**Lab: Linux CLI - Connecting, Logging In, Users, Groups**

Cleanup from prior runs (we will explain these command later):

cd /home/w205/user/labs/week\_03

rm -rf temp\_1

rm -f test.bash  
  
rm -f temp.tar  
  
rm -f temp.tar.gz  
  
rm -f temp.tgz

cd

clear

Website that will graphically explain shell commands to you:

[https://explainshell.com/Links to an external site.](https://explainshell.com/)

Man (manual) pages are old school way to get information about shell commands. Note that it has a simple screen based browser. Use the arrow keys, the enter key, the space bar, etc. to move around. When you are done, hit a q to quit:

man hostname

man uname

man cat

If you google for Linux man pages, you will see lots of websites that have online Linux man pages that you can query and read. This is usually much easier.

The Linux prompt shows you the: username, hostname (IP address in our case), and directory.

You can scroll back and forth through previous command history using the up and down arrows. You can edit previous command. You can hit enter anywhere in the edit and it will be sentd (don't have to be at the end of the line).

You can see the history with the history command:

history

When typing command in the Linux command line, the screen will scroll. If you want to start with fresh screen, you can use clear:

clear

If the screen gets messed up, say you accidentally print a binary file, you can reset it:

reset

Get the hostname (for AWS, note is it the internal, not external name!):

hostname

Get the kernel name, kernel release version, and the machine type:

uname --kernel-name --kernel-release --machine

Get the Linux "distro" (distribution) information:

cat /etc/os-release

Get the username of the user currently logged in:

whoami

Get the username and id, primary group name and id, other groups:

id

Get the list of all users:

cat /etc/passwd

Get the list of all groups:

cat /etc/group

The .ssh directory holds the authorized\_keys file. This file stores public keys. Users are authenticated by possessing a private key matching the public key. Public keys can be publicly posted. Private keys must be kept private and closely guarded:

cat ~/.ssh/authorized\_keys

Get the amount of disk space that is free:

df -h

**Lab: Linux CLI - Files, Directories, Ownership, Permissions**

Show the current directory:

pwd

Change directory - cd by itself always takes you home:

cd

cd to a relative path name (does not start with / aka root or ~ aka home):

cd docker

. means current directory:

cd ./clusters

.. means parent directory:

cd ../images

Absolute path name, starts with / aka root:

cd /home/w205/docker

absolute path name, starts with ~ aka home:

cd ~/docker/images

ls lists all the files and directories in the current directory:

cd

ls

ls -l lists in long format, more info:

* type (d = directory, - = file, l = symbolic link)
* number of hard links
* user who owns the file
* group
* size in bytes
* last updated date time
* name

ls -l

ls -l docker

ls -l ./docker

ls -l ./docker/.././docker

ls -l user/labs/week\_06

Adding the -h will make it give files sizes in bytes, K, M, G, etc.:

ls -lh

ls -lh user/labs/week\_06

Adding the -a will make it give hidden directories and hidden files (in Linux what makes a directory or file hidden is if the name starts with a .)

ls -lha

ls -lha user/labs/week\_06

Permissions:

* r = read
* w = write
* x = execute (for files, it means execute as a program; for directories, it means cd to the directory)
* first set of rwx is for the user
* second set of rwx is for the group
* third set of rwx is for others (anyone who is not the user nor in the group)
* root is the super user that can read any file regardless of permissions

ls -lha ~/docker/mounts/postgres/

ls -lha ~/docker/mounts/postgres/data

When we want to use root privileges, we can logout and login as root (assuming we have the ssh key for root). As an alternative, we can be granted sudo privileges. sudo = "super user (root) do". We can put sudo in front of any command and run the command as root:

sudo ls -lha ~/docker/mounts/postgres/data

mkdir create a directory:

cd /home/w205/user/labs/week\_03

mkdir temp\_1

cd temp\_1

mkdir temp\_1\_1

mkdir /home/w205/user/labs/week\_03/temp\_1/temp\_1\_2

mkdir temp\_1\_3

mkdir temp\_1\_4

Recursively list out a directory tree:

ls -lhR

cp copies a file:

cp /home/w205/user/labs/week\_06/temp\_holidays\*.csv temp\_1\_1

cp /home/w205/user/labs/week\_06/temp\_products\*.csv temp\_1\_2

cp /home/w205/user/labs/week\_06/temp\_customers\*.csv temp\_1\_3

rm deletes a file:

cd temp\_1\_1

rm temp\_holidays\_2.csv

cd ..

