Linux Lab Introduction

- Objectives
- Linux
- Terminal/Shell
- Navigating the File System
- Show Files
- Change Directory
- Exercises
- Copy (`cp`), Move (`mv`), Remove (`rm`), and Make Directory (`mkdir`)
  - Exercises
- Using an Editor
  - Exercises
- Run a Python Program
- Edit and Run a Python Program
- Wild Cards (using an asterisk)
  - Exercises
- Useful Keyboard Shortcuts
- Running Commands Sequentially
- Redirection
- Piping
  - Exercises
- Man Pages
- File Permissions
  - Exercises
- Changing Permissions, Owner, & Group
  - Exercises
- Final Notes
- Log out

Linux Lab Introduction

Objectives

1. Learn basic terminal commands and how to work with a text editor
2. Become familiar with the Linux environment
3. Learn to run a Python program from the command-line
4. Learn about file permissions
5. Learn about redirection and pipes

Linux

Linux is an operating system much like OS X or Windows. It has windows, programs, web browsers, and so on. Files are stored in directories (folders) that, in turn, are stored in other directories. Although you can access Linux's features using your mouse, as you perform more and more complex tasks, you will find that using the mouse is ineffective. Linux allows us to interact with the computer entirely through text using a program called the terminal. (Macs provide a similar terminal application, and there are ways to use text-based commands on Windows too. But, Linux provides the lowest barrier to entry.) In this lab you will learn how to use the terminal to perform some basic operations in Linux. You will need these skills for the rest of your time at UChicago.

We show many examples of sample output below. The output you see when you run the commands may vary a bit. For example, most of you are not named "Gustav Martin Larsson".

Terminal/Shell

On your personal computer, you probably navigate your hard drive by double clicking on icons. While convenient for simple tasks, this approach is limited. For example, imagine that you want to delete all of the music files over 5 MB that you haven't listened to in over a year. This task is very hard to do with the standard double-click interface but is relatively simple using the terminal.

On the virtual desk, click the Application button (at the top left) and type "terminal" in the input box. Click the "terminal" icon to open the terminal window. (See Logging into the CS Department Virtual Desktop Server (https://classes.cs.uchicago.edu/archive/2020/fall/30121-1/Virtual-Linux-Desktop.pdf) for screen shots.)

A terminal window will open and you will see text of the form:

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

where `username` has been replaced by your CNetID and `computer` is the name of the virtual machine you happen to be using. This string is called the prompt. When you start typing, the characters you type will appear to the right of the `$`.

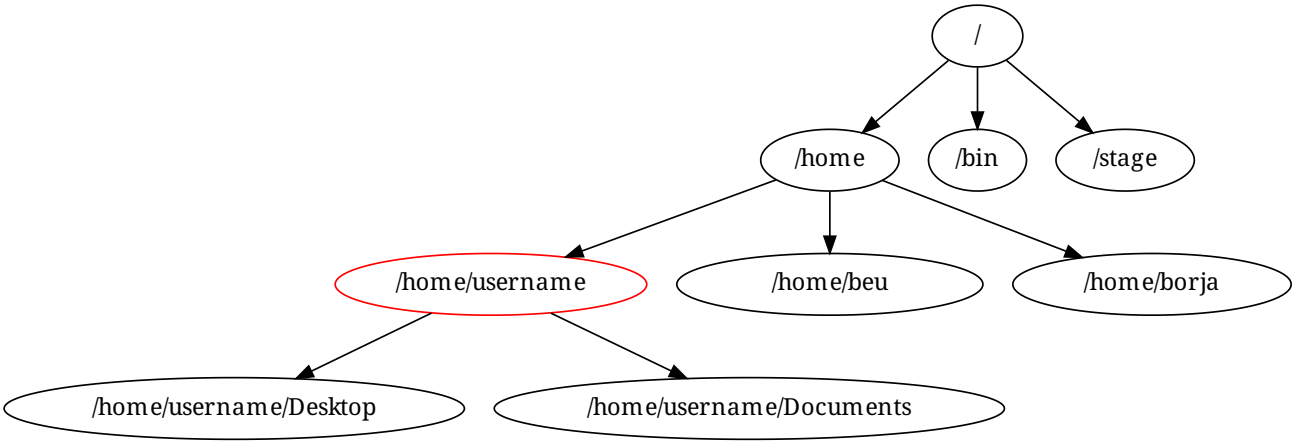
The program that runs within a terminal window and processes the commands the you type is called a *shell*. We use `bash`, which is the default shell on most Linux distributions, but there are other popular shells, such as `ksh`, `tcsh`, etc.

