Delving more deeply into UNIX

Buffalo Chapter 3

Overview

1) A Little Review

2) Unix Exercise and Tutorials

3) New UNIX material:

- Standard Out and Standard Error
- Creating and navigating through directories
- Using wildcards
- The Pipe and behold, `grep`!
- Redirecting streams in pipes
- Managing processes
- Checking process exit status

A little review...

- What are some ways you can make an analysis pipeline **reproducible**?
- What are some ways you can make an analysis pipeline **robust**?
- What are some other key lessons we discussed from Buffalo Chapter 1 or, better yet, that you learned in your own reading?
- What are the kernel, the shell, and commands?

Unix Exercise and Tutorials:

- As a reminder, in the `Week_01` folder of the GitHub repository, I have placed a UNIX exercise and instructions for how to log in and complete this on hpc-class
- I would continue working through this this weekend, and, since Monday is Labor Day and there are no classes, type questions in Slack.
- If you would like additional, basic tutorials on UNIX some online resources include:

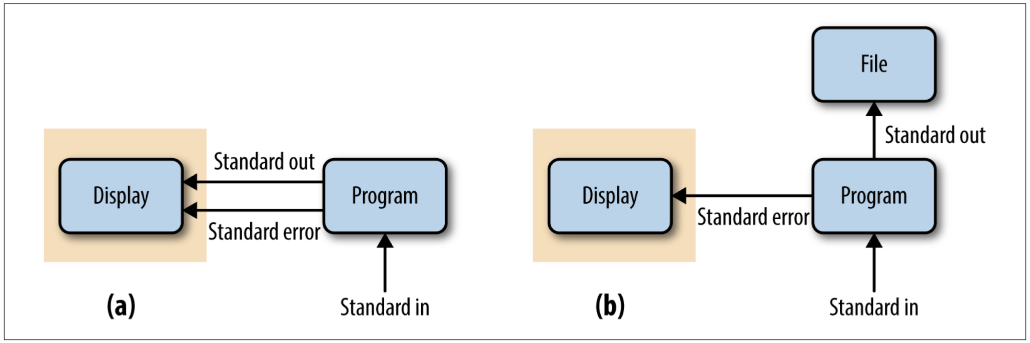
https://sites.google.com/site/eeob563/computer-labs/Lab-1

ftp://ftp.imicrobe.us/biosys-analytics/lectures/unix_and_perl_v3.0.pdf

- And before going further: Any ideas how and when UNIX can be useful?

And now for something new...

Standard Out and Standard Error



- What is the difference between standard output and standard error?
- Within the `Week_01` folder you have several examples files; once inside the folder try this:

```
$ cat file1 file2 file3
```

- In the output, what is standard out and what is standard error?

Standard Out and Standard Error

- How can standard output and standard error be redirected?

```
$ cat file1 file2 file3 > data 2> error
```

- How can standard out be appended to an existing file?

```
$ cat file4 file5 >> data
```

Creating and navigating through directories:

- Let's set up a project as in Buffalo Chapter 3:

```
$ mkdir zmays-snps
$ cd zmays-snps
$ mkdir data
$ mkdir data/seqs scripts analysis
```

- Try this in your course folder and use the `ls` and `cd` commands to make sure you understand what is happening with the last line of code
- From within the `seqs` folder, how might you navigate to the `zmays-snps` folder in one line of code?

The `rm` command and why nerds use underscores in their folder/file names:

- Let's create a folder in `zmays-snps` with a space in its name: `raw sequences`

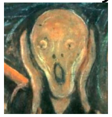
```
mkdir raw\ sequences
```

- Now in `zmays-snps` let's create two more folders:

```
$ mkdir raw sequences
```

- Over the next few weeks we pile data into `raw` and `sequences` and then one night when we're under-caffeinated we decide to remove the `raw sequences` folder:

```
$ rm -rf raw sequences
```

- What just happened?
  

- How *should* we have removed the `raw sequences` folder?

Using shell expansion to make your life easier:

- Let's go ahead and delete the nice `zmays-snps` directory we've created:

```
$ rm -rf zmays-snps
```

- Note that this folder and *all its subdirectories* are erased...thank you `-rf` option

- Now let's try creating the entire project directory in a single line of code:

```
$ mkdir -p zmays-snps/{data/seqs,scripts,analysis}
```

- Explain exactly what's going on here...

- Note the use of the `-p` option for the `mkdir` command which allows for creation of intermediate directories as required

- Let's use shell expansion to create some files in our reconstructed project directory:

```
$ cd zmays-snps/data
$ touch seqs/zmays{A,B,C}_R{1,2}.fastq
```

Wildcards can make your life easier too!