rm temp\_1\_2/temp\_products\*header.csv

mv "moves" aka renames a file:

cd temp\_1\_1

mv temp\_holidays.csv temp\_holidays\_old.csv

rmdir deletes a directory, but it must be empty:

cd ..

rmdir temp\_1\_4

rmdir temp\_1\_3

Recursive rm -r can delete directories that are not empty:

rm -r temp\_1\_3

Permissions default mask:

umask

Permissions octal numbers:

* r = read = 2^2 = 4
* w = write = 2^1 = 2
* x = execute = 2^0 = 1
* 1 = --x
* 2 = -w-
* 3 = -wx
* 4 = r--
* 5 = r-x
* 6 = rw-
* 7 = rwx

Change permissions on a file or directory using octal notation:

chmod 765 temp\_1\_1

cd temp\_1\_1

chmod 431 \*.csv

cd ..

chmod -R 555 temp\_1\_1

Change permissions on a file or directory using UGO notation:

* u = user
* g = group
* o = other (warning: NOT owner!)

cd temp\_1\_1

chmod u=rwx,g=rx,o=r \*

chmod u-r,g+w,o+w \*

cd ..

Unlike most other OSs, Linux separates files from directory entries; Linux files are noted by inode.

cd temp\_1\_2

ls -lhi

Hard links: multiple directory entries for same file; file is only actually deleted when the last hard link is deleted; hard links must be in the same file system:

ln temp\_products\_3.csv temp\_products\_4.csv

ls -lhi

cat temp\_products\_3.csv

cat temp\_products\_4.csv

rm temp\_products\_3.csv

ls -lhi

rm temp\_products\_4.csv

ls -lhi

Symbolic links are just pointers from one directory to another or one file to another; symbolic links can go across file systems; deleting a symbolic link has no effect on the orginal file or directory:

cd ..

ln -s temp\_1\_2 temp\_1\_3

cd temp\_1\_2

ln -s temp\_products.csv temp\_prods.csv

rm temp\_prods.csv

Lab: Linux CLI - Commonly Used Commands

Reset permissions:

cd ~/user/labs/week\_03

chmod 775 temp\_1

chmod 775 temp\_1/\*

chmod 664 temp\_1/temp\_1\_1/\*.csv

Redirecting standard input, standout output, standard error:

* < redirects standard input
* > redirects standard output and overwrites
* >> redirects standard output and appends, does not overwrite
* 2> redirects standard error

cd temp\_1/temp\_1\_1

wc -l <temp\_holidays\_old.csv

date

date >date.txt

cat date.txt

date >>date.txt

cat date.txt

date >>date.txt

cat date.txt

date >date.txt

cat date.txt

ls not\_a\_file

ls not\_a\_file 2>error.txt

cat error.txt

less is a utility to view a file; man pages that we used earlier uses less; the older utility was called more, so they say that "less is more":

seq 10000

seq 10000 >numbers.txt

less numbers.txt

head returns the first few lines of a file; optionally pass the number of lines to return:

head numbers.txt

head -100 numbers.txt

head temp\_holidays\_old.csv

tail returns the last few lines of a file; optionall pass the number of lines to return:

tail numbers.txt

tail -100 numbers.txt

tail temp\_holidays\_old.csv

grep matches a pattern in a file and returns lines that match the pattern:

grep 871 numbers.txt

grep er temp\_holidays\_old.csv

grep -v returns the opposite, the rows that do NOT match:

grep -v er temp\_holidays\_old.csv

| is a pipeline; Linux spawn a process for each command; Linux creates an "anonymous pipe" aka "un-named pipe" between the processes; Linux routes standard output to pipes and routes standard input from pipes for intermediate commands