The procedure for completing this lab is as follows. For each section, read through the explanatory text and the examples. Then, try these ideas by doing the exercises listed at the bottom of the section.

Navigating the File System

Files in Linux are stored in directories/folders, just like in OS X/Windows. Directories can hold files or other subdirectories and there are special directories for your personal files, your Desktop, etc.:

Name	Linux	Mac	Windows
Root directory	/	/	C:\
Home directory	/home/username	/Users/username	C:\Documents and Settings\username



(../../_images/filesystem.username.svg)
The virtual desktop connects to a network file system. Effectively there is one very large shared hard drive. Your files are available from any of the vDesk servers and all of our directories live in the same space.

The figure above illustrates how Linux organizes the file system. Your own computer might have a slightly different organization (e.g., you might replace `/` with `C:`), but the idea is the same.

For the above and from this point forward, consider that the text "username" is replaced with your own actual username, which is just your CNetID.

Show Files

The terminal will start in your home directory, `/home/username/`, which is a special directory assigned to your user account. Any desktop that you get to via the CS vDesk server will automatically connect to your home directory and all files that you created or changed in previous vDesk sessions will be available to you.

Two very useful commands are `pwd` and `ls`:

`pwd`	Prints your current working directory - tells you where you are in your directory tree.
`ls`	Lists all of the files in the current directory.

The following is an example using these two commands in a terminal window:

```
username@computer:~$ pwd
/home/username/
username@computer:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
username@computer:~$
```

Try these commands yourself to verify that everything looks similar.

Notice that the directory path and list of files that you see if you open your home folder graphically are identical to those provided by `pwd` and `ls`, respectively. The only difference is how you get the information, how the information is displayed, and how easy it is to write a script that, say, processes all the Python files in a directory.

Change Directory

`cd <path-name>`	change to the directory path-name

`cd ..`	move up/back one directory
`cd`	move to your home directory

How can we move around in the file system? If we were using a graphical system, we would double click on folders and occasionally click the "back" arrow. In order to change directories in the terminal, we use `cd` (change directory) followed by the name of the destination directory. (A note about notation: we will use text inside angle brackets, such as `<path-name>` as a place holder. The text informally describes the type of value that should be supplied. In the case of `<path-name>` , the desired value is the path-name for a file. More about path-names later.) For example if we want to change to the `Desktop` directory, we type the following in the terminal:

```
cd Desktop
```

Here is an example of changing to the desktop directory in the terminal. We use `pwd` and `ls` to verify where we are and where we can go:

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

username@computer:~/Desktop$
```

Notice that after we `cd` into the `Desktop` the command `pwd` now prints out:

```
/home/username/Desktop/
```

rather than:

```
/home/username/
```

In the beginning, there are no files in the Desktop directory, which is why the output of `ls` in this directory is empty.

We can move up one step in the directory tree (e.g., from `/home/username/Desktop` to `/home/username` or from `/home/username` to `/home`) by typing `cd ..` Here "up" is represented by " `..` " In this context, this command will move us up one level back to our home directory:

