Advanced Homework 6

Assigned: Friday, February 12, 11:00AM

Due: Before the first lecture on Friday, February 26

Submission Instructions

To receive credit for this assignment you will need to stop by someone's office hours, demo your running code, and answer some questions.

> This assignment asks to you work with any EECS project. You may use a current project, a previous project, or the EECS 280 W15 project we've been using throughout the homeworks so far.
>
> If you do use a current EECS project, remember to commit everything before experimenting (you are using version control for all your projects, right?). Some of the examples reference the EECS 280 W15 project, however they should be easy enough to translate to any other project.

1 Compiler-Assisted Makefiles

As we saw in the regular homework, it can be somewhat hard to correctly list all of the dependencies of a file. However, there is one program that knows all of the files that the compiler needs to read in order to compile your code, and that program is the compiler!

Try running `gcc -MM simple_test.cpp` (also try running `gcc -M simple_test.cpp`, do you understand the difference?). What does `-MMD` do?

Modify the makefile to auto-generate the dependencies for `simple_test` and use them when building `simple_test`.

Notes / tips:

- You **do not** need to set up the whole project to use auto-generated dependencies. This is tricky to do correctly. The goal is for you to play with this a little just to see how the mechanisms work. People have since developed tools that are easier to use and understand than gcc+make for auto-generating dependencies.[1]
  - If your solution is full of `%`'s, you should be prepared to explain in depth how everything is working and what everything is doing.
- Auto-generated dependencies generally add additional rules. Their purpose is to make sure that *updates* work correctly. They are not needed for the first build. This means that to start, you only need to modify an existing rule that already builds code correctly to also build dependency information.

Submission checkoff:

- ☐ Explain the changes you made to your Makefile
  - ☐ Show that auto-generated dependencies are working correctly (i.e. `make && touch recursive.h && make`).

[1] This is not to say that gcc's dependency generation is not useful. Indeed, ninja uses it. Ninja, however explicitly understands that generating dependencies is a step in the build process, making it conceptually easier and cleaner than make.

2 Alternative Build Systems

Make is a (relatively) simple program and comes installed on nearly every system. However, as we've seen, make can be hard to use, and it can be hard to get things completely correct. There are many other build systems out there. This question asks you to try out my personal favorite and another one of your own choosing.

2.1 tup

The reason I like tup is that it aspires to be *provably correct*. It uses FUSE to monitor every file system call, which means that if your compile command reads or writes a file, tup knows about it. This also means that tup can *implicitly* do the same thing as gcc's `-M` options—the first time the build runs it builds up a database of all the static files (mostly header files) gcc reads and watches them for changes. Because the build system is doing this, it works automatically and transparently for every command, not just gcc. For generated files this lets tup verify that your dependencies are correct, e.g. if a rule attempts to run `./simple_test`, a file built by a different rule, but does not list `./simple_test` as an input dependency, tup will alert you with an error if you forgot to express a dependency.

Tup also supports variants, which will automatically build multiple variations of your code. As an example of why this is useful (likely near and dear to the heart of all 281 students), often it is useful to start and get things working with optimizations turned off (`-Og`)[2] and then turn optimizations to max (`-O3 -flto -DNDEBUG`) to get the speediest programs. Unfortunately, you can sometimes introduce bugs[3] when doing debug builds and then not notice until the next time you do a release build. Wouldn't it be great to just always build both?

Create a copy of an EECS project and convert it to use tup.

Set up (at least) two variants (e.g. "debug" and "release" variants).

Submission checkoff:

- ☐ Show off basic tup functionality and your Tupfiles
- ☐ Show off your variants and explain the differences between them, why did you choose those build options?

2.2 Another build system of your choosing

There are many to choose from, some of the more appropriate ones to consider may be cmake, scons, gradle, or ninja.

Choose another build system, read about it, and try to understand what it does that is different, "better", or "worse" compared to `make` or `tup`.

You don't need to set up a whole project using this build system, though playing with it a little may help deepen you understanding of what it provides and how it's different.

Submission checkoff:

- ☐ Which build system did you learn about? Why that one?
- ☐ What type of projects is this build system "best" at (what was it designed for)?
  - ☐ What does it do differently that makes it better for this application?

[2] Note, that's `-Og` *not* `-O0`. Level 0 turns off *all* optimizations and emits very slow and naïve code.
From the gcc man page, "Optimize debugging experience. -Og enables optimizations that do not interfere with debugging. It should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience."

[3] Or really useful warnings. Some warnings the compiler can only discover when optimizations are turned on.