Slide 1

Administrivia

- Reminder: Homework 5 due today.
- Homework 6 on the Web; due a week from Wednesday (in deference to its being midterms time . . .)

Slide 2

`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?

Slide 3

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 4

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`; partially discussed in GNU manual.
- `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 5

Minute Essay

- Anything noteworthy about Homework 5?