as in

$ ps ax | grep pts | wc

in which the output from `grep pts` is piped into `wc`, which counts the number of words, lines, and characters in the output of the `grep` command.