Lecture Agenda and Notes Draft:
Unix on Local Machine 2:
Data Wrangling, Editing and Pipes

Recap: File Descriptors: stdin, stdout, stderr [5 min]

1. File descriptors. A file descriptor is a number that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.
   In Linux, libc opens for each launched application 3 unique file descriptors by default, numbering them as 0,1,2. man stdio / man stdout:

Name	File descriptor	Description	Abbreviation
Standard input	**0**	The default data stream for input, for example in a command pipeline. In the terminal, this defaults to keyboard input from the user.	**stdin**
Standard output	**1**	The default data stream for output, for example when a command prints text. In the terminal, this defaults to the user's screen.	**stdout**
Standard error	**2**	The default data stream for output that relates to an error occurring. In the terminal, this defaults to the user's screen.	**stderr**

Grep and searching for file contents

journalctl-2021sep.log --- files available in the folder marked with yellow

1. *grep* utility is for searching for content in text files
2. Data Wrangling: transformation of data from one format into another format, data modification. For example: you have the text to get statistics from. When using pipe between two commands in a shell, we do data wrangling.
3. Play with Regular expressions at https://regexr.com
4. Usage examples
    a. *grep -i 'abcd' testfile* --- returns strings with abcd in any case
    b. *grep -w 'test' testfile* --- returns only those lines where test is a separate word
    c. *grep -r '456' /home/* --- recursively traverses the directory and outputs lines that fall under the pattern
    d. *grep -v 'practical' testfile* --- inverse, outputs strings without an occurrence
    e. *grep -r -l "Network" /var/log/** --- outputs only file names containing the pattern
    f. *grep -A1/-B1/-C1 '123' testfile* -- outputs a string after / before / before and after the occurrence
5. Search with regular expressions:
    a. *grep -E 'ab.d' textfile*
6. Given journalctl-2021sep.log get data containing "ssh" using *grep*, then select "Disconnected from" from the result of the grep. Save the result to the file "my_ssh.log"
7. Using *less*, view the contents of "my_ssh.log"

Data wrangling and sed [20 min]

1. Introduce sed: stream editor, a kind of programming language over streams. Using sed, perform the operation of removing the prefix in rows with the same content in the middle using:

    i. `sed 's/.*Disconnected from //'` tell what does it do

b. Give an example of how sed works, using command line, on a toy line, to demonstrate how regular expressions work:

    i. `sed ‘s/[ab]//’` with *aba* string, to remove first occurence,

    ii. и `sed ‘s/[ab]//g’`, to remove all occurences

    iii. `sed -E` use it for “common” regular expressions (where there’s no need to write \(or \))

    iv. `sed -E ‘s/(ab)*//g’`, to remove all *ab* from the string

c. Turn back to our file, run `head -10` on the file

d. Give an example, when sed might remove important information:
`echo ‘Disconnected from invalid user Disconnected from 84.211 ‘ | sed 's/.*Disconnected from //'`.

    If there’s a user with the name “Disconnected from” then there’s an issue.

e. Tell about the following commands

    i. `sort`

    ii. `uniq -c`

    iii. `wc -l`

    iv. `sort -nk1,1`

    v. `head -10, tail -10`

f. Tell how to find proper flags for the commands using `man`. Steps:

    i. From a course like this, you will learn what the command is used for

    ii. Then you encounter a new problem, and try to find the appropriate flags in the man of this command

g. Next, we work with the output of the command:
`cat my_ssh.log | sed -E 's/^.*Disconnected from ([0-9.]*) port [0-9]+ \[preauth\]?/\1/' | sort | uniq -c | sort -nk1,1 | tail -n20 | awk '{print $2}' | paste -sd"\n"`

    Tell about

    i. `awk ‘{print $2}’` --- processes the stream by columns

    ii. `paste -sd,` --- connects strings with a delimiter

    iii. `awk ‘$1 == 1 && $2 ~ /^c.*e/ {print $0}’` --- takes rows with one in the first column and the second element satisfying the regular expression, outputs the entire row

    iv. to count rows one can do `wc -l`, or `awk ‘BEGIN {rows=0} $1 == 1 && $2 ~ /^c.*e/ {rows+=1} END {print rows}’`

