Administrivia

- Reminder: Homework 5 due today. (E-mail.)
- Homework 6 on the Web, due next Wednesday.

Slide 1

Minute Essay From Last Lecture

- Almost everyone is finding *something* useful, though exactly what and how much varies. One person seems to be using a lot of what we've talked about and says it's speeding up his workflow. Good!
- One person mentioned that the course was helping her understand better what's happening in a terminal window. I hadn't thought of that as a goal for the course, but I should!

Slide 2

Slide 3

Homework 3 Essays

- Several people mentioned that the problems were helpful in understanding and/or fun. Good! A few mentioned liking having the option to make up one's own problem.
- One person commented on how much time he spent worrying about proper use of quotes. Not atypical, sadly — it's one of the uglier parts about writing shell scripts.

Slide 4

`make` — Recap

- Originally intended to make it easier to "build" large programming projects, recompiling only as needed.
- Input is a text file with a textual representation of dependency graph (in terms of targets and dependencies — "rules") and "recipes" for re-creating targets. Can be almost arbitrarily complex, including variable definitions, etc.
- `make` has many predefined rules (e.g., one to make `foo` from `foo.c`). Many/most make copious use of variables (e.g., `CFLAGS`) to allow you to supply some details. Use them when you can?

  (Review slides from last time?)

Slide 5

make — Overriding Variables at Runtime

- Something else that can be useful in makefiles is providing variables that can be overridden at runtime. For example, if in the makefile you have

  ```
  CFLAGS = -Wall -pedantic $(OPT)
  OPT = -O
  ```

  you can override `$OPT` with e.g., `make OPT=-g foo`.

Slide 6

Implicit Rules (Pattern Rules)

- In addition to predefined implicit rules, you can define similar rules — e.g., a makefile to compile `.c` files using the MPI C compiler:

  ```
  MPICC = /usr/bin/mpicc
  CCFLAGS = -O -Wall -pedantic

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

  `$<` is the first prerequisite (`.c` file here); `$@` is the target.

  (Note that this is for GNU `make`. Non-GNU `make` has a similar idea (“suffix rules”) with slightly different syntax.)

  (Note also that this is kind of a bogus example — you could get the same effect by just setting `CC` to point to the compiler you want.)

Slide 7

Other Uses For `make`

- One of the more painful aspects to using `make` is getting the dependencies right, in particular for `#include`'s in C programs. `make` can help with this, together with compiler option `-MM`. Discussed in some detail in GNU `make` manual, under "Generating Prerequisites Automatically".
- `make` can be used to automate things other than compiling programs. It's particularly useful for defining implicit rules. For example, I like using it to automate generating PDF (and HTML) from LaTeX source, sometimes with some preprocessing. (Possibly less necessary than it was, now that we have `pdflatex`.)

Slide 8

Minute Essay

- Have you used `make` in another class? (I hear Dr. Fogarty uses it in some classes, though he supplies the makefile(s)?)