CS 35L Week 1 Worksheet

Emacs, Linux, & I/O Redirection

Emacs

If you are struggling to remember all of the commands for Emacs, check out this cheat sheet. Helpful for the assignment and exams!

1. Why Emacs?
    a. Which text editors/IDEs have you used before Emacs?
        i. Some examples: VS Code, XCode, etc.
    b. What are the differences between using them and Emacs?
        i. Main difference is that most are Graphical User Interfaces (GUIs) rather than part of the Command Line Interface (CLI).
    c. What's the most difficult part of using Emacs so far?
        i. Many students probably struggle with using the key bindings/shortcuts and getting used to a text editor for the first time.
2. Can you ssh into lnxsrv15.seas.ucla.edu?
    a. You need to be connected to the UCLA VPN if you are off campus.
    b. If you have trouble with your account, make sure to reach out to the SEASNet admin.
    c. Hopefully the answer to the above is yes. Please attend office hours or ask on Piazza if you need some help with this.
3. Moving around. Start by running the command **emacs test.txt** to create a file.
    a. How do we exit emacs?
        i. C-x C-c (note that C is Emacese for the CTRL key)
    b. Try creating a dribble file.
        i. Run M-x open-dribble-file. Remember to do this on parts of the homework (won't need to do this for later assignments). For Mac users the meta key is ESC by default while for Windows users its the Windows key.
            1. If you are a Mac user and want to change your meta key, you can follow this tutorial.
        ii. Try typing some characters and then save your dribble as **discussion.drib.** Keep track of the number of keys you type.
        iii. Try running **wc -c discussion.drib**. What does this do? Does the number of characters match the number of keys you type?
            1. This is a helpful sanity check on the homework to see that the number of keystrokes you typed is recorded in the dribble file.

You can also use **xxd discussion.drib** or **less discussion.drib** if you want a more visual output.

iv. Try running **wc -c <discussion.drib**. Is there any difference in the output compared to the previous part (iii)?

1. Yes, the output no longer includes the filename. In the previous part to this question, wc accepts a filename as an argument and will output the filename. When wc reads from standard input, there is no filename associated with the input (standard input is not a named file), so there is no additional output besides the character count.

v. Try running **cat discussion.drib | wc -c**. Is there any difference in the output compared to the previous part (iv)?

1. No, there is no difference. The output is the same because wc once again reads from standard input, and we've piped the standard output from the file using cat to standard input. (The standard output from the file using cat is just the contents of the file.)

4. Try working with the people around you to figure out the following using this cheat sheet to figure out how to do the following. This is similar to how you will do Assignment 1.
   a. First, type M-x open-dribble-file. How do you cancel this command?
      i. C-g
   b. How do you move forward/backward a character?
      i. Forward (to the right): C-f
      ii. Backward (to the left): C-b
   c. How do you move up/down a line?
      i. Forward (down): C-n
      ii. Backward (up): C-p
   d. How do you move to the start of a line?
      i. C-a
   e. How do you search and move to the first instance of a word?
      i. C-s, hit ENTER, type in what you are looking for, and then hit ENTER
   f. What other key bindings would be helpful to know?
      i. Any that use the Meta key is a common one. Referencing the cheatsheet is helpful.

The Linux Shell

The following is meant to introduce you to some of the basic Linux commands and get some practice with using the shell.

1. For the Assignments, you need to complete them on the SEASNet servers lnxsrv11, lnxsrv13, or lnxsrv15, with **/usr/local/cs/bin** prepended to your PATH.
    a. Before prepending to PATH, run the command `**which cat**`. What is the output?
        i. /usr/bin/cat
    b. Now, run `**export PATH=/usr/local/cs/bin:$PATH**`, then run `**which cat**` again. What is the output?
        i. /usr/local/cs/bin/cat
    c. What is the effect of `**export PATH=/usr/local/cs/bin:$PATH**`?
        i. Prepend /usr/local/cs/bin to your PATH so that when your terminal looks for a command, it checks /usr/local/cs/bin first.
    d. How do you "permanently" prepend /usr/local/cs/bin to your path?
        i. You can edit your .profile file in your home directory to run the above command every time you log on to the SEASNet server.
2. Run the command `**man man**`. What does it say? You can navigate the man page by using the arrow keys to move up/down a line. 'd' goes Down one page, 'u' goes Up one page, 'q' Quits.
    a. Shows the man page for man.