v. *bc -l* to execute mathematical expression

h. Tell about *xargs*, how it passes a list as arguments

Command-line editing with Vim

1. Vim is a text editor with which you can modify code and texts directly in the console
2. Vim has the following feature: at each moment of working with the text, the editor is in one of three states
    a. Normal mode --- navigation and manipulation with text (remove lines, etc.)
    b. Insert mode --- type and delete text
    c. Command line mode -- for command typing, this commands may save a file or move cursor to some location
3. When you call the *vim textfile.tx*t, an existing or empty file will open. By default, you get into Normal Mode. You can switch to Insert Mode by pressing i on the keyboard in Normal Mode
4. Pressing Esc will switch from Insert Mode to Normal. Pressing again will still leave you inside Normal Mode.
5. To exit **Vim without saving type :q!**, i.e. in Command Line Mode send an exit without saving command. One can **close Vim using ZQ** (shift+z, shift+q) in Normal Mode
6. Open **greet.py**, write down a function that takes the student name and prints two lines “Hello Student. How are you doing?”. Call this function in a script **greet_me(“Tutor”)**
7. To exit with the saving type :wq or :x!
8. Enable line numbers with *:set number*
9. For moving to line 2 type **:2**, then **:3**. To travel to the end of the file use **:$**
10. Press **$ в Normal Mode** and you’ll get to the end also
11. Delete one line of code using **dd в Normal Mode**.
12. To Undo use **u в Normal Mode** (not Ctrl+Z)
13. Now copy part of the line and insert it as a line below^
    a. In **Normal Mode** press **ctrl+v** and highlight the text
    b. then press **y**, means **yank**
    c. Press **o** and free line appears below (side effect you’re in the **Insert Mode)**
    d. Press Esc, then **p** (paste), to paste
14. Comment out Python code?
    a. IN **Normal Mode** press **ctrl+v** choose vertically where it is needed to add “#”. Hashes will be paste before highlighted area
    b. then enter Insert-at-Left Mode with **Shift+i**, press “# “ and **Esc**. They’ll appear on first positions
15. To remove comments
    a. Highlight “#” with **ctrl+v** and press **d**
16. To find the text in the file

   a. Type **/hello** then **Enter**.

   b. To jump between occurrences use **n and N**

17. To tell you that you can configure vim for yourself using the vimrc file

Executable files and Shebang. Shell and Python Scripting

1. The Unix system determines what is executable depending on the Permission. Show what's inside /usr/bin *ls -lh /usr/bin/…* , their x bit means they're executables
2. There are two types of executable: binary precompiled and scripts, shebang is used in scripts. The shebang string defines how to execute executable: as a shell script or as a python script
3. Header options
    a. For python #! /usr/bin/python3 or #! /usr/bin/env python3
    b. For bash #!/usr/bin/bash or #!/usr/bin/env bash
4. Bash scripting: the following done in command line
    a. One can define variables using: *foo=bar.*
    b. Spaces are really critical: *foo* = *bar* won't work. *foo* = *bar* means to run a *foo()* function on two parameters *"=" and "bar"*
    c. To get value of a variable use dollar sign: *echo $foo*
    d. Quotes: double quotes and single quotes work similar if define a literal string: *echo "hello" the same as echo 'hello'*
    e. But *"hello $foo"* (has substitution) is not the same as *'hello $foo'* (no substitution)
    f. Bash allows to define functions: *mcd.sh* =>
        i.
        ```
        mcd() {
           mkdir -p "$1"
           cd "$1"
        }
        ```
        ii. *"$1"* is the first argument passed to a bash script
        iii. one can load this function using *source mcd.sh* then call *mcd test*
    g. *$?* stands for an error code of a previous command (you can use it in pipes), example: *grep something mcd.sh; echo $?*
    h. *$_* stands for the last argument of previous command, example: echo "hello"; echo $_
    i. One can store the output of a command to a variable using: *foo=$(pwd)*
    j. Go to example.sh and discuss what's inside
5. Python scripting [for processing complex text data]: sys.stdin, sys.stdout, sys.stderr, print, buffered IO
    a. goal: write a python script that can be used as part of pipeline e.g. cat <file> | grep YY | ./myscript.py | grep XX

Reading

- Tutorial on Find https://danielmiessler.com/study/find/
- Data Wrangling (mostly about sed) Lecture https://missing.csail.mit.edu/2020/data-wrangling/
- Game based on Vim https://vim-adventures.com/
- Learn Vim For the Last Time: A Tutorial and Primer https://danielmiessler.com/study/vim/
- Vim Cheat Sheet https://vim.rtorr.com/
- Sed, a stream editor https://www.gnu.org/software/sed/manual/sed.html
- Bash Scripting Cheat Sheet https://devhints.io/bash