Objectives

This minicourse will explain

1. the Linux environment,
2. essential shell commands,
3. working text editors,
4. running Python programs from the shell,
5. file permissions,
6. I/O redirection, and
7. remote access tools

Introduction

Like macOS and Windows, GNU/Linux (or "Linux") is an operating system with windows, programs, and web browsers. Files are located within directories (folders), which may themselves be located within other directories. You may access these features through a GUI (Graphical User Interface) using a mouse, but as your perform complicated tasks this form of interaction becomes inefficient.

However, a computer can be entirely controlled by keyboard using the shell. This minicourse teaches you how to use the shell for common operations.

The Shell

Imagine deleting all mp3 files over 5 MiB that haven't been listening to in the past year. This is substantial task to carry out with a GUI, but is trivial using the shell.

Click *Application* (top left), type "terminal", and click the *terminal* icon to open a terminal. Alternatively, press Ctrl+Alt+T. A terminal will open, showing a shell with the prompt

```
username@computer:~$
```

Where `username` is your CNetID and `computer` is the machine's name. Start typing and notice the characters begin to the right of `$`.

Terminals can administer multiple shells. Ubuntu uses the *Bourne Again Shell* (bash) as the default shell, but there are other options:

- the Korn Shell (ksh),
- the C Shell (csh), and
- tsch.

File System Navigation

Files are stored in directories, and directories can be stored in other directories. Just like macOS and Windows, there are special directories for Linux:

- the root directory `/` on Linux is the same as `/` on macOS and `C:\\` on Windows, and
- the home directory `/home/username/` in Linux is the same as `/Users/username/` on macOS and /C:\Documents and Settings\username` on Windows.

The lab's computers are connected by a network file system: there is one very large shared hard drive, and your files are available from any computer and all of our directories live in the same space.

The following diagram illustrates how Linux organizes the file system:

```
/
    /home
        /home/username
            /home/username/Desktop
            /home/username/Documents
        /home/mromney
        /home/bobama
    /bin
    /stage
```

Take `username` to be your CNetID, your actual username.

Show Files

The shell initializes in your home directory (`/home/username/`), the unique directory assigned to your account. Invariant of the computers you use in CSIL, the shell will alway initialize in your home directory, and all files you interacted with in previous session will still be available.

Here are two commands for working with directories:

pwd
Print the absolute pathname of the current working directory. `ls`

List information about files (in the current directory by default).

Let's run these commands:

```
$ pwd
/home/username/
$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

Observe that these are the same files you see when you open your home folder with the GUI. The information provided by the shell is the same; the only difference is how it is obtained and displayed.

Changing Directory

`cd /home/username`
Move to `/home/username`.

`cd ..`
Return to the containing directory.

`cd`
Return to your home directory.

With a GUI, click on folders and the *back* arrow to navigate the file system. With a shell, use `cd` (Change Directory):

```
$ cd
$ pwd
/home/username
$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos

$ cd Documents
$ pwd
/home/username/Documents

$ cd ..
$ pwd
/home/username

$ cd ..
$ pwd
/home
```

When a directory contains no files, `ls` doesn't output anything.

~ signifies your home directory, so

