Little Languages

Lecture 03:

On the

Highway to Shell

In-class Quiz Friday: vim 101

- Know the **Verbs** and **Locations**, for example:

- Know how to form three kinds of **commands**:
  1. **cursor_to**
  2. **operation,** and
  3. **line_operation**

- **Example questions:**
  - Question: What terminal key represents the ***location*** of the ***end of the file***? Answer: G
  - Question: What command string would **delete** an **entire line**? Answer: **dd**

What is a **shell**?

- The core of an operating system is the ***kernel***. Its job is to "fairly" share a machine's resources between running processes (programs).
  - Ensure programs make progress by sharing time on the CPU.
  - Ensure running programs have memory isolated from other programs.
  - Broker access to devices like file system storage, network adapters, and so on.

- A ***shell*** is a wrapper around the kernel that enables users to more easily employ an operating system's services

  - Originally, all shells were command-line interfaces (CLI) like the one you're using in the VM.
    - Glenda Schroeder invented the Multics Shell in 1965 at MIT
    - Ken Thompson's original Unix (1971) shell was modelled after Multics'
    - Stephen Bourne's iteration sh (1977) most closely resembles the shells we use today
    - Your default shell in this class both on your host machine and VM is **bash** (1989) which is short for "Bourne Again Shell"

  - Since Xerox Parc invented the modern graphical user interface (GUI) in 1973 and Apple/Microsoft commoditized them in the 1980s, most consumers experience GUI shells.

Glenda Schroeder
invented the first command-line user interface shell while at MIT in 1965.

Photo Source: tech-heroines.org

An important point to understand about a shell... *it's **just** a program.*

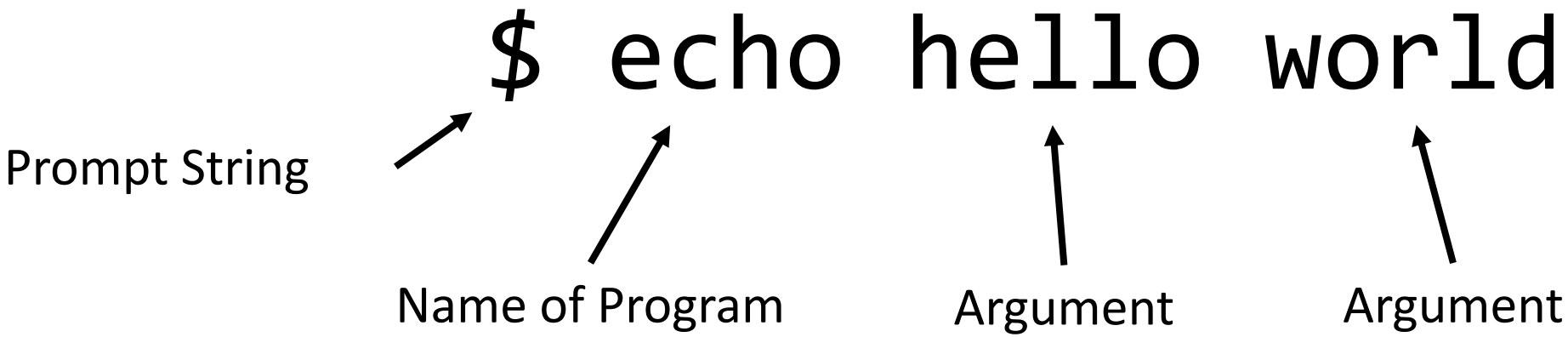
- Your keystrokes are fed into its standard input just like programs that read from **`System.in`** in Java

- You can write your own shell!
  - The current plan for the finale problem set this semester is to implement a simple shell.