grep 817 numbers.txt

grep 817 numbers.txt | wc -l

grep 817 numbers.txt | grep -v 5

grep 817 numbers.txt | grep -v 5 | wc -l

Create a couple of files of random numbers:

echo 5 >file\_1.txt

echo 17 >>file\_1.txt

echo 16 >>file\_1.txt

echo 81 >>file\_1.txt

echo 92 >>file\_1.txt

echo 35 >>file\_1.txt

echo 21 >>file\_1.txt

echo 16 >>file\_1.txt

echo 5 >>file\_1.txt

echo 35 >>file\_2.txt

cat file\_1.txt

echo 5 >file\_2.txt

echo 17 >>file\_2.txt

echo 16 >>file\_2.txt

echo 82 >>file\_2.txt

echo 92 >>file\_2.txt

echo 35 >>file\_2.txt

echo 23 >>file\_2.txt

echo 16 >>file\_2.txt

echo 5 >>file\_2.txt

cat file\_2.txt

diff finds the differences between two files

diff file\_1.txt file\_2.txt

cp file\_1.txt file\_3.txt

diff file\_1.txt file\_3.txt

sort a file:

sort file\_1.txt

sort -g file\_1.txt

sort file\_2.txt

sort -g file\_2.txt

uniq removes duplicates from a file, however, the file MUST be sorted going into uniq:

uniq file\_1.txt

sort -g file\_1.txt | uniq

wc -l file\_1.txt

sort -g file\_1.txt | uniq | wc -l

uniq with -c option will count the number of duplicates, as before the file must be sorted going into uniq:

sort -g file\_1.txt | uniq -c

gzip compresses a file using gnu zip; you will find it does much faster and smaller compression than zip:

gzip numbers.txt

Use the -d option to uncompress

gzip -d numbers.txt.gz

tar stand for tape archive; in the modern usage, it can create a single file that holds a directory tree similar to a zip file:

cd ~/user/labs/week\_03

tar cvf temp.tar temp\_1

To view the contents of a tar, use the tvf option:

tar tvf temp.tar

To extract a tar tile, use the xvf option:

rm -r temp\_1

tar xvf temp.tar

We can also compress a tar using the cvfg which will use gzip to compress; some use the file extension .tar.gz, while others use .tgz:

tar cvfz temp.tar.gz temp\_1

tar xvf temp.tar.gz

tar cvfz temp.tgz temp\_1

tar xvf temp.tgz

find will walk a directory structure and find directories and files; it has a lot of options!:

find . -name \*.csv

xargs will read a list of files from standard input and run the command that follows it on each file individually; commonly used with find and a pipe:

find . -name \*.csv | wc -l

find . -name \*.csv | xargs wc -l

file will tell you what type of file a file is:

file temp\_1/temp\_1\_1/\*

find ~/user/labs | xargs file

top shows the top processes in terms of CPU usage and memory usage; type a q to quit top:

top

**Lab: Linux CLI - vi Editor**

Linux (and it's predecessor Unix) command line has a bad reputation for being very cryptic, unintuitive, un-user friendly, etc. Frankly, much of that criticism is legitimate.

Most would agree that the most difficult, cryptic, unituitive, un-user friendly part of Linux is the vi (or vim) editor. A lot of Linux and Unix command line aversion probably comes from the vi editor.

vi works in a very strange way compared to modern editors such as notepad on Windows. The reason that vi is so strange is for historical reasons. In the 1970's when most computers were on punched cards, Unix had terminal with text based screens. Computers at that time had very small CPU and memory. vi was designed on top of the streams system in Unix, and more specifically, the streams editor, or sed. The advantage of vi (and the sed engine underneath it) is that is can edit very large files with only a small piece of the file needing to be in memory.