3. What do the following commands do? Try using commands such as 'j', 'k', 'G', and 'g'. If the man pages are too long for you, check out the TL;DR pages.
    a. which: show's command's path
    b. wget: downloads an HTML page
    c. cp: copies files (with -r you can also copy directories)
    d. mv: moves files/directories, helpful for renaming things as well
    e. ls: lists files (by default in the present working directory)
    f. chmod: modifies the permissions on a file
    g. find: looks for a file in your current directory

4. Execute each of the following commands, and then describe what each one did (Hint: **ls** is your friend!):
    a. cd ~/Desktop: changes your present working directory to ~/Desktop
    b. mkdir foobar: creates a new directory named "foobar"
    c. cd foobar: changes your present working directory to ./foobar
    d. touch silent.txt: creates a new file named "silent.txt"
    e. echo "woof" > dog: creates a file named "dog" containing the contents, "woof"

f. echo "oink" > pig.animal: creates a file named “pig.animal” containing the contents “oink”
g. cat silent.txt: prints the contents of the “silent.txt”
h. cat dog: prints the contents of the file named “dog”
i. cat pig.animal: prints the contents of the file named “pig.animal”
j. Did all of the cat commands succeed? What does this tell you about the meaning/purpose of file extensions (.txt, .pdf, .docx, etc.)?
    i. All of the commands succeeded. File extensions are actually meaningless when it comes to determining the file type. If you are interested, the way to determine the file type is to use the **magic bytes** of a file.
k. cd ...: changes the present working directory to
l. rmdir foobar: This will error. rmdir cannot remove non-empty directories.
m. **(Be careful you type this command exactly!)** rm -rf foobar: this is how you recursively remove directories and its contents.

5. The following question is from the **Winter 2024 Midterm**. It has a nice emphasis on the POSIX file system and understanding relative paths and names.

Suppose there are no symbolic links anywhere in the system, and that all files named in this problem exist.

a (3 minutes). Give a pathname that is equivalent to `./../.../..../.../../.` but is as short as possible.

Remember that

- `.` = current directory
- `..` = parent directory
- Everything else with dots is just the NAME of a directory

The above simplifies to `../.../..../`
Note the leading `../` is still necessary since this is a relative path.

b (3 minutes) Likewise, but for `/./../.../..../.../.././` instead.

Same as the above, except this is for an absolute path. The above simplifies to `/.../..../`

There is also a part three to this question but it is a bit confusing so it's removed for now.

I/O Redirection & Shell Scripting

A big emphasis of the shell scripting part of Assignment 1 is using pipes and I/O redirection. If you are struggling to remember all of the syntax for shell scripting, check out this cheat sheet.

1. Remember to always start your bash scripts with the line "**#!/bin/bash**".
    a. If you try to run a shell script but you get the error "Permission denied", make sure that you have given that file execution permissions by running:

```
chmod +x <file-name>
```

Note: Don't just give execution permissions to any file! Before you execute any script, always make sure that you know who wrote it and what it does!

2. Write a shell script named "hello.sh" that, when run, outputs "Hello, world!"

```
#!/bin/bash

echo "Hello World"
```

3. For each command, explain what the standard input, standard output, and standard error is when the command is run. Assume you are running each of these inside a Bash script. Try not to run each command (midterm & final practice 😀).

