Slide 1

Administrivia

- Reminder: Homework 5 due Wednesday. Homework 6 to be on Web soon.

Slide 2

The `make` Utility

- Motivation: Most programming languages allow you to compile programs in pieces ("separate compilation"). This makes sense when working on a large program — when you change something, just recompile parts that are affected.
- Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.

Slide 3

Makefiles

- First step in using `make` is to set up "makefile" describing how files that make up your program (source, object, executable, etc.) depend on each other and how to update the ones that are generated from others. Normally call this file `Makefile` or `makefile`.

  Simple example (assuming `main.c` `#include`s `defs.h` and `foo.h`):

```
main:   main.o foo.o
        gcc -o main main.o foo.o
main.o: main.c defs.h foo.h
        gcc -c main.c
foo.o:  foo.c
        gcc -c foo.c
```

- When you type `make`, `make` figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Slide 4

Useful Command-Line Options

- `make` without parameters makes the first "target" in the makefile.
  `make foo` makes `foo`.
- `make -n` just tells you what commands would be executed — a "dry run".
- `make -f otherfile` uses `otherfile` as the makefile.

Defining Rules

- Define dependencies for a rule by giving, for each "target", list of files it depends on.
- Also give the list of commands to be used to recreate target.

  NOTE!: Lines containing commands must start with a tab character. Alleged paraphrase from an article by Brian Kernighan on the origins of UNIX:

  > The tab in makefile was one of my worst decisions, but I just wanted to do something quickly. By the time I wanted to change it, twelve (12) people were already using it, and I didn't want to disrupt so many people.

Phony Targets

- Normally targets are files to create (e.g., executables), but they don't have to be. So you can package up other things to do . . .
- Example — many makefiles contain code to clean up, e.g.:

```
clean:
	-rm *.o main
```

  To use — `make clean`.

Variables in Makefiles

- You can also define variables, e.g.:
  - List of object files needed to create an executable. Then use this list to specify dependencies, command.
  - Pathname for a command, options to be used for all compiles, etc.
- Example:

```
objs = main.o foo.o
CFLAGS = -Wall -pedantic
main:   $(objs)
        gcc $(CFLAGS) -o main $(objs)
```

Predefined Implicit Rules

- `make` already knows how to “make” some things — e.g., `foo` or `foo.o` from `foo.c`.
- In applying these rules, it makes use of some variables, which you can override.
- A simple but useful makefile might just contain:

```
CFLAGS = -Wall -pedantic -O
```

- Or you could use

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

and then optionally override the `-O` by saying, e.g., `make OPT=-g foo`.

Slide 9

Implicit Rules (Pattern 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) $<
```

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

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

Slide 10

Other Uses For `make`

- `make` can be used to automate things other than compiling programs. It’s particularly useful for defining implicit rules.

  Example: Makefiles to run `latex` and associated programs.

Minute Essay

- Did you learn about makefiles in PAD I? and/or have you used them before?