The first thing you notice about vi is that if you bring up a file for editing, you cannot type and have what you type show up. As part of the way it saves memory, it has 3 modes:

* movement mode - the default mode vi will be in when you bring up a file; allows you to move around in the file, delete, copy, etc., but not type in new content
* insert mode - allows you to type in new content, but only in the current position; if we want to insert in multiple locations, we have to keep switching to movement mode to move, insert mode to insert, movement mode to move, etc.
* command mode - allows you to type a sed command

Why is vi still in use? Here are some possible reasons:

* Linux culture of wanting to keep things cryptic - somehow thinking that cryptic equals sophistication, rather than thinking user friendly equals sophistication
* We have always used vi
* vi is always available; other options may not be available
* Free alternatives to vi have cybersecurity issues
* Desktop alternatives are in wide spread use

Linux alternatives: emacs, nano, pico, etc. But these are not always available, probably because no one wants the liability for cybersecurity risks.

Desktop alternatives: For example, on Windows, WinSCP is a GUI Windows application that allows you to use a directory and file browser, edit files locally on Windows, and have them automatically be transferred to and from Linux. In addition to WinSCP, there are lots of commercial products. These products tend to be costly. Free desktop programs are all over the place, but beware of trojan horses. A lot of these give you a free product, but they contain malware, or harvest your SSH keys (which they must access) and phone them home for later hacking.

Remote development environments: Even more powerful than desktop alternatives are remote development environments. These environments not only allow you to remotely edit files, but also remotely execute code in a debugger. For example, for Python, there is PyCharm. You run the development environment on Windows, edit files in their IDE (integrated development environment) in Windows, and even remotely run your code in a debugger from Windows.

vi not a good solution for doing a lot of editing. The purpose of this lab is just to teach you the basics of vi, so you can use it for small editing when you have to, such as when it's the only editor available.

Basic commands:

* vi filename
* Movement mode: arrow keys, page up, page down, 0 beginning of line, $ end of line, x delete character, dd delete line
* Movement mode to insert mode: i for insert, a for append, note that INSERT appears lower left
* Insert mode to movement mode: ESC key
* Movement mode to command mode: : note that hitting the enter key will run the command and return to movement mode
* Save changes: :w
* Save a file and exit: :wq
* Quit: :q
* Quit with unsaved changes: :q!

**Lab: BASH Shell Commonly Used Features**

Multiple shells are available in Linux:

* sh (Bourne Shell) - original shell from the 1970s developed by Stephen Bourne at Bell Labs
* csh (C Shell) - Developed in the late 1970s by Bill Joy, a graduate student at Berkeley, similar to C programming language
* ksh (Korn Shell) - David Korn at Bell Labs
* bash (Bourne Again Shell) - most widely used modern shell, keeps a lot of the design of sh and extends it
* Others: tcsh (TC Shell), zsh (Z Shell)
* Python is also an alternative!

Since bash is the most widely used, we will use bash.

You will notice several hidden files in the home directory related to bash:

* .bash\_history - a history of commands you have entered on the command line
* .bash\_logout - run when you logout
* .bash\_profile - run when you login
* .bashrc - run when you create a non-login shell; it's a bit complicated to know when we generate a login shell versus a non-login shell, so we typically run .bashrc from .bash\_profile and put most of our customizations there

cd

ls -la | grep bash

vi .bash\_profile

vi .bashrc

When we type a command on the Linux command line, searches for the command in the path:

echo $PATH

Commands must have execute permission set. Note that . (current directory) is not in the path. This is for security reasons. To run a command or script in the current directory, use ./script.bash

Alias's allow us to run command without using the path.

alias

We can find where a command or script is located; which uses the path; whereis searches for typical locations even if not in path:

which ls

which bash

whereis ls

whereis bash

We can set environment variables using the shell; variables are only set in the current shell unless we export them:

env

MY\_VARIABLE=5

export MY\_VARIABLE

echo $MY\_VARIABLE

env | grep MY\_VARIABLE

Create a small example bash script:

cd ~/user/labs/week\_03

vi test.bash

#!/usr/bin/bash

export TEST\_VARIABLE=500

The first line #! is called "hash bang" or "shebang"; it tells Linux which shell to use to execute the script. Note that in Linux terms, Python is considered a shell and requires this also.