```
$ cd /home/username
```

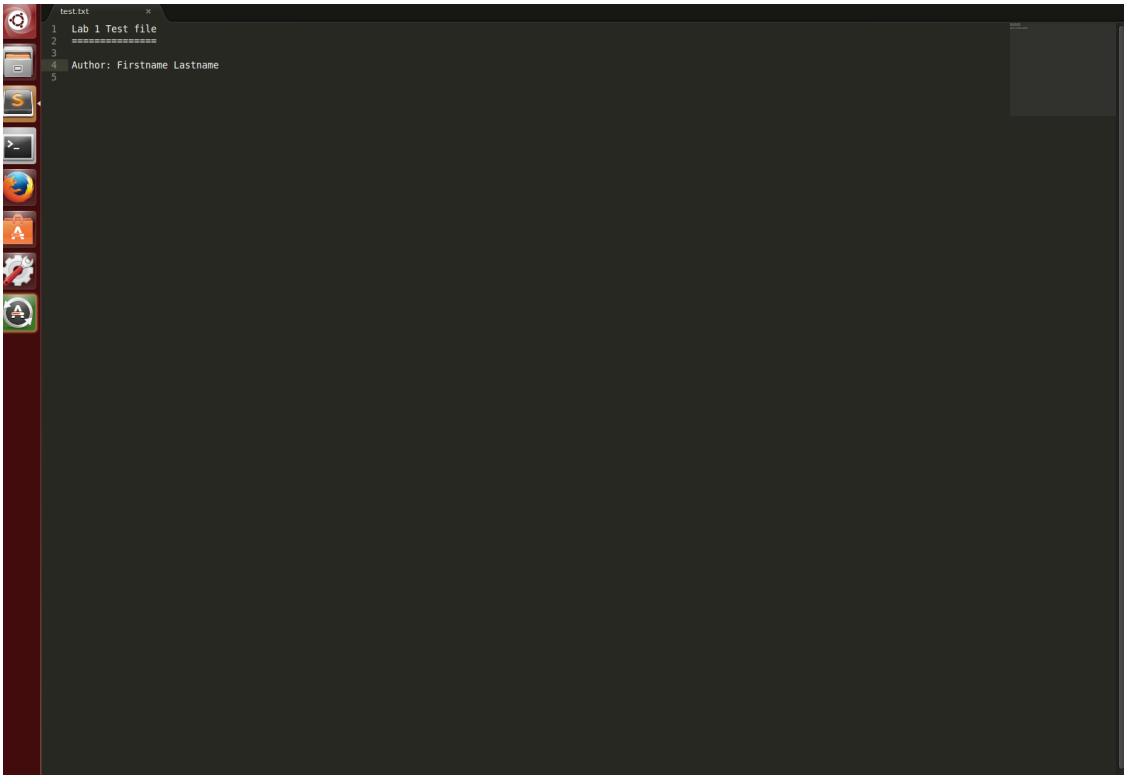
is identical to

```
$ cd ~
```

Paths beginning with `/` are called *absolute* paths, specifying the destination independent of your current location (like `/home/username`). All other paths are *relative* paths, specifying the destination in terms of moving from the current location (like `Documents`). The difference is akin to "go to 1600 Pennsylvania Avenue" (an *absolute* path) compared with "go 10 miles north" (a *relative* path).

Running `cd` with no arguments returns you to your home directory, always.

Using an editor



We can view and edit files with text editors: today we use Sublime Text. Create a file to edit:

```
$ touch test.txt
```

Now open the file with Sublime Text:

```
$ sublime-text test.txt
```

You should see an empty file. You can navigate using the arrow keys, and save your changes using Ctrl+s.

Exercises

1. Add your name to the file,
2. Saving the file,
3. Close and reopen the file and ensure your name is still there, and
4. Close `sublime-text`.

Copy, Move, Remove, and Make Directory

`cp <source> <destination>`
Copy `<source>` to `<dest>`, or multiple sources to a directory.

`mv <source> <destination>`
Rename `<source>` to `<dest>`, or move sources to a directory.

`rm <file>`
Remove `<files>`.

`mkdir <directory-name>`
Create the `<directory-name>`, if it does not already exist.

To copy `test.txt` to `copy.txt`:

```
$ cp test.txt copy.txt
```

To copy `test.txt` to the `Documents` directory:

```
$ cp test.txt Documents
```

Exercises

1. Execute the above copy commands and check the results with `ls`.

Use `mv` similarly to `cp`, but remember it doesn't keep the source file.

2. Rename `copy.txt` to `copy2.txt`. Check with `ls`.

Make new directories with `mkdir directory-name`:

3. Make a `backups` directory.

Directories can act as destinations.

4. Copy `copy2.txt` to `backups`.

List a directory's files with `ls directory-name`:

5. Verify the previous exercise by listing the files in `backups`.

Remove files with `rm filename`:

6. Remove `copy2.txt` from `backups`.

To copy (or remove) entire directories, add the `-r` flag:

