Advanced Unix Command Line

You have been provided with a tar file that contains some folders/files etc. for doing the below activities. Download the file and untar it (tar -xvf filename ; replace filename with the name of whatever you downloaded). It should create a folder called unix. Navigate to that folder. Then, navigate to the `file-analysis` folder. Most commands below assume you are inside the `file-analysis` folder. Note that you can also perform these activities on cLabs in Lab0. Doing on clab is highly recommended.

wc

1. Ensure you are inside the file-analysis folder. wc prints word (-w), character (-m), line (-l), byte-count (-c) in a file. wc can accept zero (reads from standard input) or more inputs. Default output: lines, words, characters. The first command prints the lines, words and characters (in that order) of file bigfile. The second command prints only number of lines. Try the remaining commands also, they are straightforward.
    a. wc bigfile
    b. wc -l bigfile
    c. wc -w bigfile
    d. wc -m bigfile
    e. wc -c bigfile
2. We can use wc with multiple files (below we are using it across two files)
    a. wc /proc/cpuinfo /proc/meminfo

grep

1. Ensure you are inside the file-analysis folder. Grep: grep searches for PATTERNS in each FILE and prints each line that matches that pattern. Note that a line is not decided by full-stops in the para (like we humans do) but by a newline (\n). “-i” ignores the case. The first command will not show a sentence starting with “For”, whereas the second command will. The third command matches the exact pattern made of two words.
    a. grep “for” bigfile
    b. grep -i "for" bigfile
    c. grep -i "for a" bigfile
2. Regex Metacharacters (^ beginning of line; $ end of line; . match any single character)
    a. grep -i "^for" bigfile
    b. grep -i "cried.$" bigfile
    c. grep -i "h.s" bigfile
    d. grep -i "t..t" bigfile
3. More metacharacters: Suppose you want to use “or” operator i.e. “|”. The first command will fail, since it tries to match the same exact pattern. The second works, since by using \, you are escaping the | i.e you will interpret it as the “or” operator. However, this tends not to be clear when reading regular expressions. The best way around is the use of the

third command, use of -E (has to be capital) in which case you don't need to escape special characters. Use the man page of grep to know more.

   a. grep "cried|could" bigfile
   b. grep "cried\|could" bigfile
   c. grep -E "cried|could" bigfile

4. Quantifiers: * zero or more times; ? zero or one time; + one or more times; {n} exactly n times; {n,} at least n times; {,m} at most m times; {n,m} from n to m times
   a. grep -E "to*" bigfile
   b. grep -E "to?" bigfile
   c. grep -E "to+" bigfile
   d. grep -E "to{1}" bigfile
   e. grep -E "to{1,}" bigfile
5. Groups and Ranges: ( ) group patterns together; { } match a particular number of occurrences (saw this in 4.d and 4.e above); [ ] match any character from a range of characters. In the third command, the interpretation of "^" is not beginning of a line, rather since it appears within [], it means any letter except i. The fourth command, any letter in the alphabet range from A to Z (notice capitals, it will only match capital letters).
   a. grep -E "(fear)?less" bigfile
   b. grep -i "h[ia]s" bigfile
   c. grep 'h[^i]s' bigfile
   d. grep "[A-Z]" bigfile
6. Special characters. \s indicates white space. So, the first command will search for a three letter word that starts with h and ends in d; which has a space in front and at the end. \b matches any character that is not a letter or number without including itself in the match.
   a. grep -E "\sh.d\s" bigfile
   b. grep -E "\bh.d\b" bigfile
7. Grep is not restricted to just file search
   a. ls -l | grep file
8. More options of grep: -v negation, -r recursive search, -w whole word, -n show line number, -c count (count of lines, not number of occurrences). In the first command, every line that has the word "Wolf" is not printed! In the second command, only those lines that have "Wolf" are printed. In third command, we are using -r to search across all files in that folder (specified via *) i..e grep can work across files also, need not limit to one file.
   a. grep -v "Wolf" bigfile
   b. grep "Wolf" bigfile
   c. grep -r "Wolf" *
   d. grep -w "his" bigfile
   e. grep "his" bigfile
   f. grep -n "Wolf" bigfile
   g. grep -c "Wolf" bigfile

find

1. Locate files/directories with known patterns. Ensure you are in the parent of the student-files folder. The first command, -name tells to match the name of the given file (i.e. files ending in jpg) in the folder student-files. The second command is asking it to check only for files (-type f) ending in ".c" and be case insensitive in file names (-iname) in the current directory.
    a. find student-files/ -name "*.jpg"
    b. find . -type f -iname "*.c"
2. The first command lists all directories in the current directory (-type d)
    a. find . -type d
3. The first command finds files of size more than 60kbytes in current directory (use "ls -Rl ." to check the file sizes to verify if the command worked fine). The second command shows files with permission set to 644 in the current folder (permissions will be covered shortly). The last command, finds files starting with f in dir1 folder and then deletes the files within (be very careful when you execute this, since it will remove files)
    a. find . -type f -size +60k; ls -Rl
    b. find . -perm 644
    c. find dir1/ -name "f*" -delete

cut:

