

INTRODUCTION TO UNIX (II)

Notes by Sean Hogan, last revised 10/5/11

(I) INTRO

If you haven't, go look through the Intro to unix I handout.

This time we'll go over other nice things, like compression and archiving, piping, file transfer, running a program and signals.

There are files for this set of exercises. run the following commands, you'll learn what they mean in a bit.

```
$ scp your-name@naos.cs.uchicago.edu:/var/tmp/unix-ii.tar .
$ tar xvzf unix-ii.tar
```

You'll now have a unix-ii folder willed with goodies.

(II) Compression/archiving

This is very much the same as making a zip file or something. Archiving lets you put multiple files into a single file that you can send around from place to place. Compression uses algorithms to help save space in that archived file, which is why compressed/archived directories usually take up less space than the normal directory and its contents.

Say we have a directory, "stuff" and it has two files, "1" and "2"

Then:

```
seanhogan@rigel:~/unix$ tar cvzf stuff.tar stuff
stuff/
stuff/1
stuff/12
```

The "c" flag means we want to create an archive. "v" means for the tar command to be verbose, "z" tells it to use a certain type of compression, and "f" says that the next argument will be the name of the output archive.

So in this case, we compressed and archived the "stuff" folder and its contents, and put the output into stuff.tar. We can move this stuff.tar file around normally.

Now it'd be nice to get our stuff back out, so we do the following:

```
seanhogan@rigel: ~/unix: tar xvzf stuff.tar
```

And whatever we archived will now be in whatever folder we're in.

Syntax:

```
tar cvzf OUTPUT_NAME INPUT_FILE_1 INPUT_FILE_2 ...
tar xvzf TAR_FILE
```

(III) File transfer

|
We covered this a little bit last time, but it's good to go over again.

sftp (secure file transfer), scp (secure copy) are commands used to move files around.

scp is nice when you know where your file and destinations are explicitly. For example, I can back up my homework from my dorm room just by executing:

```
$ scp hw.tex seanhogan@naos.cs.uchicago.edu:hw/hw.tex
```

which has the effect of copying my "hw.tex" into the folder "hw" (relative to my home directory) on a different computer.

Likewise you can copy stuff from a remote computer to some other computer -

```
$ scp name@cpuone:file.tex name@cputwo:files/file.tex
```

What if we don't remember the structure of the filesystem of the computer we want to get stuff from/put stuff on? sftp can help with that.

```
$ sftp seanhogan@naos.cs.uchicago.edu
```

After giving my password, the above will give me a prompt:

```
>
```

At which only some commands work. Most important is "ls -l" and "cd". Then I execute

```
$ put file.txt
```

And from my current computer's working directory, the file "file.txt" will be transferred into whatever folder I'm in on the remote computer.

Likewise:

```
$ get file.txt
```

Will do the inverse operation (a copy of file.txt will appear in the folder I'm working in.)

Note that if these files exist, put/get will overwrite them! Be careful.

***Helpful with sftp will be the "pwd", print working directory command, for getting your bearings.

EXERCISES:

(1) In your unix-ii folder is a file, "your-letter.txt". Put whatever you want in it using your favorite text editor (e.g., "gedit your-letter.txt"), and make sure to rename the file to something else (e.g., "seanhogan-letter.txt"). Make a copy of it with a different name. Use scp, and then sftp, to move it to the folder "/var/tmp/mailbox" on naos.cs.uchicago.edu .

(2) I have a file called "message.txt" located in /var/tmp/mailbox on naos.cs.uchicago.edu . Use scp or sftp to copy it to your working directory on the computer you're logged into.

(IV) More general remote access (ssh) (secure shell)

ssh lets you log into other unix-like systems and get stuff done. Convenient uses include but are not limited to:

- Running programs on computers far away
- Doing your CS labs while you're at home
- etc...

Basically at some point this will likely become useful.

Say I'm at home. Then running this:

```
$ ssh seanhogan@naos.cs.uchicago.edu
```

will prompt me for a password. This password is the password of your CS account.

Then you'll be at the prompt for the computer you ssh'd into.

```
seanhogan@naos:~$
```

This is why the identifier (name@cpu) is useful, often you will be working in the terminal on your computer, and somewhere else. Everything else works as normal.

