The Shell

Operating Systems

- Ubuntu is an example of a Linux distro (distribution).
- Debian is the upstream distro for Ubuntu from which it inherits thousands of *packages*. You can can check the packages with `dpkg -l`.
- Linux is an OS **kernel**.

Software levels of abstraction:

```
+------------------------+
|          apps          |
+------------+-----------+
|            |   libs    |
|   kernel   +-----------+
|            | C stdlib  |
+------------+-----------+
|        hardware        |
+------------------------+
```

sh and Emacs and any of their independent instances are themselves **applications**, one of many that sit atop the operating system.

Introspection: When a program looks at itself ("when we use tools to find out more about our tools"). Knowing how to perform introspection is a portable, universal skill that lets you explore or relearn something about an unfamiliar program.

Superusers and the Concept of Privilege

- **Superusers** ("root") are the only ones with permission to `kill` PID 1, `system`.
- The **sudo** command lets you run a command AS "root":

```
sudo sh
```

Why have multiple users instead of just root?

The concept of **minimization of privileges** aka "principle of least privilege". If a program breaks or a user makes a mistake, the damage is limited.

By the way, you can kill a program by PID with:

```
kill PID1 PID2 ...
```

The (ba)sh Program

A shell is itself a program, and programs themselves are just files that can be executed by the operating system.

sh is the predecessor to **bash** (Bourne Again SH). sh was designed to work on 16-bit machines so it's a very little language. Bash adds some features in addition ot the original sh.

There is also a lot of other shell languages ending in sh. Having so many distinct shell languages becomes a problem, so the **POSIX standard** was created as a spec for shells.

Creating another instance of the shell from within the shell itself to execute a one-off command:

```
sh -c <command>
```

This is what you call a **subprocess**, or a **child process**. Within your running instance of `sh` (the CLI you're typing into), another *instance* of `sh` is spawned as a child of that `sh`. In fact, every time you run a command, an instance of their little program is attached to your shell as a child process. You can see this for yourself when using commands like `ps` to show processes and their parentage.

ASIDE: Some Mischievous Things

Things you can do to annoy your system administrator:

Telling the shell to go into an infinite loop:

```
sh -c 'while true; do true; done'
```

A no-op for a number of seconds, doesn't use CPU

```
sleep 10
```

The `truncate` command sets a certain file to a certain size, ending at the size you specify (filling with null data if the size expands I presume):

```
truncate --size=10TB bigfile
```

If you inspect the filesystem, you'll find that you didn't actually use up that much space, just convince the directory listings that that much space had been allocated for *something*. This is still annoying though because it will trip up the sysadmins when they perform **backups**.

Shell Commands

Basic Output Manipulation

Some basic commands and their most common arguments you should integrate into your workflow (and ones that will be used without warning from here on out). These are typically used as something to pipe output *into* to improve your experience:

- `wc [-cmlw]`: Output statistics about the number of bytes (`-c`), characters (`-m`), lines (`-l`), and/or words (`-w`) of the input stream or file(s).
- `head -n N`: Output only the first `N` lines of the input text.
- `tail -n N`: Output only the last `N` lines of the input text.
- `more`: Paging utility that lets you scroll through text a screenful at a time instead of having it be outputted all at once to the console. Like a read-only editor, it also supports interactive commands reminiscent of Vi keybindings and features like regex search.
- `less`: The direct upgrade to `more`, it also supports backwards navigation and more inclusive support for keybindings.

Command Arguments

Within shell languages, the simplest commands look like:

```
word0, word1, ..., wordn
```

`word0` is the name of the program and `word1, ..., wordn` are the arguments to that program. Arguments typically follow conventions:

- Normal (**positional**) arguments, like `a`.
- **Options** (aka **flags**) `-ejh` which is also equivalent to `-e -j -h`. Jamming them together is a thing you can do on Unix, but as we know from Assignment 2, this behavior becomes tricky when there are options that take arguments.
- Options that *don't* take any arguments are sometimes referred to as **switches**, and they set some kind of configuration simply by being present or absent (think of it like a boolean option, where presence means true and absence means false).

Command Substitution

The **command substitution** `$()` syntax allows you to write a shell command *as if you wrote the output* of the sub-expression word for word out as the arguments to the outer command.

Note that this can have significant edge cases like in this example from lecture:

```
mkdir empty
cd empty
wc $(find . -type f)
```

The program then looks like it's jammed because if the sub-expression `$(find . -type f)` returns nothing, it's *as if* you just typed the command:

```
wc
```

`wc` has defined behavior where if you did not provide any command line arguments, then it reads from stdin. Thus, the program is actually prompting the user to input something to the `wc` program.