1. Ensure you are inside the file-analysis folder. cut: helps cut parts of a line by delimiter, byte position, and character ("-f" specifies field(s), "-b" specifies bytes(s), "-d" specifies a delimiter, --complement complements the selection and --output-delimiter specifies a different output delimiter string)
2. Below are some examples involving students.csv file. If you do cat students.csv, you will see various fields separated by comma (which is the delimiter). The first command specifies what delimiter to use (, in this case) and to output fields 1 and 3 (serial no and name). The second command is using -4 which means fields 1 to 4.
    a. cut students.csv -d ',' -f 1,3
    b. cut students.csv -d ',' -f -4
3. You can change the output delimiter as well. First command now separates the field by space when outputting.
    a. cut students.csv -d ',' -f 1,3 --output-delimiter=" "
4. When using complement, it outputs fields that are complement of 1,2,3 i..e 4,5,6,7,8
    a. cut students.csv -d ',' -f 1,2,3 --complement
5. Here are examples of -b. Note I am using echo and redirecting its output to cut. You don't necessarily have to use a file
    a. echo "abcdefg" | cut -b 1,3,5
    b. echo "abcdefg" | cut -b 1-4

paste:

1. Helps merge lines of files horizontally. -d can be used to specify a delimiter when pasting. And -s pastes one file at a time in serial.
    a. paste fruits1 fruits2
    b. paste -d '_' fruits1 fruits2
    c. paste -s fruits1 fruits2

sort

1. sorts the records in a file. The first command sorts in ascending order and the second command sorts descending (reverse)
    a. sort fruits1
    b. sort -r fruits1
2. When you have numbers, sort will still do character wise sort without taking the numerical value under condition i.e. (13 will come before 9). Use -n to sort numerically. In third command, we are combining two options, -n and -r as -nr.
    a. sort list1
    b. sort -n list1
    c. sort -nr list1
3. One can also sort by specifying a field separator via -t option and key via -k option. Below will sort the data in students.csv based on 8th field i.e. key (hostels). Note separator is needed to tell what the fields are.
    a. sort -t ',' -k8 students.csv

uniq

1. Finds repeating adjacent lines in a file and displays as a single line. First check the contents of fruits file using ‘cat fruits’. Then run the following commands. The first command displays each fruit name only once. Second command displays the number of times each line appears with the line.
    a. uniq fruits
    b. uniq -c fruits
2. You can control what lines you want to display using the following. First command displays only the repeating lines. Second command displays only the unique lines.
    a. uniq -d fruits
    b. uniq -u fruits
3. You can also compare how the lines are compared. -i ignores case while comparison. -s skips the given number of characters and -f skips the given number of fields. Note that here, fields are whitespace separated. The first command prepares a file. Try the following commands and see output.
    a. cat fruits1 fruits2 > fruits3 && paste fruits3 fruits > fruits4
    b. uniq -f 1 fruits4
    c. uniq -s 10 fruits4
    d. uniq -i fruits4

zip

1. zip archives files with compression. The first command recursively zips all contents of fun_dir as well as a file students.csv. The second command will encrypt the zip via -e option, it will ask for a password. The third command excludes Articles folder from being included in the zip. zip -r example.zip fun_dir/ students.csv
    a. zip -re example.zip fun_dir/ students.csv
    b. zip -r example3.zip fun_dir/ -x fun_dir/Articles
2. The -s option can split the zip file into multiple zip files. This is generally useful for very large files. The second command unzips this file. Note that you first combine all the parts together and then unzip
    a. zip -s 64k -r example4.zip fun_dir/
    b. zip -s 0 example4.z* –out combined.zip && unzip combined.zip
3. unzip unarchives. The -l option does not decompress but lists what is inside the zip including file sizes, modification dates etc. Useful to know before unzipping. The second command unzips the given zipped file into a specified directory, which is specified via the -d option. If you instead just typed “unzip example.zip”, it would unzip in the current directory itself (if similar folder/files are present, it will ask if it should overwrite; you can change this behaviour by -o (always overwrite) or -n (never overwrite)). The fifth command excludes the memes folder from being extracted.
    a. unzip -l example.zip
    b. unzip example.zip -d dir1/
    c. unzip example.zip -o
    d. unzip example.zip -n
    e. unzip example.zip -x fun_dir/Memes/*

tar

1. This is also used to archive files and directories, and also extract them. The first command creates a tar file while the second command also compresses it using gzip.
    a. tar -cvf example.tar.gz fun_dir/
    b. tar -cvzf example2.tar.gz fun_dir/
2. To untar the files, we use the -x option instead of -c.
    a. tar -xvf example.tar.gz
    b. tar -xvzf example2.tar.gz

Redirection >, >> and <

1. > redirects the output of a command into a file. >> appends instead of overwriting
    a. ls -l > ls-out
    b. ls >> ls-out; cat ls-out
    c. ls > ls-out; cat ls-out
2. When a program prints both to the stdout and also has some error messages printed to stderr, you can use fd> to redirect output accordingly. commands.sh is a program (bash script, which we will cover later, for now you treat it just as a black box program; you

have). The first command will direct the output to the out file, the error messages are printed on the screen. The second command directs the error messages to out file, while the output is print to stdout. The third command sends both to two different files. The fourth command sends both to same file. The fifth command sends the error messages to a special file called /dev/null that discards anything written to it (used to suppress error messages)

    a. ./commands.sh 1> out
    b. ./commands.sh 2> out
    c. ./commands.sh 2> error 1> out
    d. ./commands.sh > out 2>&1
    e. ./commands.sh 2> /dev/null

