Lecture 17 - An example Fetch Execute Machine

2 billion PCs with GUI interfaces.

3.5 billion smart phones (requiring embedded programming)

31 billion IoT devices. 82% run Linux (Linux or RTOS Linux). 73% have a command line.

“Pull back the curtain, and let the world see…”

This is an interpreter that we are all going to use some in the class. This is the real beginning of us looking at “system” software.

The Command Line. This seems to be the biggest stumbling block for the homework.

How it Works.

You get a prompt. On a Mac or Linux using either `bash` or `zsh`

```
$
```

if you see something with a this indicates that you have to use the `root` account. That is not required for anything in this class.

```
#
```

On Windows (Power Shell)

```
C:\>
```

It reads in a line - this is the “fetch” part.

Then it parses the line - blanks are used as delimiters. This is why blanks in file names are bad. Some tools won’t even work with blanks in file names. Examples: The 3d Rendering tool Blender, the Ethereum stack of tools, some compilers like Go from version 1.0 to 1.8.

From this point on I am going to show how it works on a Mac - because the process is basically the same except for some noted details on Windows. You can change your prompt. Mine is usually a `+=>` on this system. I will set it to a `$` at this point.

Let’s take a look at a file:

```
+=> PROMPT="$ "
$
```

Let's take a look at a file.

```
$ cat abc.txt
```

and another file xyz.txt

```
$ cat xyz.txt
```

now both of them:

```
$ cat abc.txt xyz.txt
```

This is what `cat` derives its name from - you can concatenate files with it.

When you want the output from a file you can pipe it into a new file.

```
$ cat abc.txt xyz.txt >both
```

Now:

```
$ cat both
```

The `>` is the pipe to a file. There are also pipes between programs and for input from a file. we will get back to pipes in a bit. First - how to find the documentation on the commands:

```
$ man cat
```

On windows just use google. Google "man path".

You can add line numbers!

```
$ cat –n both
```

or (I switched the order)

```
$ cat –n xyz.txt abc.txt
```

So the shell picks apart the command into a set of terms. An array of `[ "cat", "–n", "xyx.txt", "abc.txt" ]`. Then it says - is this a command that is built in. It has a bunch of them. `cat` is not built in. So it uses a thing called the `PATH` variable to lookup the command. The path has a set of directories where it can find executables and it searches it in order. We can see where (on a Linux or Mac it finds "cat") with the `which` command.

```
$ which cat
```

It is in `/bin/cat` - this is the directory /bin and the file is the executable `cat`.

It looks at the file and determines that it is a program to load. On a windows system the PATH is system wide. On Mac/Unix/Linux it is per-shell.

The shell calls the program loader to load the executable into memory. The loader will actually make a copy of the executable on disk - and the the shell will "fork" (on windows it runs a new process). The "fork" is where the parent process divides into 2 process - a parent and a child. The child is then replaced with the newly loaded "cat" command code. The arguments are passed to the child. So the child gets the array `[ "cat", "–n", "xyx.txt", "abc.txt" ]` and gets to decide what to do with it.

So in our homework when you write a program and it looks at the command line arguments for `––in` and a file name - this is what it is looking at.

Fork has all sorts of interesting computer architecture implications. It was designed in a world where there was really only 1 CPU and a small amount of memory - so the system could effectively appear to run more than one program and context-switch between them. Unix was build on a PDP-11 with 128k of memory. I was a system admin on a similar power system running System V Unix - with 28 users. It worked.

So... What will "cat" do.

1. It will parse the command line arguments for things that start with a `–` symbol like `–n`.
2. It will take each file and open it, read it and copy it out to its output.

Ok. We have "cat". What other commands. Let's try 'ls' - it is a directory listing.

```
$ ls
```

output:

```
Lect-17.html Lect-17.html.pdf Lect-17.md Makefile Wizard-Oz1.jpg
abc.txt      xyz.txt
```

`ls` lists the files in a directory. We can list in other directories also. For example the `/bin` directory where `cat` came from.

```
$ ls /bin
```

On windows try

```
C:\> ls c:/windows/system32
```

on this system:

```
[        bash      cat     chmod     cp          csh     date    dd
df       echo      ed      expr      hostname    kill    ksh     launchctl
link     ln        ls      mkdir     mv          pax     ps      pwd
rm       rmdir     sh      sleep     stty        sync    tcsh    test
unlink   wait      path    zsh
```

So lots of commands that we can look into. Note that `ls` is in this list.

Command	One liner on it
df	show free space on disks.
rm	remove (delete) a file.
bash	the shell itself.
echo	print out what is on the command line (generate output).
rmdir	delete a directory.
ed	a very old - but useful editor.
chmod	change permissions on a file.
cp	copy one file to another.
mv	rename a file or move it to a new path.
date	get the current date.
pwd	tell where you are in the directory tree.

There are more than 1 shell. `sh`, `bash`, `ksh`, `csh`, `zsh`, `tcsh` are all shells.

There are some weird commands like `sleep` and `[`.

Most systems use more than one directory in the PATH and you can set the PATH.

```
echo $PATH
```

or in power shell.

```
C:\> echo $Env:Path
```

One of the builtin commands to the shell is the `cd` command. The shell has the concept of a current working directory. This is what PWD prints out. You can change the directory with the `cd` command.

```
$ pwd
```

In my case will be:

```
/Users/pschlump/go/src/github.com/Univ-Wyo-Education/S20-2150/Lectures/Lect-17
```

We can go up 1 directory with:

```
cd ..
```

Now `pwd` will output

```
/Users/pschlump/go/src/github.com/Univ-Wyo-Education/S20-2150/Lectures
```

and `ls` is:

```
Lect-01 Lect-02   Lect-03 Lect-04 Lect-05 Lect-06 Lect-07 Lect-08
Lect-09 Lect-10   Lect-11 Lect-12 Lect-13 Lect-14 Lect-15 Lect-16
Lect-17 py-cli.py todo.1
```

We can go back down the tree with

```
$ cd Lect-17
```

So when you double click on a program what is really happening is that it is running it as the “shell” and starts it with some sort of a command line. In the workd of “mac” it is the “open” command.

For example I can “open” a .jpg file

```
$ open Wizard-Oz1.jpg
```

We can also see where the “open” command is.

```
$ which open
```

What will “open” do?

Copyright