Build Systems

`gcc -I inc -o app $(find . -name *.c) -lsomelib`

Overview

- Building programs
- Build systems
- `make`
- Other build systems

What are programs?

- Sequence of instructions to perform
- Typical computers speak binary ("machine code")
  - Not all computers speak the same kind of binary
  - "Add 5 to variable"
  - x86-64: `[0x48,0x83,0xc0,0x05]`
  - aarch64 (ARMv8): `0xe2800005`
- Compiled high-level languages get turned into this binary form
  - e.g. C, C++, Java*
- Interpreted high-level languages are interpreted by another program
  - The other program is probably in binary form
  - (Or not, but at some point there will be binary!)
  - e.g. Shell scripts, Python

Building programs

- Traditional compiled programs have multiple steps to produce an executable
- Source code: human readable code in a (high-level) language
    - Assembly code: human readable "low-level" code that maps to CPU-understandable commands
    - x86-64: `add rax, #5` -> `[0x48,0x83,0xc0,0x05]`
    - aarch64 (ARMv8): `add r0, r0, #5` -> `0xe2800005`
- Object code: chunk of CPU-understandable machine code
    - File formats include additional metadata for tools to deal with
    - Can have dangling references to functions or other data to be resolved when *linking*
- Executable: fully put together ("linked") chunks of machine code
    - Ready for the operating system to load and run as a new process!
    - On many systems has the same file format as object code

Steps

- **Compiling**: turn high-level code into lower-level code
  - High-level source to lower-level "high-level" language (e.g. Java to C)
  - High-level source to assembly
  - High-level source to object code
- **Assembling**: turn low-level (assembly) code into object code
- **Linking**: putting object code together into something usable
  - Object code is usually just floating chunks of machine code
  - Produces executables: has a starting point (e.g. `main`), has resolved dangling references
  - Produces libraries: code that other programs can call on

Building programs

```
#
#
#
gcc -o app file1.c file2.c file3.c
```

Building programs

```
gcc -c -o file1.o file1.c
gcc -c -o file2.o file2.c
gcc -c -o file3.o file3.c
gcc -o app file1.o file2.o file3.o
```

Building code

```
gcc -o app file1.c file2.c file3.c
```

```
gcc -c -o file1.o file1.c
gcc -c -o file2.o file2.c
gcc -c -o file3.o file3.c
gcc -o app file1.o file2.o file3.o
```

- It can be annoying to type these out every time you compile...

Build systems

What is a build system?

- Tool/system to automate building software
  - Compilation
  - Packaging
  - Testing

A simple build system

```
---
build.sh
---
#!/bin/bash
gcc -o myapp src/file1.c src/file2.c src/file3.c src/main.c
```

```
---
build.sh
---
#!/bin/bash
gcc -o myapp $(find src -name "*.c")
```

Some issues

- Can be a bit of work to custom write a script, especially with larger projects
- Will blindly compile everything, every time
- What if we made a small change to one file and didn't want to recompile all the code?

Aside: incremental building

- It takes my lab computer about 30 minutes to do a clean build of LLVM and Clang while maxing out my CPU's 8 (logical) cores
    - Hours if I restrict how many cores I give it...
    - Took ~4 hours to clean build Android + camera driver at one of my internships
- Imagine if I had to recompile *everything* every time I made a small code change
- Put together independent bits instead of compiling/building everything every time
- Classic model: C and C++ programs
    - Compile individual C and C++ files into *object code* (`.o` files)
    - *Link* the object code files into the final output executable binary
    - Change only one C or C++ file? Just build the object code for that file, then link the object code
- ...now a simple shell script doesn't seem to cut it

Make

(We'll be focusing on GNU Make as it's probably the most popular)

- Classic tool that helps with build automation
- Provides more abstractions over a plain shell script
- Invoke it by running `make`
- Will look for a `Makefile` (or `makefile`) to run
  - (It's actually possible to run without a `Makefile`, but we won't really get into that)

General Makefile rule structure

- The `Makefile` will specify **rules** that have **prerequisites** that have to be met/built before running its **recipe** to build a **target** file

```
target: prerequisites
	recipe # <- actual tab character, not spaces!
```

- Make is able tell if the built **target** file is newer than **prerequisite** files to avoid unnecessarily performing the **recipe**
- The **recipe** consists of shell commands
- `make <target>` will build a specific **target**

Simple example

```
myapp: src/file1.c src/file2.c src/file3.c src/main.c
    gcc -o myapp src/file1.c src/file2.c src/file3.c src/main.c
```

A bit more sophisticated

```
myapp: src/file1.c src/file2.c src/file3.c src/main.c
    gcc -o $@ $^
```

Invocation

`$ make myapp`

Philosophy

- Overall idea is to have **rules** that depend on other **rules** that build smaller bits of the system
  - e.g. building the final executable depends on object code, which depends on corresponding source code files
- This composability means that we can incrementally build our project
  - Invaluable with enormous code bases: don't want to recompile *every* file of the Linux kernel if you made a single line change to one file
- Can have additional rules that run/test/debug the application and clean the directory of build output

Make concepts

Make gives us more abstractions to make our lives easier
It's a pretty deep tool; we're going to look at the basics

- Targets and rules
- "Phony" targets
- Powerful variable assignments
- Functions and other expansions
- Automatic variables
- Pattern matching