- When you enter a command the shell takes your input, parses it, and carries out the command(s)... *the shell has its own little language.*
  - Typically your command starts processes by telling the operating system to execute a program file completely separate from the shell (i.e. echo, ls, cat, git, cargo, pandoc, ...)
  - A few commands are built directly into the shell (i.e. cd - change directory)

A simple shell command, intuitively…

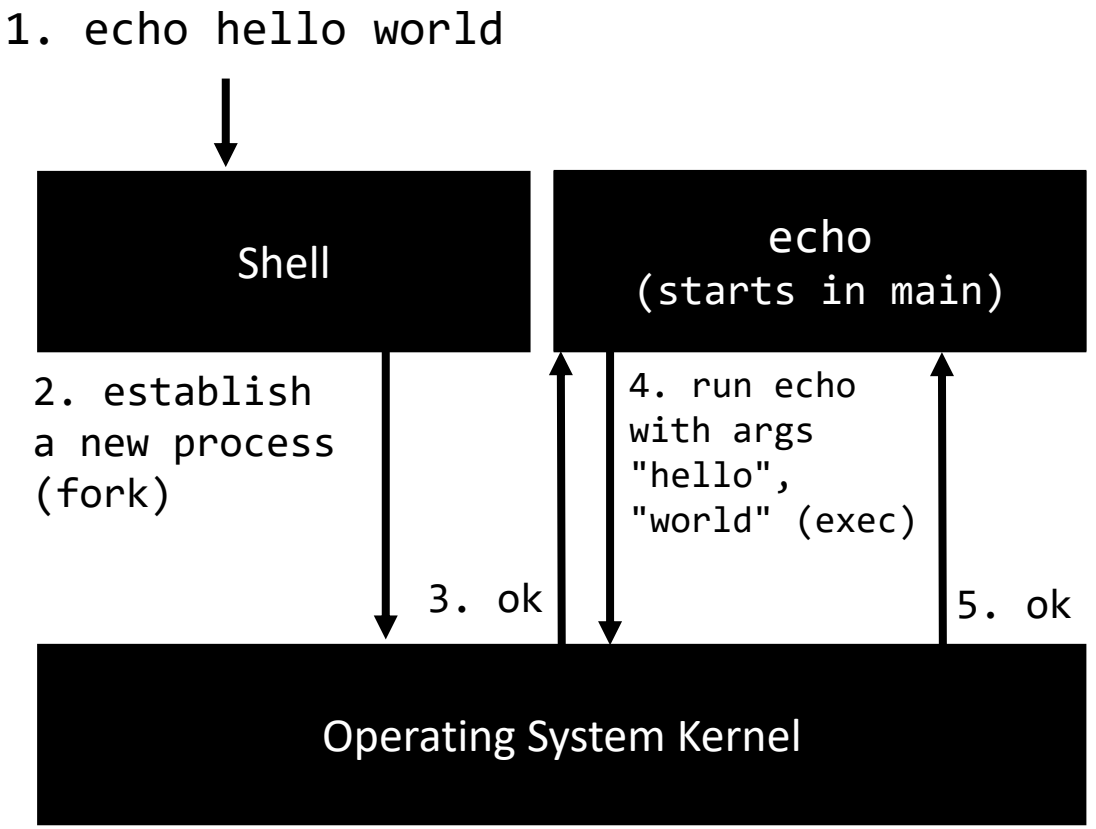
- The first word you give in a shell command is the *name* of a program file
- Subsequent words you give are *arguments* given as inputs to the program to modify its behavior



What happens when you issue a shell command?

- The shell parses your input based on its grammar (1)
- Ultimately it makes a *system call* to the operating system asking it to start a new process (2) the operating system establishes the new process (3)
- That process then asks the operating system to load and execute the echo program it with the given arguments (4). The operating system does so and passes the program the given arguments (5).
- Each programming language has its own convention for how you access the arguments it is given when it starts. For example, in Java:

```
public static void main(String [] args) {
```



Shell Grammar*

```
command      -> run_program

run_program  -> program | program arg_list

program      -> PROGRAM_NAME

arg_list     -> arg | arg SPACE arg_list

arg          -> WORD | QUOTED_WORDS
```

Hands-on: draw a parse tree for the command below.

```
git commit -m 'first commit'
```

* A grammar that's good enough for now. This is a *very* hand waving and much simplified shell grammar.

Accessing command-line arguments in Rust

- To access a program's command-line arguments in Rust, call:

`std::env::args()`

- This function returns a `std::env::Args` value, which is an `Iterator` that yields a `String` value for each argument.

