Lecture 21 (3-6-26)

Compiling and Building (C)

Motivating Questions

1. Why bother with **C**? Why not just **Python** or **C++**?
2. What happens when we **compile** a program? A library?
3. What exactly is a **Makefile**? How does it work?

C : Language of Unix

- One of the creators of **Unix**, **Dennis Ritchie,** developed **C** as a means of writing portable **assembly code:**
  - C is an **imperative procedural** language
  - Provides **low-level** access to **memory**
  - Maps efficiently to **machine instructions**
  - Requires minimal **run-time**
  - First-class citizen of **Unix**
- C is not interpreted, so you need to do the translation before hand.
  - via compiling

`hello.c`

```c
#include <stdio.h>

int main(int: argc, char *argv){

    printf("Hello World\n");

    return 0;
}
```

```
file hello.exe
# this command gives you info on the file
```

```
readelf -a hello.exe | less
#this command also gives you info on the file
#ELF : Executable and Linking Format -> default format on LINUX
```

Compiling

- Compiling is a **pipeline**
- First step: **Preprocessing**
  - copy and pastes all your header files into your program
  - things such as libraries, headers
  - the preprocessor **expands** all those headers
  - usually indicated with a pound (`#`) sign
- Second Step: **Compiling**
  - a grammar check is done (syntax)
  - optimization is done (such as **instruction selection**)
    - `printf()` was changed to `puts` in the assembly
      - since our `hello.c` doesn't need a **formatting** string
- Third Step: **Assemble**
  - take assembly code and produce an object file (binary)
  - not all binary code is executable
    - it needs to be packaged
- Fourth Step: **Link**
  - takes the binary code and packages up the binary code to produce an executable
- NOTE: often the first three steps are refereed to as "compile" and step 4 is referred to as "link"
- What does it mean to be "dynamically linked" as an executable?
  - the program does not contain all the library code it needs inside the executable file. Instead, it loads shared libraries at runtime.
  - with "dynamically linked" libraries, you dish out a security update to everyone using the library much easier in comparison to statically linked libraries
- There are also "statically linked" executables:
  - these simply package the libraries you need in the executable
  - gives you a much faster startup time, since you don't have to go looking for libraries
  - but they are much larger in size than "dynamically linked" executables

```
strace hello.exe
```

- `strace` tells you all the system calls the program makes to the operating system

---

Makefiles

- A makefile is a **directed acyclic** graph
  - each rule you specify is a node in that graph
- "Convert these sources into this target, using these commands"

```
# Target:       SOURCES
#        COMMAND

CC=    gcc
all:    hello.dynamic hello.static

hello.o:    hello.c
     $(CC) -c -o hello.o hello.c

hello.dynamic:    hello.o
     $(CC) -o hello.dynamic hello.o

hello.static:    hello.o
     $(CC) -static -o hello.static hello.o
```

- `$@` and `$<` are **automatic** variables in Makefiles
  - you should be aware of these