Only caring about that last summary line, you can use the `tail` command:

```
wc $(find . -type f) | tail -n 1
```

Running Multiple Commands

Commands in Sequence

`;` is the **sequencing operator**, equivalent to a newline but allows you to automatically queue subsequent commands in one line if you're at the command line:

```
ls -d .; echo a
```

This forms the basis of shell **scripting** - a sequence of commands to automate certain tasks.

> A big part of scripting is knowing when you don't have to care about efficiency. "Your time is more important than the computer's time." This is a different paradigm then coding with algorithms and asymptotic time, etc. in mind.

Commands in Parallel

```
ls -d . & echo a
```

Command Pipeline

Piping output of one command as the input to another. The **pipe** is itself a buffer. Commands that write can write to it, commands that read can read from it, so you can set up a sequence of writing and reading commands to **pass content between them**:

```
ls -d . | less
```

Instead of `ls -d` running until completion and *then* running `less`, *every time* `ls -d` outputs something to the buffer, `less` can read from it and execute its program.

This is **massively useful** because it allows you to set up a sequence of little languages that each process or transform a single stream of text output as it makes its way into some final form.

Example from discussion:

1. Listing all files ended with `.html` under ~/...
2. ...sorting it...
3. ...and then outputting it into a file named `list_html.txt`:

```
ls ~/*html | sort > list_html.txt
```

The `ps` Program

- Like the task manager: lists processes and their information
- For a live view, use the `top` program instead
- You can also use `less /proc/cpuinfo` to view information about the currently running processor
- By default lists the processes on the local machine, but you can use some useful flags like:

```
ps -ejH | less
```

You can pipe the output into the grep command to filter it by a keyword, as if searching for a process by name:

```
ps -ef | grep emacs
```

PATH and Resolving Command Names

If the name of a program does not contain any slashes, then the shell automatically looks through directories listed in the `PATH` **environment variable** to find the program name. This is why when you want to run an executable in the current directory, you have to use the `./executable` syntax:

```
$ a.out
bash: a.out: command not found
```

```
$ ./a.out
Hello world
```

Alternatively you can *append* the current directory to the PATH:

```
PATH=$PATH:.
```

Now it would work because `a.out` is now treated like a command that's on your `PATH`:

```
$ a.out
Hello world
```

Had to say it one last time for y'all:

> For the last time, make sure `/usr/local/cs/bin` is prepended to your `PATH`!

Y'all better understand why that's so important now. Cheers.

> You can type the *null byte* with `^@`! Alternatively, you can use `printf '\0'` to use familiar C code and pipe that into whatever you want.

Quoting

The shell itself has special characters, so you must use **quoting** to input what you intend to. A common use case is quoting a file name that has a space in it so the shell does not treat the space as a delimiter for arguments.

Special characters include *whitespace* as well as:

```
` ~ # $ & * ( ) = [ ] \ | ; ' " < > ?
```

There are two forms of quoting: single (`'`) and double (`"`).

Single Quoting

Using single quotes makes the shell treat everything inside the quotes as a *single word*. More importantly, *every* character inside is preserved *literally*. This means that you cannot include the single quote `'` itself because escaping is not possible.

```
$ echo 'hello\there\n"general kenobi"'
hello\there\n"general kenobi"
```

Double Quoting

Alternatively you can use *double quotes* to enclose "most any characters". `$` is still special, used to **interpolate** other variables or sub-expressions. Special characters are also possible with **escape sequences**, like `\"` to represent a double quote mark itself. Interestingly, `\n` does not work the way you would expect.

```
$ echo "hey there's\n\"general kenobi\""
hey there's\n"general kenobi"
```

Quoting can even include newlines itself (RET at the command line), but that's so controversial that it's probably going to be removed in an upcoming POSIX standard release.

Globbing (Wildcard) Pattern Matchinig

Used in file management: `man 7 glob`

The shell will **expand** strings containing these special characters to every string that matches the pattern, separated by whitespaces. For example, `*` in a directory containing files named `file1`, `file2`, and `file3` would expand to:

```
$ echo *
file1 file2 file3
```

This is done *by the shell* and NOT the programs typically associated with them like `ls`. When you run something like `ls hello/*`, bash first expands the string `hello/*` to `hello/some_file hello/another_file ...` and *then* passes it to `ls`.

- `?` - match one, any character
- `*` - match any number of characters, including the empty string
- `{pdf,jpeg}` - match multiple literals
- `[]` - character set that supports *ranges*, similar to regex
- `[!]` - complement of a character set if `!` is the first character
- `\` - escape character

Shell Scripting Constructs

Looping

```
for i in {1..100}
do
    echo $i
done
```

Conditionals

```
if [ condition ]
then
    # body
fi
```

You can use the `expr` command to evaluate an expression. You can also use **compound expansion** with `((expression))` syntax.