3. You can use < to read from a file instead of stdin. If you just type cat, whatever you type at a terminal it will echo back. You can get out by typing ctrl+c. But it you use cat < smallfile it will take the input from that file (note “cat smallfile” will also work as such)
    a. cat
    b. cat < smallfile
4. You can use both > and < also
    a. wc -l < smallfile > lines

Pipe |

1. Pipe takes output from one command and feeds it as input to another command. The first command feeds the output of ls to wc. The second command uses two pipes and essentially tells the number of (sub)directories in the current directory. The third command also uses two pipes. The cat combines two files and feeds it to sort, which then feeds it to a command uniq which removes duplicates (-i asks it to ignore case)
    a. ls /etc | wc -l
    b. ls -l | grep "^d" | wc -l
    c. cat fruits1 fruits2 | sort | uniq -i

Command Substitution

1. Output of a command replaces the command itself. Usage $(command) or `command`.
2. We will cover this via an example using date and touch commands. Date is a command which tells the current date. With option +%s, it tells the seconds passed since January 1st, 1970 at 00:00:00 UTC, which is referred to as the Unix epoch (standardized reference point for measuring time across different systems). touch is a command which can be used to create empty files.
    a. date +%s
    b. touch file.txt ; ls
3. Suppose you want to uniquely name files, you could use the date command to timestamp the file as follows. The output of the date command is now part of the touch

command. Both the below commands have the same result, though timestamp will be different since time of execution of commands is different.

a. touch file-`date +%s`.txt ; ls
b. touch file-$(date +%s).txt ; ls

ps and pkill

1. ps displays a list of processes currently executing. When you type ps in a shell without any options, it will show only the processes associated with that shell. In the second command, we are using options, aux, “a”: display the processes of all users; “u”: user-oriented formats; “x”: list processes without a controlling terminal
    a. ps
    b. ps aux
2. ps command also allows you to sort the output. For example, to sort the output based on the memory usage , you would use:
    a. ps aux --sort=-%mem
3. pkill can terminate a process. It is typically used if you find some process not responding. To pkill a process, we need its process name. Type below in sequence. The first set of commands, the first entry is starting a process that will just sleep for 60sec, & instructs it to run in background, so the terminal is freed up (you can type sleep 60 without & and see what happens). The second command terminates the sleep process. Then type ps again to see that it is indeed killed. In the second set of commands, start two sleep processes to sleep for 60 and 50 seconds respectively. Then, the third command kills only the sleep 60 process. Then run ps aux to check that sleep 50 is still running.
    a. sleep 60 & ;pkill sleep; ps
    b. sleep 60 &; sleep 50 &;pkill -f ‘sleep 60’; ps aux

Access Control

1. Permissions for a file or directory may be any or all of : r - read; w - write; x - execute. Access permissions are displayed using “ ls -l”. Observe a few things, commands.sh has rwx permissions i.e. why it could be executed via ./commands.sh. The file oddball has read permissions but no write permissions. Cat works but open it in nano and try to write, you will not be able to write!
    a. ls -l
    b. cat oddball
    c. nano oddball

2. Let us change some permissions now. First command uses the numeric mode to remove all permissions for all users. Neither cat nor nano to write will work now. The second command uses symbolic mode to change permission of a user to read. Now the cat command should work but cannot edit via nano still. The third command uses numeric mode to assign both read and write permissions to the file. Now both read and write should work.
    a. chmod 000 oddball; cat oddball; nano oddball

    b. chmod u=r oddball ; cat oddball; nano oddball
    c. chmod 640 oddball; cat oddball; nano oddball
3. Another example which removes read permissions from a group recursively from a given folder. Go through the files before and after and see the diff.
    a. ls -lR fun_dir > before ; chmod -R g-r fun_dir; ls -lR fun_dir > after

su and sudo

1. su is short for super user or switch user. It does, as its name suggests, switch users. If no user is specified, it switches to root. This requires password of the root you're switching to
    a. su <username>
2. The first command executes a single command as the other user. The second command specifies what shell to use after switching user (This requires the path of the shell, not only the name. su -s bash won't work but su -s $(which bash) will work). The third command preserves the current user's environment variables after switching (env prints environment variables).
    a. su <username> -c ls
    b. su <username> -s /bin/bash
    c. env; su <username> -p; env
3. sudo runs a single command as root user. It requires current user's password, unlike su. The second command runs ls as the specified user, not the root. The third command lists all the paths the current user can run as root. The fourth command clears the cached password, forcing the user to enter password again for the next sudo. The fifth command preserves the environment variables (like su -p)
    a. sudo ls
    b. sudo -u <username> ls
    c. sudo -l
    d. sudo -k
    e. sudo -E