To run the script, we could specifically call it from bash:

bash test.bash

We could run it by setting the executable permission and calling it by name; recall that the current directory is not in the path, so we have to use ./test.bash

chmod u+x test.bash

test.bash 🡨 will not work

**./test.bash**

Let's see if our variable is present in the environment:

echo $TEST\_VARIABLE

env | grep TEST\_VARIABLE

Running a script creates a new shell, runs the script, and then exists the shell; our variable was set, but in another shell, not our shell; sourcing a script means to run a script in the current shell; we can source with the word source or with . space:

source test.bash

echo $TEST\_VARIABLE

env | grep TEST\_VARIABLE

. ./test.bash 🡨 another way to **source** it!

echo $TEST\_VARIABLE

env | grep TEST\_VARIABLE

Review the scripts in the scripts directory:

cd ~/scripts

crontab allows you to schedule commands or scripts or programs to run; one caveat, it does NOT run your .bash\_profile nor .bashrc! the format is:

* minute, 0 to 59
* hour, 0 to 23
* day of month, 1 to 31
* month, 1 to 12
* day of week, 0 to 6, 0 = Sunday
* command

Special macros:

* @yearly command => 0 0 1 1 \*
* @daily command => 0 0 \* \* \*
* @hourly command => 0 \* \* \* \*
* @reboot command => runs when the VM is started, booted, rebooted

Examples:

* \* \* \* \* \* command => every minute
* 30 8 10 6 \* command => 8:30am on June 10
* 0 11,16 \* \* \* command => 11am and 4pm every day
* 0 10-14 \* \* \* command => 10am, 11am, 12 noon, 1pm, 2pm every day
* 0 10-14 \* \* 1-5 command => 10am, 11am, 12 noon, 1pm, 2pm, Monday through Friday
* \*/10 \* \* \* \* command => every 10 minutes

crontab -l

crontab -e 🡨 editing crontab

**Lab: GitHub - Typical Workflow Example**

Create a new private repository:

GitHub => upper right dropdown => Your repositories => upper right green button New

* Repository name => ucb\_mids\_w205\_project\_test
* Choose Private
* Check Add a README file
* Click green button Create repository

Copy an https URL for the repo to clone it by clicking on the green button dropdown Code => choose HTTPS => click the copy button

In the Linux command line:

cd ~/user/projects

git clone https\_link\_copied\_above

cd repo\_directory\_name

git status

git branch project

git checkout project

vi my\_file\_1.txt

git add my\_file\_1.txt

git commit -m "initial add"

git push origin project **\*\* project is branch name. if we want to push it to main then we need to “main” instead of “project”**

Now update the file, create another file, stage, commit, and push

vi my\_file\_1.txt

vi my\_file\_2.txt

git add my\_file\_1.txt

git add my\_file\_2.txt

git commit -m "updates"

git push origin project

Once you are done create a pull request in GitHub

## ****Space management of your VM****

In the week 3 labs, we saw the command:

df -h

Which produces output similar to:

Filesystem Size Used Avail Use% Mounted on

devtmpfs 3.9G 0 3.9G 0% /dev

tmpfs 3.9G 0 3.9G 0% /dev/shm

tmpfs 3.9G 492K 3.9G 1% /run

tmpfs 3.9G 0 3.9G 0% /sys/fs/cgroup

/dev/xvda1 100G 19G 82G 19% /

tmpfs 798M 0 798M 0% /run/user/1001

In the week 3 labs, we also saw a Python program that can help with space management:

cd ~/scripts

sudo ./walk\_directory\_tree.py

Which will show you the options below:

walk\_directory\_tree.py

usage: walk\_directory\_tree directory option

option 1: print directories in alphabetical order

option 2: print directories and sizes sorted ascending

option 3: print directories and sizes sorted descending

option 4: print files in alphabetical order

option 5: print files and sizes sorted ascending

option 6: print files and sizes sorted descending

## ****Quiz****

**Q) In Linux, how do we designate a file or directory as hidden?**

A) Name the file or directory starting with a dot

\*\* In Linux, a file or directory is designated as hidden by naming it with a leading dot (.) at the beginning of its name. This is a convention followed in Unix-like operating systems. Files or directories that start with a dot are not normally displayed by default when listing files in a directory. For example, a file named .example.txt would be a hidden file.

