Using Linux (Unix / MacOs) at the command line.

Where are you

All the stuff at the command line is relative to your current path. You can find out where you are in the system with the `pwd` command.

From my 'host' system when I login to my remote system.

```
$ ssh pschlump@192.168.0.36
...
$
```

I can type `pwd` return and find my current location.

```
$ pwd
/home/pschlump
```

Using Ubuntu the (and most Linux systems) the home directories for users are under /home. On a Mac (BSD Linux) system they are under /Users.

Bring up a terminal window and give this a try.

File paths and files

Unix uses the `/` character as the file path separator. This is the only character that you can not use in a file name. I don't use non-printing characters like blanks or tabs in file names because they are incredibly irritating and break things. Also I don't start file names with a minus `-` . That will also cause problems.

Each user has an account that is usually (but not always) in the /home directory. The leading '/' indicates that the path starts and the top of the tree and works down. If I start a path without the '/' at the beginning then that path is relative to where I currently am in the directory tree.

Let's create a file to test with and put it in our home directory. (Assuming that you are already logged into your Linux system - either using the console and a terminal window or using SSH from above)

```
$ ls -a >example.1
```

There is nothing special about the `.1` as a file extension. File extensions are just a part of the file name. Let's create a second file with a different extension.

```
$ ls -a >bob.this.is.a.file
```

I think the file extension is `.file` on this one. You can make the argument that the extension is `.this.is.a.file` or that the file has no extension.

There are 2 kinds of files - text and binary. Source code is text. You can edit it with a text editor. Programs on Windows like Word tend to create binary files. Most images are binary files. Structured Vector Graphics images are text.

As far as the system is concerned a file is a set of bytes of data and it makes no difference between text and binary. So if you use a text editor to edit a binary file you will just see a lot of unusual garbage on the screen.

In Unix and Linux a directory is just a special kind of binary file that the operating system knows about.

Let's go and see our 2 files.

```
ls
```

Lists out the set of files.

```
ls -l
```

Makes for a longer list with details on the files.

Let's move up one in the directory tree and we will do a couple of ls commands to poke around.

```
$ cd ..
$ pwd
$ ls
```

Now let's look at what is in the pschlump directory.

```
$ ls pschlump
```

Now let's take a look from the top.

```
$ cd /
$ ls home
$ ls home/pschlump
$ ls home/pschlump/example.1
$ ls -l home/pschlump/example.1
```

Now back to our home directory.

```
$ cd
```

or

```
$ cd /home/pschlump
```

or

```
$ cd ~
```

That last one is cd ~ tilde. In file paths a leading ~ referees to your home directory.

File permissions

In the land of Unix/Linux/MacOS all files have 3 sets of permissions. Usually by default when you create a file you are the only person that can modify the file. You own it. This applies to directories and to you home directory. There is a special account called 'root' that has unlimited permissions and it can access or modify any file or directory. When you 'sudo [command] what you are doing is becoming the root account for this single command. When you install software like PostgreSQL on your system this is required because you chagne files that you do not own. These files belong to the root account. root also has the ability to create new users. One of the parts of the postgres install is creating a postgre` user that the database runs under.

Remember that directories are just special binary files - so they have permissions also. Directoris are a little different - as you will see.

First let's go and look at the permissions of our 2 files.

```
$ cd
$ ls -l
```

The first column has a `-` in it. If it were a directory it would have a `d`. Let's make a directory.

```
$ mkdir mydir
$ ls -l
```

This is a long listing on `ls`. Normally options are with a `-` as the first character and some letters telling you what the options are. In this case the `l` is for long.

Now you have a `file` that is a directory and it has a `d` in as the first letter. Unix has other special files that have other characters at the beginning - but we won't need to touch on that subject too much. If you see a 'l' at the beginning this is a symbolic link.

The first line is the total size in file-system blocks of the set of files.

```
Total 12
```

Your number will be differnt than mine because I am running a different kind of file system.

The column data is:

```
total 12
-rw-rw-r-- 1 pschlump pschlump   34 Mar 13 10:32 bob.this.is.a.file
-rw-rw-r-- 1 pschlump pschlump   15 Mar 13 10:32 example.1
drwxrwxr-x 2 pschlump pschlump 4096 Mar 13 10:32 mydir
```

The 2nd column is usually a 1 - this is how many different file names are associated with the data for this file. Notice that the 1 is a 2 for the directory. We will get to that in a second. Then there are 2 columns. These are the user and the group that the person is in. In my case they are `pschlump` and `pschlump`. The next column is the size in bytes of the file. Then there are a set of 3 coluns that are the last time the file was modified. The last column is the file name.

Back to that 1st column - right after the `-` or `d`. This is important. There are 3 sets of 3 characters. The 3 sets are the owner, the group and others. These are the permissions that the file has between the owner of the file, the group that the owner is using right now and anybody else on the system.

Each set is a set of 3 on/off flags. The first one is `r` or a `-`. When the flag is on you have a `r` when it is off you have a `-`. `r` is for the readability of the file.

The 2nd in the set is the write-permission of the file. This is a `w` or a `-`.

The 3rd in the set is if the file is executable. When you compile a file the resulting binary file is executable. We can also build scritps that will use some tool to run them. In this class we will build

Python scritps and make them executable so that python can run the file. Also note that when you enter a command you are using the `bash` scripting tool - it is the default Linux shell and you can write and run scripts in `bash` also.