Now, say you're in the maclab but you want to look at your friends' silly pictures or something. But they're on a different computer, bellatrix. Then you can just run (assuming you're logged into one of the linux machines):

```
$ ssh bellatrix
```

And you'll be logged in there, without having to type the full name or your password.

Note also that:

```
$ ssh -X name@computer
```

Will allow you to remotely look at the GUIs for applications on these linux machines. So if you ssh -X into say, naos, and run "chromium-browser", you'll be able to interact with it as if you were running it on your current computer (except it'll be a heck of a lot slower, since all the visual data must be sent over whatever connection you're using.)

You can leave an ssh session with CTRL-D.

(V) Piping

Note that for the most part, every command we've been using has sent text to the terminal window - e.g., the listing of a directory, the contents of a file, etc. When something like this happens, we say a command has sent its output to "stdout" (standard output). By default, stdout is the terminal window. However, you can redirect such text! We call this piping, and we'll learn the basics which should be fine for most of whatever you need.

There's a file called not-sorted.txt in you unix-ii folder that we'll use for this.

Now run the following:

```
$ cat not-sorted.txt
$ wc not-sorted.txt
$ cat not-sorted.txt | wc
```

Notice that "wc not-sorted.txt" and "cat not-sorted.txt | wc" have the same output. The difference is we piped the stdout of "cat not-sorted.txt" (the contents of "not-sorted.txt") to the stdin (standard input) of "wc". The normal way to feed something into stdin of wc would be to just run "wc not-sorted.txt", so "not-sorted.txt"'s contents would go into wc's stdin., which then prints to stdout. Of course you could very well pipe the output of wc to some other program.

This, of course, is an entirely useless example. Here's a slightly more useful one:

```
$ ls | wc
```

Since ls lists every file/directory in the working directory as separate lines (maybe), wc will count that many words, or the number of files+directories. The output of wc is LINES WORDS CHARACTERS.

```
$ sort not-sorted.txt | head
```

Will sort the text in "not-sorted.txt" and give me the first 10 lines. Note the difference from plain running

```
$ sort not-sorted.txt
```

Don't forget about the man command! All of these tools have many different flags - Explore. some examples with sort:

```
$ sort -k 2 <file>
```

Sorts by the 2nd column of data. In the example case, the number column. You'll notice the numbers are

sorted in alphabetical order, though, but we can fix this:

```
$ sort -k 2 -n <file>
```

and backwards...

```
$ sort -k 2 -n -r <file> | head
```

Gives us the top 10 words with the highest numbers!

Also in the future, any programs you run that print out text can have their output piped! This will allow you to store output in files, and append. They use different symbols, though:

Create empty "file" and stick program's output into it

```
$ ./program > file
```

Append program's output to file.

```
# ./program >> file
```

Try this with commands, e.g.:

```
(1) $ ls > lsoutput.txt  
(2) $ ls -l >> lsoutput.txt  
-----
```

MORE STUFF

grep is a powerful tool. We'll just do something basic with it though.

```
$ grep d not-sorted
```

returns all lines in the file that have a "d" in them. This can be useful and chained to something like "wc"

```
---  
EVEN MORE
```

Enter this:

```
$ echo $PATH
```

This shows the value of your "PATH" variable. It's something that your terminal or shell program looks for whenever you enter a command. E.g., the "cat" command is actually a program in /bin/:

```
$ which cat
```

You can modify these variables of course.

```
$ export PATH=$PATH:/my/dir
```

will append "/my/dir" to your list of directories for the shell to look through when entering a command.

You can set your own variables as well.

```
$ export F00=/my/path
```

```
$ cd $F00
```

will take you to /my/path .

Now you need to set these on every terminal session. There are some script files that your shell may run on startup. This is specific to what you use. On the linux machines you can write these export commands in your ~/.bashrc file. Then when you start the terminal, just enter

```
$ bash
```

and the script should run.

Sometimes when you run a program it will run indefinitely or do something wrong.

Often, typing "Ctrl-C" will send a "Interrupt signal", which (usually) kills, or stops the program. "Ctrl-D" sometimes kills everything, and exits the terminal window.

These will be more useful later.

This is all I really have to say. The rest you are now able to learn through necessity/classes/experimentation/research! I would advise that you try and learn some text editor that isn't gedit (one reason is that you will be able to edit text remotely without a slow GUI). The main contenders are either "vim" or "emacs".