```
$ cp -r backups backups2
$ rm -r backups
```

Don't shoot yourself in the foot by deleting something important.

Python

`python3 file.py`
Run `file.py` as a Python3 program.

Create a new python file by running `sublime-text` and copying

```
print("Hola mundo.")
```

into the file, saving it as `hola_mundo.py`, and exiting Sublime Text. Run this with

```
$ python3 hola_mundo.py
```

Note: Python2 and Python3 are different languages, so make sure you use `python3` instead of `python` at the shell.

Editing and Executing

Let's modify and re-execute `hola_mundo.py`.

```
$ sublime-text hola_mundo.py
```

Change `"Hello, World"` to "Hello, " followed by your name. If your name were Maite Perroni the line would become

```
print("Hola Maite Perroni")
```

Now save the file and rerun the program. Is your shell not responding? Close Sublime Text and run

```
$ sublime-text hola_mundo.py &
```

Now your shell will allow you to continue typing commands.

File Permissions

File permissions place restrictions accessing files. They dictate who can view (*read*), edit (*write*), and run (*execute*).

All files are owned by a user; each user is a member of 1 or more groups. (To see your groups, execute `groups` from the shell.) File permissions have three scopes:

User
The file's owner.

Group
The file's group. Allows group members to access the file.

Others
For everyone who is not the owner or in the file's group.

When a user satisfies multiple scopes the most personal one is chosen: *user*, then *group*, then *others*. Additionally, each scope has three types of permissions:

read
Can I view the file? When set for directories, this only allows the content *names* to be viewed, not the contents themselves.

write
Can I edit the file? When set for directories, this allows creating, deleting, and renaming files in the directory.

execute
Can I run the file (or program)? When set for directories, this allows accessing file contents and metadata.

Each permission has a unique value:

- read = $2^2 = 100_2$,
- write = $2^1 = 010_2$, and
- execute = $2^0 = 001_2$.

The file's total permission is calculated by adding the individual permissions. If `test.txt` has read and write permissions, then the total permissions is $100_2 + 010_2 = 110_2 = 6$.

You can describe the permissions for all scopes at once with 3 numbers: e.x. 761 describes a file with:

- (user scope) read, write, and execute permissions,
- (group scope) read and write permissions, and
- (other scope) execute permissions.

To list a files information (including permissions) run `ls -l`:

```
$ ls -l /usr/bin/python3.4
-rwxr-xr-x 1 root root 3709944 Oct 14  2015 /usr/bin/python3.4
```

The file's owner is named `root`, which is the account that can access all commands and files on Linux. (Other accounts may borrow the power of `root`.) The file's group is also `root`. The permissions are `-rwxr-xr-x`. These permissions are listed from most to least personal (user, group, others). For this file

- the owner can read, write, and execute,
- the group can read and execute, and
- other users can read and execute.

Exercises

Files and directories you create have your username for both the *user* and the *group*. On our machines, new files give read and write permissions to *user* and *group* and no permissions to *other*, while new directories have read, write, and execute permissions for *user* and *group*.

1. Verify this by running `ls backups/copy2.txt` and `ls -d backups` in your home directory.

`ls -d` lists the directory instead of its contents. The permission string for `backups` begins with `d`, telling us `backups` is a directory. (Regular files begin with `-` instead.)

Changing Permissions, Owners, and Groups

`chmod <permissions> <path-name>`
Set a file's permissions.

`chmod <changes> <path-name>`
Modify a file's permissions.

`chown <username> <path-name>`
Change a file's *owner*.

`chgrp <group> <path-name>`
Change a file's *group*.

`chmod` can be used to change permissions in two ways: using a 3 digit octal number to set permissions, and toggling specific permissions.