Targets and rules

```
target: prerequisites
	recipe # <- actual tab character, not spaces!
```

- **Prerequisite** targets will be built before the **target**'s **recipe** will be able to be run
- The **recipe** consists of shell commands
- `make <target>` will build a specific **target**
- The **target** is assumed to be some actual file that gets produced
- Make is able tell if the built **target** file is newer than **prerequisite** files to avoid unnecessarily performing the **recipe**
  - Done by determining if the **target** is "newer" than the **prerequisites** by modification timestamp
  - `touch` can update this timestamp
- If there are no **prerequisites** and the **target** file is present, the **recipe** won't be run again

Exercise 1:

- Download and extract this archive

```
https://www.eecs.umich.edu/courses/eecs201/fa2022/files/examples
```

  - You can use `wget` or `curl -O`
  - Unarchive with `tar xzf make.tar.gz`
  - `cd` into the `make` directory

Write a Makefile that:

- Produces an output executable file called `app`
  - `gcc -o <output file name> <source files>`
- Have proper prerequisites
  - `gcc` shouldn't run again if you've already built `app`
  - Try `touch`ing a source file and see what happens when you `make`

"Phony" targets

- What if you have a **target** that you want to be a word/concept?
  - e.g. `clean`, `all`, `test`
- If a file called `clean` or `all` is present, the target won't ever be run
- The `.PHONY` target can specify phony targets that don't have actual files

```
.PHONY: all clean test
```

- Common phony targets
  - `all`: build everything
  - `clean`: delete generated files
  - `test`: run tests

Variable assignments

- You can define variables in Makefiles as well (you can put spaces around the '='!)
- Often times are things like compilation flags, compiler selection, directories, files etc.
- To use Makefile variables and "expand" them, you use `$(varname)` or `${varname}`
  - Parentheses are more common

Two flavors of variables

- Define how they get assigned and how they get expanded
- `varA = $(varB)` recursively **expands the right hand side whenever the variable gets expanded**
  - If `varB` gets reassigned after this, an expanded `varA` will expand the current value of `varB`
  - "`varA`'s value is whatever `varB`'s is"
- `varA := $(varB)` performs a simple expansion, taking on the **value of the right hand side at assignment time**
  - If `varB` gets reassigned after this, an expanded `varA` will expand to the value of `varB` when `varA` got assigned
  - "`varA`'s value is "some-value""
- `varA ?= bar` will assign variable `varA` if it hasn't been assigned before

Exercise 2:

- Modify the Makefile from the previous Exercise 1
- Use a variable called `CC` for the C compiler (e.g. `gcc`)
- Use a variable to have a list of the source code files
- Use a variable for the output executable

Automatic variables

- In a rule, Make has some automatically assigned variables
- `$@`: **target**'s file name
- `$<`: First **prerequisite**
- `$?`: All **prerequisites** newer than the target
- `$^`: All **prerequisites**
- ...and more

Exercise 3:

- We'll be adding the object code step
- Modify the Makefile from the previous exercises
- Use a variable for object code files
  - Each file will have the same name but the `.o` extension
- Add rules to compile each object code file with prerequisite
  - Use the `-c` flag to tell the compiler to produce object code
- Modify the output executable to "compile" (link) the object code
  - Be sure to make sure the prerequisites reflect this
- Take advantage of automatic variables

Functions and other expansions

- There are some functions that can be expanded to help with text manipulation
- `$(wildcard <pattern>)` can expand to a file list for files that match the pattern
  - `$(wildcard *.c)`
- `$(shell <commands>)` can run a shell command and expand to the command's output
  - `$(shell find . -name "*.c")`
- `$(<var>:<pattern>=<replacement>)`, known as a substitution reference, can perform replacements
  - `$(SOURCES:%.c=%.o)`
- There's a lot more cool ones as well, check out the manual :)

Using what we've learned so far...

```
CC := gcc
BIN := myapp
SRCS := $(shell find src -name *.c)
$(BIN): $(SRCS)
    $(CC) -o $@ $^
```

Pattern matching

Pattern rules

```
%.o : %.c
    $(CC) -c -o $@ $<
```

- Uses `%` to match file names
- This example compiles `.c` files into `.o` files
- Note that this is a general rule that applies to all `.o` files
  - This is known as an **implicit rule**

Static pattern rules

```
OBJS := $(SRCS:src/%.c=obj/%.o) # substitution reference
$(OBJS): obj/%.o : src/%.c # static pattern rule
    $(CC) -c -o $@ $<
```

- Can narrow down a pattern rule to a particular list of targets

Exercise 4:

- Automatically find the source code files in the directory
- Automatically determine the object code files based off of the source code files
- Replace the object code rules with a (static) pattern rule

Make

- This is a brief overview of some of the features of Make
- This is by no means a comprehensive look at Make: refer to the manual for more features and details
- **Make isn't just for compiling code: you can use it to build anything that has a sense of dependencies**

Other build systems

- Make is a fairly general build system, but other build systems have more abstractions and may be tailored towards a particular language
- General: Ninja, CMake (actually more of a Makefile generator)
- Java: Ant, Maven, Gradle
- Ruby: Rake
- Continuous integration: Jenkins, Travis CI

Demo

Questions?