```
username@computer:~/Desktop$ pwd
/home/username/Desktop/
username@computer:~/Desktop$ cd ..
username@computer:~$ pwd
/home/username/
```

Notice that the current working directory is also shown in the prompt string.

`~`	shortcut for your home directory

`.`	shortcut for the current working directory
`..`	shortcut for one level up from your current working directory

The tilde (~) directory is the same as your home directory: that is, `~` is shorthand for `/home/username`. Here's another useful shorthand: a single dot (`.`) refers to the current directory.

Usually when you use `cd`, you will specify what is called a *relative* path, that is, you are telling the computer to take you to a directory where the location of the directory is described relative to the current directory. The only reason that the computer knows that we can `cd` to `Desktop` is because `Desktop` is a folder within the `/home/username` directory. But, if we use a `/` at the *beginning* of our path, we are specifying an absolute path or one that is relative to the the "root" or top of the file system. For example:

```
username@computer:~$ pwd
/home/username/
username@computer:~$ cd /home/username/Desktop
username@computer:~/Desktop$ pwd
/home/username/Desktop
username@computer:~/Desktop$ cd /home/username
username@computer:~$ pwd
/home/username
```

These commands achieve the same thing as the ones above: we `cd` into `Desktop`, a folder within our home directory, and then back to our home directory. Paths that start with a `/` are known as *absolute paths* because they always lead to the same place, regardless of your current working directory.

Running `cd` without an argument will take you back to your home directory without regard to your current location in the file system. For example:

```
username@computer:~/Desktop$ cd
username@computer:~$ pwd
/home/username
```

To improve the readability of our examples, we will use `$` as the prompt rather than the full text `username@computer:~$` in the rest of this lab and, more generally, in the course going forward. Keep in mind, though, that the prompt shows your current working directory.

Exercises

Use `pwd`, `ls`, and `cd` to navigate to the `lab1` subdirectory.

Copy (`cp`), Move (`mv`), Remove (`rm`), and Make Directory (`mkdir`)

`cp` <source> <destination>	copy the source file to the new destination
`mv` <source> <destination>	move the source file to the new destination
`rm` <file>	remove or delete a file
`mkdir` <directoryname>	make a new empty directory

Sometimes it is useful to make a copy of a file. To copy a file, use the command:

```
cp <source> <destination>
```

where `<source>` is replaced by the name of the file you want to copy and `<destination>` is replaced by the desired name for the copy. An example of copying the file `test.txt` to `copy.txt` is below:

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

`<destination>` can also be replaced with a path to a directory. In this case, the copy will be stored in the specified directory and will have the same name as the source.

Move (`mv`) has exactly the same syntax, but does not keep the original file. Remove (`rm`) will delete the file from your directory.

If you want to copy or remove an entire directory along with its the files, the normal `cp` and `rm` commands will not work. Use `cp -r` instead of `cp` or `rm -r` instead of `rm` to copy or remove directories (the `r` stands for “recursive”):

Make sure you want to remove *everything* in the named directory, including subdirectories, *before* you use `rm -r`.

You can make a new directory with `mkdir directoryname`, where `directoryname` is the desired name for the new directory.

Exercises

Try the following tasks to practice and check your understanding of these terminal commands.

1. Execute the above copy command and use `ls` to ensure that both files exist.
2. Move the file `copy.txt` to the name `copy2.txt`. Use `ls` to verify that this command worked.
3. Make a new directory named `backups` using the `mkdir` command.
4. Copy the file `copy2.txt` to the `backups` directory.
5. Verify that step (4) was successful by listing the files in the `backups` directory.
6. Now that we have a copy of `test.txt` in the backups directory we no longer need `copy2.txt`. Remove the file `copy2.txt` in this directory.

It can be tedious (and, when you are tired, challenging) to spell directory or file names exactly, so the terminal provides an auto-complete mechanism to guide you through your folder explorations. To access this functionality simply start typing whatever name you are interested in the context of a command and then hit tab. If there is only one way to finish that term hitting tab will fill in the rest of the term, for instance, if we typed `ls b` and then hit tab it would automatically finish the word `ls backups` and then await our hitting enter. If there is MORE than one way to finish a term, like if we had another folder called `backups-old`, then hitting tab twice with cause the terminal to display all of the options available.

Training yourself to use auto-completion (aka tab completion) will save you time and reduce the inevitable frustration that arises from mistyping filenames when you are tired or distracted.

Using an Editor

List the files in the `lab1` directory. You should see the following:

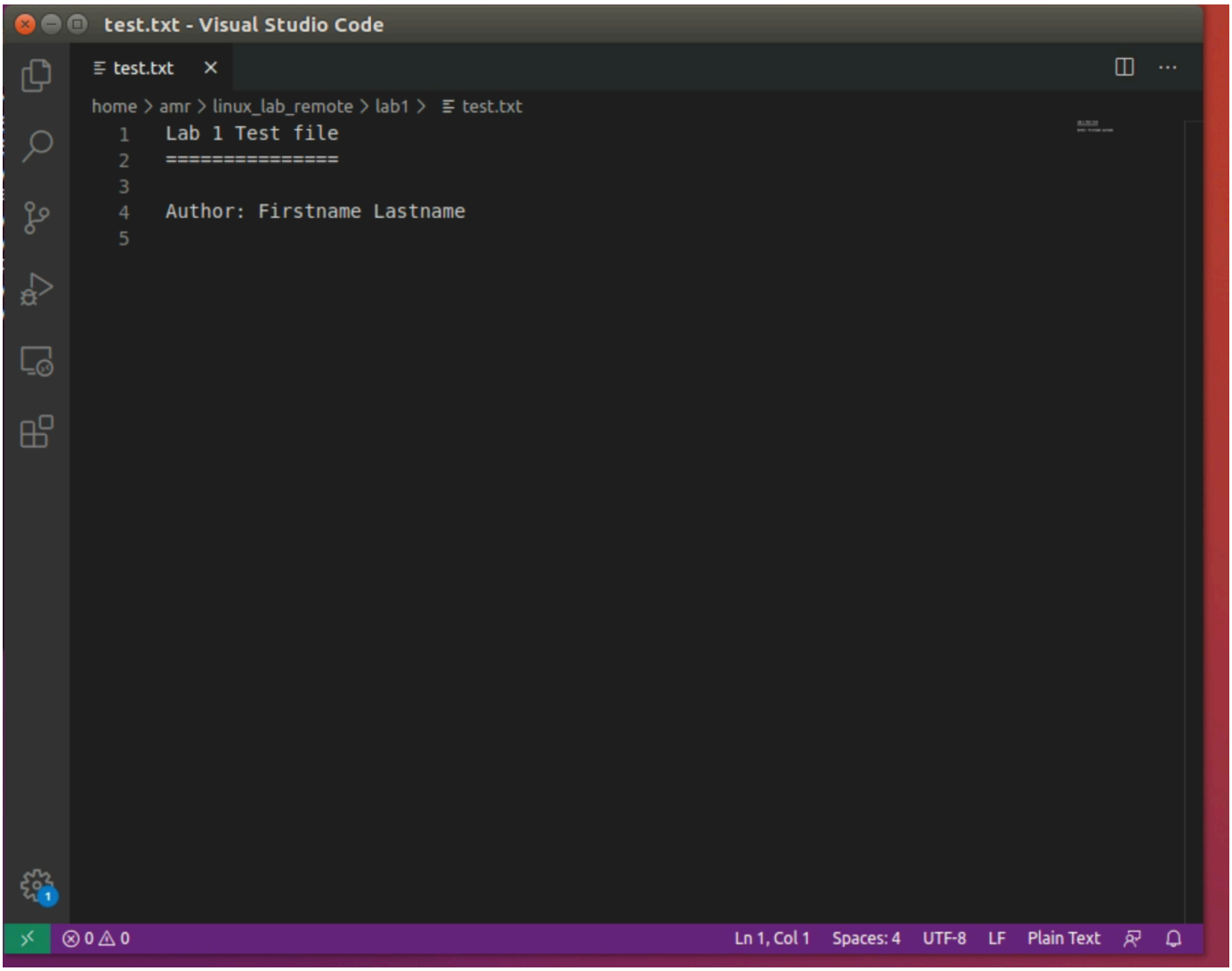
```
my_echo.py  hello_world.py  test.txt
```

How do we view and edit the contents of these files? There are many high-quality text editors for Linux. We will use Visual Studio Code (https://code.visualstudio.com), which is good for writing code.

You can open a specific file, say `test.txt`, using the `code` command from the Linux command-line by typing:

```
code test.txt
```

When you run this command, you will get a new window that looks like this:



(../../_images/ubuntu-vscode-1.png)
Specifically, you'll see the following text:

```
Lab 1 Test file
==========

Author: Firstname Lastname
```

If the file is blank, quit `code` and ensure that the file `test.txt` exists in your local directory (use `ls` to list the files in your local directory). If it does not, use `cd` to navigate to the `lab1` subdirectory inside the `lab1_linux_remote` directory.

Note: somewhat counterintuitively, the menu bar for Visual Studio Code is at the top of the Browser window. You need to run your mouse over the name to see the menu options.

For now, we will use Visual Studio Code (`code`) in a very basic way. You can navigate to a particular place in a file using the arrow keys (or your mouse) and then type typical characters and delete them as you would in a regular text editor. You can save your changes using the `save` option in the file menu or use the keyboard shortcut `Crtl-s` . To quit, you can use the file menu `quit` option or the keyboard shortcut `Ctrl-q` .

As an aside, you can also launch `code` from the application launcher: simply click the Application button (at the top left of your screen), type "code" in the input box, and then click on the Visual Studio Code icon. You can then use the `file` menu to navigate the correct file. As with the terminal application, you might want to pin the icon for launching Visual Studio Code to your launch bar (right click your mouse and choose the "Lock to Launcher" menu item.)

Exercises

Make sure that you are comfortable with this level of usage:

1. Add your name after `Author:` in this file
2. Save the file
3. Close and reopen the file in `code` and ensuring that your name is still there
4. Finally, close `code`.

Run a Python Program

`python3 file.py`	runs the python program file.py

In this class, you will learn Python. To run a Python program, use the command `python3` and the name of the file that contains your program.

Use `ls` to verify that there there is a file named `hello_world.py` in your `lab1` directory. Now, run the program in `hello_world.py` by typing (don't forget about auto-complete!):

```
python3 hello_world.py
```

This program is a very simple. It just prints "Hello, World!" to the screen.

Note

There are several variants of Python, including Python 2.7 and Python 3. We will be using Python 3 and the corresponding `python3` interpreter. The CS machines have Python 2.7 installed as the default Python. As a result, the command `python` runs a version of Python 2.7. There are some differences between the two languages and Python 3 programs may not run properly using a Python 2.7 interpreter.

Edit and Run a Python Program

In this section you will modify and rerun the program in `hello_world.py`. This change is very simple but goes through all the mechanical steps needed to program.

Open the file `hello_world.py` with the command:

```
code hello_world.py
```

The file contains a single line of code:

```
print("Hello, World!")
```

Change this line so that it instead says "Hello " and then your name. For example if your name were Gustav Larsson, the line would read:

```
print("Hello, Gustav!")
```

Do the following steps:

1. Save the file `hello_world.py` in Visual Studio Code (forgetting to save is a surprisingly common error)
2. Rerun the program using `python3`

Let's reinforce the steps to programming in Python with the terminal:

1. Change your `.py` file with an editor
2. Save the file
3. Run the file with `python3`

Forgetting to save the file (step 2) is a very common mistake!

Wild Cards (using an asterisk)

Sometimes when we enter a string, we want part of it to be variable, or a wildcard. A common task is to list all files that end with a given extension, such as `.txt`. The wildcard functionality, through an asterisk, allows to simply say:

```
$ ls *.txt
```

The wildcard can represent a string of any length consisting of any characters - including the empty string.

It is important to be **careful** using wildcard, especially for commands like `rm` which cannot be undone. A command like:

```
$ rm *             ### DO NOT RUN THIS COMMAND!
```

will delete **all** of the files in your working directory!

Exercises

1. Navigate to your `linux_lab_remote` directory. What do you see when you run `ls pa*`? What about `ls pa*/*`?
2. What do you expect to see when you run the command `ls ../pa*` from within your `linux_lab_remote/lab1` directory?

Useful Keyboard Shortcuts

Used at the Linux prompt, the keyboard shortcut `Ctrl-P` will roll back to the previous command. If you type `Ctrl-P` twice, you will roll back by two commands. If you type `Ctrl-P` too many times, you can use `Ctrl-N` to move forward. You can also use the arrow keys: up for previous (backward), down for next (forward).

Here are few more useful shortcuts:

- `Ctrl-A` will move you to the beginning of a line.
- `Ctrl-E` will move you to the end of a line.
- `Ctrl-U` will erase everything from where you are in a line back to the beginning.
- `Ctrl-K` will erase everything from where you are to the end of the line.
- `Ctrl-L` will clear the text from current terminal

Play around with these commands. Being able to scroll back to, edit, and then rerun previously used commands saves time and typing! And like auto-completion, getting in the habit of using keyboard shortcuts will reduce frustration as well save time.

Running Commands Sequentially

It is often convenient to chain together commands that you want to run in sequence. For example, recall that to print the working directory and list all of the files and directories contained inside, you would use the following commands:

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

You could also run them together, like so:

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

First, `pwd` is executed and run to completion, and then `ls` is executed and run to completion. The two examples above are thus equivalent, but the ability to run multiple commands together is a small convenience that could save you some time if there is a group of commands that you want to execute sequentially.

Note

The shell doesn't care about white space, so it will run any of the following as well:

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

Redirection

The examples in this section will use commands that we've not yet discussed. Refer to the man pages for information about unfamiliar commands.

As we already know, commands like `pwd`, `ls`, and `cat` will print output to screen by default. Sometimes, however, we may prefer to write the output of these commands to a file. In Linux, we can redirect the output of a program to a file of our choosing. This operation is done with the `>` operator.

Try the following example and compare your output with ours:

```
$ cd
$ touch test-0.txt
$ ls > test-1.txt
$ cat test-1.txt
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
test-0.txt
test-1.txt
Videos
$ echo "Hello World!" > test-2.txt
$ cat test-2.txt
Hello World!
$ cat test-2.txt > test-1.txt; cat test-1.txt
Hello World!
$ rm test-*
```

Two important things to note:

1. If you redirect to a file that does not exist, that file will be created.
2. If you redirect to a file that already exists, the contents of that file will be **overwritten**.

You can use the append operator (`>>`) to append the output of command to the end of an existing file rather than overwrite the contents of that file.

Not only can we redirect the output of a program to a file, we can also have a program receive its input from a file. This operation is done with the `<` operator. For example:

```
$ python3 my_echo.py < my-input.txt
```

(Change back to your `lab1` directory before you try this command.)

In general, all Linux processes can perform input/output operations through, at least, the keyboard and the screen. More specifically, there are three 'input/output streams': standard input (or `stdin`), standard output (or `stdout`), and standard error (or `stderr`). The code in `my_echo.py` simply reads information from `stdin` and writes it back out to `stdout` . The redirection operators change the bindings of these streams from the keyboard and/or screen to files. We'll discuss `stderr` later in the term.

Piping

In addition to the ability to direct output to and receive input from files, Linux provides a very powerful capability called piping. Piping allows one program to receive as input the output of another program, like so:

```
$ program1 | program2
```

In this example, the output of program1 is used as the input of program2. Or to put it more technically, the `stdout` of `program1` is connected to the `stdin` of `program2` .

As another more concrete example, consider the `man` command with the `-k` option that we've previously discussed. Let's assume that you hadn't yet been introduced to the `mkdir` command. How would you look for the command to create a directory? First attempts:

```
$ man -k "create directory"
create directory: nothing appropriate
$ man -k "directory"
(a bunch of mostly irrelevant output)
```

As we can see, neither of these options is particularly helpful. However, with piping, we can combine `man -k` with a powerful command line utility called `grep` (see man pages) to find what we need:

```
$ 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 extend
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 d
vgmknodes (8)        - recreate volume group directory and logical volume special fil
```

Nice.

Exercises

1. Use piping to chain together the `printenv` and `tail` commands to display the last 10 lines of output from `printenv`.
2. Replicate the above functionality without using the `|` operator. (hint: Use a temporary file.)

Man Pages

A man page (short for manual page) documents or describes topics applicable to Linux programming. These topics include Linux programs, certain programming functions, standards, and conventions, and abstract concepts.

To get the man page for a Linux command, you can type:

```
man <command name>
```

So in order to get the man page for `ls`, you would type:

```
man ls
```

This command displays a man page that gives information on the `ls` command, including a description, flags, instructions on use, and other information.

Each man page has a description. The `-k` flag for `man` allows you to search these descriptions using a keyword. For example:

```
man -k printf
```

This searches all the descriptions for the keyword `printf` and prints the names of the man pages with matches.

File Permissions

Sometimes we want to restrict who can access certain resources on the file system.

Most file systems assign 'File Permissions' (or just permissions) to specific users and groups of users. Unix is no different. File permissions dictate who can read (view), write (create/edit), and execute (run) files on a file system.

All directories and files are owned by a user. Each user can be a member of one or more groups. To see your groups, enter the command `groups` into the command line.

File permissions in Unix systems are managed in three distinct scopes. Each scope has a distinct set of permissions.

User - The owner of a file or directory makes up the *user* scope.

Group - Each file and directory has a group assigned to it. The members of this group make up the *group* scope.

Others - Every user who does not fall into the previous two scopes make up the *others* scope.

If a user falls into more than one of these scopes, their effective permissions are determined based on the first scope the user falls within in the order of user, group, and others.

Each scope has three specific permissions for each file or directory:

read - The read permission allows a user to view a file's contents. When set for a directory, this permission allows a user to view the names of files in the directory, but no further information about the files in the directory. `r` is shorthand for read permissions.

write - The write permission allows a user to modify the contents of a file. When set for a directory, this permission allows a user to create, delete, or rename files. `w` is shorthand for write permissions.

execute - The execute permission allows a user to execute a file (or program) using the operating system. When set for a directory, this permission allows a user to access file contents and other information about files within the directory (given that the user has the proper permissions to access the file). The execute permission does not allow the user to list the files inside the directory unless the read permission is also set. `x` is shorthand for execute permissions.

To list information about a file, including its permissions, type:

```
ls -l <filepath>
```

You'll get output of the form:

```
<permissions> 1 owner group <size in bytes> <date modified> <filepath>
```

For example, if we want information on `/usr/bin/python3.5`:

```
$ ls -l /usr/bin/python3.5
-rwxr-xr-x 1 root root 4460272 Aug 20 /usr/bin/python3.5
```

First thing we can notice is that the owner of the file is a user named `root`. (FYI, `root` is a name for an account that has access to *all* commands and files on a Linux system. Other accounts may also have "root" privileges.) The file's group is also `root`.

The permissions are `-rwxr-xr-x`. The initial dash (`-`) indicates that `/usr/bin/python3.5` is a file, not a directory. Directories have a `d` instead of a dash. Then the permissions are listed in user, group, and others order. In this example, the owner, `root`, can read (`r`), write (`w`), and execute (`x`) the file. Users in the `root` group and all other users can read and execute the files.

Exercises

By default, any files or directories that you create will have your username as both the user and the group. (If you run `groups`, you'll notice that there is a group with the same name as your username. You are the only member of this group.) On our Linux machines, by default, new files are give read and write permissions to user and group and no permissions to other. New directories will be set to have read, write and execute permissions for user and group.

1. Verify this claim by running `ls -l backups/copy2.txt` and `ls -ld  backups` in your `lab1` directory.

The `-d` flag tells `ls` to list the directory, instead of its contents. Notice that that the first letter in the permissions string for `backups` is a *d*, while it is a `-` for `backups/copy2.txt`.

Once you have verified the claim, go ahead and remove the `backups` directory using the command: `rm -r backups`.

Changing Permissions, Owner, & Group

`chmod` <permissions> <path-name>	set the permissions for a file/directory
`chmod` <changes> <path-name>	update the permissions for a file/directory
`chown` <username> <path-name>	change the owner of a file to username
`chgrp` <group> <path-name>	change the group of a file
`cat` <path-name>	print the contents of a file to the terminal

To change permissions, we use the `chmod` command. There are two ways to specify the permissions. We'll describe the more accessible one first: to set the permissions you specify the scope using a combination of `u`, `g`, and `o`, the permission using `r`, `w`, and `x`, and either `+` or `-` to indicate that you want to add or remove a permission. For example `uo+rw` indicates that you want to add read and write permissions for the user and others groups.

We can demonstrate this using the `cat` command to print file contents to the terminal:

```
$ echo "Hello!" > testfile
$ ls -l testfile
-rw-rw---- 1 username username 7 Aug 23 11:22 testfile
$ cat testfile
Hello!
$ chmod ug-r testfile #remove read and permissions from user and group
$ ls -l testfile
--w--w---- 1 username username 7 Aug 23 11:22 testfile
$ cat testfile
cat: testfile: Permission denied
$ chmod u+r testfile #give user scope read permissions
```

In this last example, we have added user read permissions to `testfile`.

In addition to the symbolic method for setting permissions, you can also use a numeric method: each permission has a unique value: read = 4, write = 2, execute = 1. As a result, you can describe the permissions of each scope using the sum of its permissions' values. For example, if a file has read and write permissions for the user scope, its permissions can be described as 6 (4 + 2 = 6).

You can describe the permissions of a file overall using these values for each scope. For example, 761 describes the permissions for a file with read, write, and execute permissions for the user scope, read and write permissions for the group scope, and only execute permissions for the others scope.

The symbolic approach is relative: it allows you to add and remove permissions relative the the current file permissions. The numeric method is absolute: it sets the permissions to a specific configuration. We recommend starting the symbolic approach. It is easier to get right. As you get more comfortable with setting permissions, it is useful to learn how to use the numeric method.

To change the owner of a file or directory (if you are the owner or root), use the command:

```
chown <new owner> <path to file>
```

To change a file's group (if you are the owner or root), use the command:

```
chgrp <new group> <path to file>
```

It is unlikely that you will need to use these two commands for this course.

Exercises

1. Run `echo "Hello!" > testfile` to construct `testfile`. Look at the permissions using `ls -l`.
2. Change the permissions on `testfile` to allow and read access for others. Run `ls -l testfile` to check the new permissions.
3. Remove group write access from `testfile`. Check the corrected permissions.
4. Remove `testfile` using `rm`.

Final Notes

Sometimes, a program will run indefinitely or misbehave. When this happens, you can type `Ctrl-C` to send an interrupt signal to the running program, which usually causes it to terminate. On occasion, you may need to type `Ctrl-C` a few times. Typing `Ctrl-D` sends an end of input signal, which tells the program that no more information is coming.