The only file that we have that is executable is the directory. It has `x` in all of its owner,group,other sets of permissions. The `x` on directory has a special meaning. This means that tools that want to read the directory can read it. One of those tools is `ls` that lists the contents of a directory.

There are some special files in each directory. This is the cause of our `2` in the 2nd column. These are by default hidden files. Let's take a look at what they are:

```
$ ls -la
```

Will show you all of the hidden files. That is `a` for all the `-la` option to ls.

```
total 20
drwxrwxr-x 3 pschlump pschlump 4096 Mar 13 10:32 .
drwxrwxr-x 3 pschlump pschlump 4096 Mar 13 10:32 ..
-rw-rw-r-- 1 pschlump pschlump   34 Mar 13 10:32 bob.this.is.a.file
-rw-rw-r-- 1 pschlump pschlump   15 Mar 13 10:32 example.1
drwxrwxr-x 2 pschlump pschlump 4096 Mar 13 10:32 mydir
```

You will notice 2 additional files that are directories. One of these is `.` . That is the current directory (Yes the directory referees to itself). The other is `..` . That is the parent of the current directory. Each directory contains these 2 links. So if you want to go up 1 in the tree you can do

```
$ ls -l ..
```

or go up 2

```
$ ls -l ../..
```

You can only go to the top. Try

```
$ ls -l ../../../..
```

You can use the `./file` to refer to files in the current directory or the `../file` to refer to files in a parent directory. Try `ls -l ../pschlump` where you use your home directory name.

You can only write to files that you have permissions to write to. You can only create files that you have write-permissions and execute-permission on the directory that you are writing to.

This is important because PostgreSQL is running in it's own account. So if you login to your account, then switch to the `postgres` account you can not create or modify files in your original login account.

There is a common, any body can write to it temporary path on Linux. This is /tmp and both accounts can read/write/modify files in it. Remember that if you are sharing a system with other people and you put a file in /tmp it is public. Don't use it for stuff that you need to keep secure.

We can connect to a different account using the 'su' or substitute user command. When we want to be the `postgres` account we can (You will get a password prompt just like loing - you don't know this password so just enter Contro-C (^C)):

```
$ su –l postgres
Password:
```

It is actually important that the postgres account is missing a password. Nobody should be able to login to this account directly. This is an important security feature.

To access this account first we need to become `root` - the account that has all permissions, then becom e postgres. The `sudo` command runs stuff as root - we will get prompted for our login password, then we can become the `postgres` account.

```
$ sudo su –l postgres
```

After this I am the postgres user and I can do a `ls` and a `pwd` .

```
postgres@wabl001:~$ ls
12  13   14
postgres@wabl001:~$ pwd
/var/lib/postgresql
```

A couple of thins to notice. I have both version 12, 13 and version 14 of postgres installed. The 2nd thin is that the home direcory for the `postgres` account is `/var/lib/postgresql` not a directory under `/home` .

The postgresql account has special privilages when you connect to postgres that allows for the creation of users in the databse. Database users are not the same as system users.

When you want to exit the postgres account use a Control-D to send an end of file to the bash shell. This will return you to your accont.

Accessing PostgreSQL from the `postgres` account.

We can use `psql` to access the postgres database from the postgres account. From your account - first become the Linux postgres account.

```
$ sudo su -l postgres
postgres@wabl001:~$ ls
12   13    14
```

Now use `psql`

```
postgres@wabl001:~$ psql
postgres=#
```

You are now connected to a special privileged account inside the database. You should not create tables or build stuff in this account. From this point you should create an account to work in.

Replace `newuser` with your login name on the Linux system and `mypass` with a password that you like.

```
postgres=# create database newuser;
postgres=# create user newuser with encrypted password 'mypass';
postgres=# grant all privileges on database newuser to newuser;
postgres=# \q
postgres@wabl001:~$
```

Now Control-D (^D) to exit the PostgreSQL user.

If you try to use the Linux `postgres` account this means that you have to deal with file permissions and copy from account to account. It is much easier to just setup an account that you can use directly from your login account.

Accessing PostgreSQL from your Linux account.

From your account you should now be able to use `psql`.

```
$ psql
pschlump=#
```

By setting up PostgreSQL to use your Linux account with a matching database account with the same name you can now have access to your files.

Let's say that you create a table. Use a text editor like vi to create a file called bob.sql.

```
$ vi bob.sql
```

With

```
create table bob (
  nnn int
);
```

in it.

now run it in `psql` .

```
 psql
Pager usage is off.
psql (13.2 (Ubuntu 13.2-1.pgdg18.04+1), server 12.6 (Ubuntu 12.6-1.pgdg18.04+1))
Type "help" for help.

pschlump=# \i bob.sql
CREATE TABLE
pschlump=# \d bob
                Table "public.bob"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 nnn    | integer |           |          |

pschlump=# insert into bob ( nnn ) values ( 1 );
INSERT 0 1
pschlump=# \q
```

Changing permissions on a file

Someimes you need to chagne permissions. We will need this to run our server in Assignment 04.

Let's create a tiny bash script file

```
$ cat >tiny.sh
#!/bin/bash
echo Tiny
^D
$ ls -l tiny.sh
```

To make a file executable (a script file):

```
$ chmod +x tiny.sh
```

Then run it

```
$ ./tiny.sh
```

By ptting the path to python at the top we can create an excutable python script.

```
$ cat >sample.py
print ( "Hi From Python" )
^D
$ chmod +x sample.py
```

And run it

```
$ ./sample.py
```

References

1. Su command https://phoenixnap.com/kb/su-command-linux-examples