- Navigate into your `seqs` folder and use the `ls` command in combination with the `*` and `?` wildcards to match subsets of the files we just created
- How do `*` and `?` match differently?
- Let's create new `R1` and `R2` folders and use a wildcard range to move only the "A" and "B" files into these new folders:

```
$ mv zmays[AB]_R1* R1
$ mv zmays[AB]_R2* R2
```

- Convince yourself only the appropriate files have been moved and then move the "C" files as well
- Always be careful with wildcards, particularly when using the `rm` command. For example:

How are these different?

```
$ rm -rf tmp-data/aligned-reads*

$ rm -rf tmp-data/aligned-reads *
```

The Pipe

- In this example, how are standard out and standard error streams being funneled?

```
$ cat file1 file2 | grep "AGGATA" | wc
```

- Why pipe rather than create intermediate files?
- Behold the mighty `grep` command!!
- Let's see what this command can do using an example from Buffalo Chapter 3...
- First, we need to clone supplementary material for the book into hpc-class so `cd` to the top of your directory and type:

```
$ git clone https://github.com/vsbuffalo/bds-files
```

Suppose we're working with a program that throws an error telling us that our fasta input file has non-nucleotide characters. Let's use grep in a pipe to inspect our input file:

```
$ grep -v "^>" tb1.fasta | grep --color -i "[^ATGC]"
```

Controlling streams within pipes

- Pipes can string together multiple programs and increase the efficiency of our analysis
- However, imagine your pipe includes 20 programs and multiple errors are thrown to your display during the analysis
- Which program had the issue printed to standard error?
- Let's talk through an example of how to manage this:

```
$ program1 input.txt 2> program1.stderr | \
    program2 2> program2.stderr > results.txt
```

- But what if we want to send our standard output and standard error to the same place?

```
$ program1 input.txt 2>&1 | grep "error"
```

But what if I really love intermediate files?

- Sometimes you or your collaborator may need intermediate files in a pipeline for other analyses or for debugging
- Can you retain the efficiency of the pipe while also creating intermediate files?
- You betcha:

```
$ program1 input.txt | tee intermediate-file.txt | program2 > results.txt
```

Managing processes: sending programs to the background:

- Often times our UNIX programs and pipelines will run for an extended amount of time
- It is not terribly convenient to sit and stare at our terminal for weeks at a time
- The running job also ties up our terminal if we're running it in the foreground (however, you can open up multiple tabs or terminals)
- One solution to this is running your analysis in the background using the ampersand:

```
$ program1 input.txt > results.txt &
```

- This process will be run in the background, freeing up your terminal, and a process ID will be provided:

```
[1] 25744
```

Managing processes: checking status and bringing to the foreground:

- Say the next day we come to work and want to check quickly whether our analysis is still running:

```
$ jobs
[1]+     Running     program1 input.txt > results.txt
```

- And what if, now that we're back at work, we want to stare at the process while it runs all day?
- This is where your process ID number will come in handy:

```
$ fg %25744
```

- Now you can watch your process run to your heart's content
- Question: *What happens when you run a program in the background and close your terminal application?*

Managing processes: sending active programs to the background:

- Say you start a program, realize it's going to take forever to run and then want to send it to the background…

```
$ program1 input.txt > results.txt
$ # enter control-z here...NOT CONTROL-C!!
[1]+ Stopped     program1 input.txt > results.txt
$ bg
[1]+ program1 input.txt > results.txt
```

- To irrevocably kill a job type "control-c"; your job must be in the foreground for this to work

Checking the exit status of a completed program

- Say we come into work and find our program has completed with no errors printed to our display
- To double-check that all has gone swimmingly, we can check the exit status by inspecting our shell variable:

```
$ grep -v "^>" tb1.fasta | grep --color -i "[^ATGC]"
CCCCAAAGACGGACCAATCCAGCAGCTTCTACTGCTAYCCATGCTCCCCTCCCTTCGCCGCCGCCGACGC
$ echo $?
0
```

- You can utilize the exit status in your pipelines by implementing the shell operators `&&` and `||`
- A few examples:

```
$ program1 input.txt > intermediate-results.txt && \
    program2 intermediate-results.txt > results.txt
```

```
$ program1 input.txt > intermediate-results.txt || \
    echo "warning: an error occurred"
```

Exit status operators, `true` and `false`

- There are two Unix commands that are very useful for understanding exit status, the shell variable and operators: `true` and `false`
- `true` always sets the shell variable to 0 (success)
- `false` always sets the shell variable to 1 (failure)
- Try the following commands and see if what they return makes sense to you:

```
$ true && echo "first command was a success"
$ true || echo "first command was not a success"
$ false || echo "first command was not a success"
$ false && echo "first command was a success"
```

Command substitution:

- Sometimes rather than piping, we may actually want to nest commands within other commands
- This process is called "command substitution" and here are a few examples...
- `cd` into the `chapter-03-remedial-unix` folder of your Buffalo online materials and in your terminal type:

```
$ echo "There are $(grep -cv '^>' tb1.fasta) lines of sequence in my FASTA file."
```

- What is the result, and what's going on here?
- Now `cd` into your `in_class` folder and try:

```
$ mkdir results-$(date +%F)
$ ls
```

- You can name folders and files with today's date without even knowing what that is!