```
$ echo "bonjour" > testfile
$ ls -l testfile
-rw-rw-r-- 1 username username 7 Aug 23 11:22 testfile
$ cat testfile
bonjour
$ chmod 222 testfile
$ cat testfile
cat: testfile: Permission denied
$ chmod u+r testfile
```

`chmod 222` sets only *write* permissions for all scopes, and `chmod u+r` gives *read* permissions to *user*.

Exercises

1. Change `testfile` to allow *read* and *write* permissions for *group* and *read* access for *other*. Verify this with `ls -l testfile`.
2. Remove *write* permissions for `testfile` and check again with `ls -l`.

Wild Cards

Wilcards allow us to describe multiple strings with just one:

```
$ ls *.txt
```

lists all files that end in `.txt`. The wildcard * represents any string (even the empty string).

Don't use the wildcard without thinking: (don't run this)

```
$ rm *
```

Deletes **every** file in the current directory.

Exercises

1. Navigate to root: what does `ls -d li*` output? `ls -d li*/*`?
2. Return to your home directory: what does `ls D*` output?

Environment Variables

`printenv`
Print the environment variables.

`export`
Define an environment variable.

We use variables for convience, like when we have a string we use frequently.

```
$ printenv
```

Outputs the current variables you can access. What happens when you run it in your shell?

Define new variables in all capital letters using = and `export` like

```
$ export NEWVARIABLE="~/some/path"
```

Do not add spaces around =, but feel free to add spaces in quotes. Thus,

```
$ export NEWVARIABLE=" ~/some/path"
```

is valid too.

You can also defined variables like

```
$ NEWVARIABLE="~/some/path"
```

which makes it local to the current shell. Other shells cannot access `NEWVARIABLE` in this case. Using `export` lets us save the variables.

We can access a variable's value by prepending `$`

```
$ echo NEWVARIABLE
NEWVARIABLE
$ echo $NEWVARIABLE
~/some/path
```

Exercises

1. Define a variable (with `export`) and echo it.
2. Open another shell and test the same echo command.

Man Pages

Man (manual) pages document the Unix environment: programs, functions, standards and conventions, and abstract concepts. Running `man` with a command name gives you more information:

```
$ man ls
```

Outputs the manual page for `ls`: description, flags, usage instructions, and other information.

You may also search the manual by keyword, using the `-k` flag:

```
$ man -k printf
```

Outputs the names of all manual pages containing the word `printf`.

--

Running Commands Sequentially

To print the current directory and then list all files inside, you could run

```
$ pwd
$ ls
```

But you could also run

```
$ pwd ; ls
```

`pwd` first runs to completion, then `ls` runs. The two example are equivalent, but the second offers a faster shortcut. Unlike with variables, the shell doesn't care about whitespace around `;`. These examples are all valid:

```
$ pwd;ls
$ pwd ;ls
$ pwd; ls
$ pwd ; ls
```

Useful Keyboard Shortcuts

At the shell, Ctrl+P let's you access previous commands (so you don't have to retype). If you go back, you may also go forward with Ctrl+N. If this is too hard to remember, you can also use the *up* and *down* arrow keys.

Here more shortcuts:

- Ctrl+A move to the beginning of the line.
- Ctrl+E move to the end of the line.
- Ctrl+U erase from your cursor to the beginning of the line.
- Ctrl+K erase from your curse to the end of the line.

You can save lots of time by scrolling back, editing, and rerunning old commands.

Redirection

If you see a command you don't understand, use `man` to learn more.

`pwd`, `ls`, and `cat` output to the screen by default. However, sometimes we want to save the output in a file. We do this with `>`:

```
$ cd
$ touch diamond.txt
$ ls > diamond.txt ; cat diamond.txt
Desktop
diamond.txt
Documents
Downloads
Music
Pictures
Public
Templates
Videos
```

