Week 7

Announcements

- HW5, ADV5 due by 11:59 PM on Feb 26
- HW6, ADV6 due by 11:59 PM on Mar 4
    - Will release ADV6 tonight or Saturday: will be easy and not require going to office hours

Lecture 7: Build Systems

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

Overview

- Build systems
- `make`
- Other build systems

Build systems

Build systems

Q: Who has used a build system?

What is a build system?

- Typing out `gcc -o myapp file1.c file2.c file3.c main.c` every time you want to compile can be annoying...

What is a build system?

- Typing out `gcc -o myapp file1.c file2.c file3.c main.c` every time you want to compile can be annoying...
- Tool to automate building software
  - Compilation
  - Packaging
  - Testing

A simple build system

A throwback to HW2...

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

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

```
./build.sh
```

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
  - (Hours if I restrict how many cores I give it...)

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
  - (Hours if I restrict how many cores I give it...)
- Imagine if I had to recompile *everything* every time I made a small code change

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
  - (Hours if I restrict how many cores I give it...)
- 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/C++ programs
  - Compile individual C/C++ files into *object code* (`.o` files)

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
  - (Hours if I restrict how many cores I give it...)
- 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/C++ programs
  - Compile individual C/C++ files into *object code* (`.o` files)
  - *Link* the object code files into the final output executable binary

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
  - (Hours if I restrict how many cores I give it...)
- 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/C++ programs
  - Compile individual C/C++ files into *object code* (`.o` files)
  - *Link* the object code files into the final output executable binary
  - Change only one C/C++ file? Just build the object code for that file, then link the object 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
  - (Hours if I restrict how many cores I give it...)
- 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/C++ programs
  - Compile individual C/C++ files into *object code* (`.o` files)
  - *Link* the object code files into the final output executable binary
  - Change only one C/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

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

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`

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)

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

```
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
```

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 $@ $^
```

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 compiling individual source code files into corresponding object code

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 compiling individual source code files into corresponding object code
- 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

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 compiling individual source code files into corresponding object code
- 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

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

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**

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

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**

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**
- If there are no **prerequisites** and the **target** file is present, the **recipe** won't be run again

"Phony" targets

- What if you have a **target** that you want to be a word/concept?
  - e.g. `clean`, `all`, `test`

"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

"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
```

Variable assignments

- You can define variables in Makefiles as well (you can put spaces around the '='!)

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.

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

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

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`

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 := $(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

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 := $(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

Another operator

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)`

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)`
- `$(<var>:<pattern>=<replacement>)`, known as a substitution reference, can perform replacements
  - `$(SOURCES:%.c=%.o)`

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)`
- `$(<var>:<pattern>=<replacement>)`, known as a substitution reference, can perform replacements
  - `$(SOURCES:%.c=%.o)`
- `$(shell <commands>)` can run a shell command and expand to the command's output
  - `$(shell find . -name "*.c")`

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

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

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**

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)
$(OBJS): obj/%.o : src/%.c
    $(CC) -c -o $@ $<
```

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

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

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

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 time

Questions