- To pass arguments to your executable while developing with cargo:

```
cargo run -- arg1 arg2 arg3
```

- Let's take a look at `lec03/ex0_cli_args/src/main.rs`

Each shell session has a "working directory"

- The **pwd** program prints the **p**ath of your current **w**orking **d**irectory

```
$ pwd
/home/vagrant
```

- When you sign-in to a machine your working directory is your user's $HOME directory.
  - In the case of our virtual machine, the user is `vagrant` and its $HOME directory is `/home/vagrant`

- The **cd** command **c**hanges your shell session's working **d**irectory

```
$ cd 590-material-KrisJordan
$ pwd
/home/vagrant/590-material-KrisJordan
```

`ls` lists the files in a directory

- The **`ls`** program lists the files in your working directory by default

- If you invoke **`ls`** with arguments its behavior changes.
  - To learn about the arguments of a program, refer to its manual using the man program, for example, `man ls`

    `SYNOPSIS: ls` `[OPTION]... [FILE]...`

- Example: **`ls -a`**

  Option `-a` lists all files. Don't ignore "hidden" files prefixed with a . (dot files)

- Example: **`ls /vagrant`**

  This prints all files in the /vagrant directory even if your current working directory is elsewhere.

What are the strange file entries `.` and `..` ?

- Two special, hidden entries are in *every* directory.
  - You will see them when you list all entries in a directory with **`ls -a`**

- Each is a reference to a directory
  - The single dot **`.`** is a self-reference to the directory it is contained in
  - The double dot **`..`** is a reference to the directory's parent directory

- The double dot is commonly used to change your working directory back to a parent:

```
cd .. # navigate to the parent directory
cd ../.. # navigate to the grandparent directory
```

- It can be used anywhere a file path is valid, though!

```
ls .. # list the files of my parent directory
```

Your shell has variables just like programs do.

- ***Environment variable*** names are prefixed with a dollar symbol and **$ALL_CAPS**
- For example, the **$HOME** variable is the path of your user's home directory.
- Referencing an environment variable in a command causes the shell to evaluate it and replace it with the variable's value ***before the command is run***
  (Unless you surround it in single quotes.)
- For example:

```
$ echo '$HOME is' $HOME
$HOME is /home/vagrant
```

- The **printenv** command will show you all of the environment variables of your shell.

The operating system gives every process all the environment variables it was started with

- Every programming language has a means for you to read environment vars.
- To access a program's environment variables in Rust, call:

```
std::env::vars()
```

- This function returns an **`Iterator`** that yields **`(String, String)`** tuples for the **name** and **value** of each environment variable.
- Lets take a look at **`lec03/ex1_cli_env_vars/src/main.rs`**

How does the shell know where programs are?

- If a command like **`echo`** is just another program started by the shell, then where is **`echo`**`'s` executable program file located?

- The **`echo`** program prints out the location of a program in the file system:

```
$ which echo
/bin/echo
```

- But, if ***`which`*** is just another program how does the *shell* know where `which` is?!?

$PATH - The Paths the Shell Looks in for Programs

The **$PATH** environment variable is the list of directories the shell checks for a program, in order, until the first match it finds.

```
$ echo $PATH
/home/vagrant/.cargo/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin
:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:./target/debug
```

If you're ever asked to add a directory to your PATH (including on Windows/Mac) it's so that the programs you're installing will run from the shell as if they're built-in.

Tips for Working on Rust Projects in 590

- When working on a project in Rust, you should navigate to the base directory of the project.

- You can open files in child directories like:
  `vim src/main.rs`

- In development, your compiled program file is built-in:
  `./target/debug/<name_of_your_program>`

- You'll notice the VM's $PATH has the debug directory of the current project as its last entry:
  `./target/debug`
  - So if you are in the base directory of your project and you have built your project (either via cargo build or cargo run), then you can use `<name_of_your_program>` directly in the shell.

- This $PATH entry was established (by me!) in the file **`$HOME/.bash_profile`**
  - If you want to add additional entries to your **`$PATH`** or customize your Bash shell further, feel free to tinker with that file in **`vim`**
  - When you first login to your shell, the commands in the file `$HOME/.bash_profile` are run as a part of logging in.