```
$ echo "Hola mundo" > documento.txt ; cat documento.txt
Hola mundo
$ cat documento.txt > documento2.txt ; cat documento2.txt
Hola mundo
```

Remember:

1. Redirecting to a non-existant file creates it.
2. Redirecting to an existing file replaces it.

Using >> (append) instead of > (overwrite) lets you add to an existing file, instead of replacing it.

We can also have programs recieve input from files, using <:

```
$ cat < documento.txt
Hola mundo
```

Linux has three standard I/O streams:

Standard Input (stdin)
Where programs get their input. This is usually mapped to the keyboard.

Standard Output (stdout)
Where programs send their output. This is usually mapped to the screen.

Standard Error (stderr)
Where programs send their error messages. This is usually mapped to the screen as well.

The reason that standard error and standard output are different, even though they usually output to the same place, is that you can redirect standard error to, say, a log file while still letting program output go to the screen.

The operators >, >>, and < that you learned about redirect standard input and output to files instead of the usual keyboard/screen.

Piping

Piping allows easy shorthand for one program to receive another program's output as input:

```
$ program1 | program2
```

is shorthand for something like

```
$ program1 > temporary.txt ; program2 < temporary.txt
```

For example, assuming you didn't know of `mkdir`, how could you use `man` and `grep` (which searches through text) to find candidate commands?

```
$ man -k "create directory"
```

doesn't give any results, while

```
$ man -k "directory"
```

gives way too many. However, running

```
$ man -k "directory" | grep "create"
mkdir (2)             - create a directory
mkdirat (2)           - create a directory
mkdtemp (3)           - create a unique temporary directory
mkfontdir (1)         - create an index of X font files in a directory
mklost+found (8)      - create a lost+found directory on a mounted Linux second extended
fil...
mktemp (1)            - create a temporary file or directory
pam_mkhomedir (8)     - PAM module to create users home directory
update-info-dir (8)   - update or create index file from all installed info files in
directory
vgmknodes (8)         - recreate volume group directory and logical volume special files
```

gives us what we want.

Exercises

1. Use `cat` and `tail` to display the last 10 lines of `/var/log/syslog`.
2. Do the same without using `|`.

Remote Access

`ssh` and `scp` are two tools for accessing remote computers through the command line. `ssh` allows you remotely run commands, and `scp` allows you to transfer files between your computer and the remote one.

SSH

```
$ ssh username@domain
```

begins an SSH session, allowing access to all files and programs on the remote computer. To SSH into the Linux computers, run

```
$ ssh CNET@linux.cs.uchicago.edu
```

where `CNET` is your CNetID. Try this now, so you can ask us questions if you have trouble.

To exit an SSH session, run

```
$ exit
```

`ssh` is installed on Linux and macOS, while PuTTY can be installed on Windows.

SCP

```
$ scp user1@host1:/path/to/source-file user2@host2:/path/to/destination-file
```

Copies a file from the computer `host1` to `host2`. To copy a file from your local computer, write the file path instead of the username, hostname, and filepath:

```
$ scp Documents/essay.pdf CNET@linux.cs.uchicago.edu:essay.pdf
```

copies `essay.pdf` to the Linux machines. If you won't change the filename, you can leave it out of the destination argument:

```
$ scp Documents/essay.pdf CNET@linux.cs.uchicago.edu:Documents/essay.pdf
```

is the same as

```
$ scp Documents/essay.pdf CNET@linus.cs.uchicago.edu:Documents/
```

SFTP

SFTP combines SSH and SCP to let you interactively transfer files between computers. An SFTP session starts similarly to SSH, but lets you use the commands `get` and `put` to transfer remote files.

Final Notes

When a program is misbehaving (or running indefinitely) use Ctrl+C to terminate the running program. You may need to type it multiple times to successfully terminate a program.

Typing Ctrl+D sends an end-of-input signal, telling the program that no more information is coming.