Command	**stdout (1)**	**stderr (2)**
1. echo "mistakes were made" >&2 Redirecting standard output to standard error.		"mistakes were made"
2. echo "i love emacs" | wc -c How a pipe works.	19	
3. (echo "tmp" >&2 >&3 >&4) 3>&1 4>&1 Only the last redirection is applied. This also illustrates how you can create temporary file descriptors using I/O redirection.	tmp	

If you are trying to run this command, make sure to run this on the SEASNet server in Bash and not a different shell like zsh which many MacOS computers use as their command line program. Zsh has a few different behaviors than Bash.		
4. ./doesnotexist 2>&1 | wc -c **Hint**: The standard error for running ./doesnotexist is the following. -sh: ./doesnotexist: No such file or directory Using I/O redirection to pass standard error into a pipe. Does not command is a program that does not exist.	47	
5. (echo "a" && ./doesnotexist) 2>&1 1>&2 vs (echo "a" && ./doesnotexist) 1>&2 2>&1 **Hint**: Does order matter with I/O redirection? Link **Common Misconception**. In I/O redirection, a common misconception with this example is how the order of I/O redirection may appear to be "flipped" since iIn the first example, both contents are redirected to stdout while in the second, both are redirected to stderr. You can verify by redirecting standard output and standard error into /dev/null (blocking the output of one of them). An example of this is the following command.	"a" "-sh: ./doesnotexist: No such file or directory" --- --- --- ---	--- --- --- "a" "-sh: ./doesnotexist: No such file or directory"

(echo "a" && ./doesnotexist) 2>&1 1>&2) >/dev/null I/O redirection asks the process to refer to the new file descriptor as the old file descriptor (internally, it uses dup2). For example, take the following I/O redirection. ./doesnotexist 2>&1 What 2>&1 is doing is asking old file descriptor 2 (for standard error) to refer to the new file descriptor 1 (for standard output) for the next process. Let's look at the first example closely. (echo "a" && ./doesnotexist) 2>&1 1>&2 This first tells the process to use the file descriptor 2 to refer to standard output (file descriptor 1) of the new process printing the error message to standard output. It then tells the process to use the file descriptor 1 to refer to the new file descriptor 2 (which now refers to standard output) which is why both are printed to output.		
6. echo "hello" | cat - The goal of this is to demonstrate how - allows you to take in from stdin when normally you need to take input from a file. An example of this is the **comm** command used in Assignment 1 (i.e. **comm -23 - sorted.words**). In the first example, the piped stdout is ignored but in the second it is printed to stdout.	hello	

4. The following is a question from the **Winter 2024 CS 35L midterm**. It has a nice emphasis on I/O redirection, pipes, and using the shell. **Hint**: To replace all lowercase

letters with uppercase letters, the command is **tr a-z A-Z** and pass what you are looking to convert into standard input.

(14 minutes). Consider the following shell transcript:

```
$ echo 'cat: not a dog' >a

$ echo 'cat: short for Caterpillar heavy equipment' >b

$ cat a b c

cat: not a dog

cat: short for Caterpillar heavy equipment

cat: c: No such file or directory
```

Suppose you find it hard to distinguish normal output from diagnostics. You would rather have the error message capitalized, so that in the above example the last line of output is:

```
CAT: C: NO SUCH FILE OR DIRECTORY
```

Write a shell script named 'capcat' that does this. 'capcat' should behave like 'cat', except that everything sent to standard error should be capitalized. 'capcat' should work without creating any temporary files. If you cannot implement 'capcat' as stated, implement as much as you can, and briefly explain which subset you're implementing and why there is trouble implementing the rest of 'capcat'.

```
#!/bin/bash

(cat "$@" 2>&1 >&3 | tr a-z A-Z >&2) 3>&1
```

Explanation:

- cat "$@" calls cat on the first argument of the bash script. Remember to quote your arguments to prevent argument splitting (for example, if a file name has a space in it, Bash may sometimes treat this as two arguments so it's a good practice to quote your arguments while shell scripting).
- Left hand side of the pipe
    - 2>&1 standard error is passed to stdout for the pipe. This is meant to pass to tr so that everything can be capitalized.
    - >&3: a temporary file descriptor is used to contain the contents of stdout since this is occupied by the redirected stderr. This is meant to contain the valid output of the program
    - Note the order is correct so that stderr is not accidentally passed into the temporary file descriptor.
- Right hand side of the pipe

    - tr a-z A-Z: converts all lowercase letters from stdin to uppercase. This is meant to capitalize the output of stderr.
    - >&2 takes the stdout of this command and returns it back to stderr since this is now all capitalized
- Outside the parentheses
    - 3>&2: the temporary file descriptor is now then passed back into stdout where this is the normal output of the program.