**Q) In Linux, we have a file with mode 754. Which of the following are correct permissions (check all that apply)?**

A) Group can read

Group can execute

World can read

User can read

User can write

User can execute

\*\* In Linux, file permissions are represented by a three-digit octal number. Each digit represents a different set of permissions: for the user (owner), group, and others (world), respectively. The mode **754** would break down as follows:

* **7** for the user: This is the sum of **4** (read), **2** (write), and **1** (execute), so the user can read, write, and execute.
* **5** for the group: This is the sum of **4** (read) and **1** (execute), so the group can read and execute.
* **4** for the world: This represents **4** (read), so the world can read.

**Q) Consider links in Linux. Which of the following statements are true (check all that apply)?**

A) A file is deleted when the last hard link to the file is deleted

Symbolic links can be made across different file systems

Multiple hard links can point to the same inode

\*\*

1. **A file is deleted when the last hard link to the file is deleted:** True. In Linux, a file's data is only deleted when the last hard link to its inode is removed. As long as there's at least one hard link, the data remains intact.
2. **Symbolic links can be made across different file systems:** True. Symbolic links (or soft links) are like shortcuts and can point to a file or directory in a different file system.
3. **Multiple hard links can point to the same inode**: True. This is the essence of hard links; multiple hard links can refer to the same inode, and hence the same data on the disk.

**Q) Consider the following Linux command “ls -l | grep source | sort | wc -l”. Which of the following statements are true (check all that apply)?**

A) All processes run in parallel

Four separate processes are spawned

\*\*

1. **All processes run in parallel:** This is partially true. In a pipeline, each command is run in a separate process. These processes can run concurrently, but they are not strictly parallel in the sense that they depend on the output of the previous command. The next command in the pipeline processes the data as it becomes available from the previous command.
2. **Four separate processes are spawned:** True. The command involves four separate processes. **ls -l** is one process, **grep source** is the second, **sort** is the third, and **wc -l** is the fourth.

**Q) Consider processes and threads in Linux. Which of the following statements are true (check all that apply)?**

A) Each thread has its own stack and shares a heap with other threads in the same process

Each process has its own stack and its own heap

\*\*

1. **Each thread has its own stack and shares a heap with other threads in the same process:** True. In a process, each thread has its own stack space but shares the heap and other memory segments (like data and text segments) with other threads in the same process.
2. **Each process has its own stack and its own heap:** True. Every process in Linux has its own separate memory space, which includes its own stack and heap. This isolation ensures that processes do not interfere with each other's memory.

**Q) In Linux, we want to run a script in the current shell without creating a new shell. Which of the following statements are true (check all that apply)?**

A) Running the script will create a new shell

We must source it instead of running it

Environment variables set will persist in the current shell

\*\*

1. **We must source it instead of running it:** True. To run a script in the current shell without creating a new shell, you need to **source** it. Sourcing a script (**source scriptname** or **. scriptname**) executes it in the current shell environment.
2. **Environment variables set will persist in the current shell:** True. When you source a script, any environment variables or changes made in the script will affect the current shell. This means that any environment variables set in the script will persist in the current shell after the script has finished executing.

**Q) In git CLI, which of the following statements are true regarding commits (check all that apply)?**

A) A commit will commit only staged files

A commit will only commit locally, not to GitHub

\*\*

1. **A commit will commit only staged files:** True. In Git, a commit captures the state of the staged files at that point in time. Only changes that have been added to the staging area (with **git add**) will be included in the commit.
2. **A commit will only commit locally, not to GitHub:** True. A commit in Git is a local operation. It records changes to the repository on your local machine. To update a remote repository like GitHub, you need to push the commits using **git push**.

**Q) git push origin project**

A)

* **git push**: This command is used to upload local repository content to a remote repository.
* **origin**: This is the default name Git gives to the remote repository from which you cloned. It's essentially an alias for the URL of the remote repository.
* **project**: This is the name of the branch you want to push. By specifying **project**, you are indicating that you want to push the changes from your local **project** branch to the remote **project** branch.