Makefiles, File I/O

CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:

Aaron Johnston
Andrew Hu
Daniel Snitkovskiy
Forrest Timour
Kevin Bi
Kory Watson
Pat Kosakanchit
Renshu Gu
Tarkan Al-Kazily
Travis McGaha

Administrivia

- Exercise 6 out today, due Wednesday morning
- *No exercise due Friday!* Exercise 7 will be released on Thursday and due the following Monday (4/22)

- Exercise Grading
  - New (imperfect) scale: Correctness [0-3], Tools [0-2], Style [0-3]
  - Can submit regrade requests via Gradescope for a few days after scores released. Can ask about it first on Piazza.

- Homework 0 grades out, pull repo to see feedback
- Homework 1 due Thursday (4/18) at 11:59 pm
  - Submit via **GitLab** (*i.e.* commit/push changes, then push tag)

Lecture Outline

- **Makefile Basics**
- File I/O with the C standard library
- System Calls

make Basics

- A makefile contains a bunch of triples:

```
target: sources
←Tab→ command
```

  - Colon after target is *required*
  - Command lines must start with a **TAB**, NOT SPACES
  - Multiple commands for same target are executed *in order*
    - Can split commands over multiple lines by ending lines with '\'

- Example:

```
foo.o: foo.c foo.h bar.h
	gcc -Wall -o foo.o -c foo.c
```

Using `make`

```
bash% make -f <makefileName> target
```

- Defaults:
  - If no `-f` specified, use a file named `Makefile`
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell
    - Set `SHELL` variable in makefile to ensure
- Target execution:
  - Check each source in the source list:
    - If the source is a target in the Makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

`make` Variables

- You can define variables in a makefile:
  - All values are strings of text, no "types"
  - Variable names are case-sensitive and can't contain '`:`', '`#`', '`=`', or whitespace

- Example:

```
CC = gcc
CFLAGS = -Wall -std=c11
foo.o: foo.c foo.h bar.h
	$(CC) $(CFLAGS) -o foo.o -c foo.c
```

- Advantages:
  - Easy to change things (especially in multiple commands)
  - Can also specify on the command line:
    (*e.g.* `make foo.o CC=clang CFLAGS=-g`)

More Variables

- It's common to use variables to hold lists of filenames:

```
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
	gcc -o widget $(OBJFILES)
clean:
	rm $(OBJFILES) widget *~
```

- `clean` is a convention
  - Remove generated files to "start over" from just the source
  - It's "funny" because the target doesn't exist and there are no sources, but it works because:
    - The target doesn't exist, so it must be "remade" by running the command
    - These "phony" targets have several uses, such as "`all`"…

"all" Example

```
all: prog B.class someLib.a
      # notice no commands this time

prog: foo.o bar.o main.o
      gcc -o prog foo.o bar.o main.o

B.class: B.java
      javac B.java

someLib.a: foo.o baz.o
      ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
      gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

Writing a Makefile Example

- "`talk`" program (find files on web with lecture slides)

`main.c` `speak.h` `speak.c` `shout.h` `shout.c`

Revenge of the Funny Characters

- Special variables:
  - **$@** for target name
  - **$^** for all sources
  - **$<** for left-most source
  - Lots more! – see the documentation

- Examples:

```
# CC and CFLAGS defined above
widget: foo.o bar.o
        $(CC) $(CFLAGS) -o $@ $^
foo.o: foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c $<
```

And more…

- There are a lot of "built-in" rules – see documentation
- There are "suffix" rules and "pattern" rules
  - Example:

```
%.class: %.java
	javac $<  # we need the $< here
```

- Remember that you can put *any* shell command – even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day…)

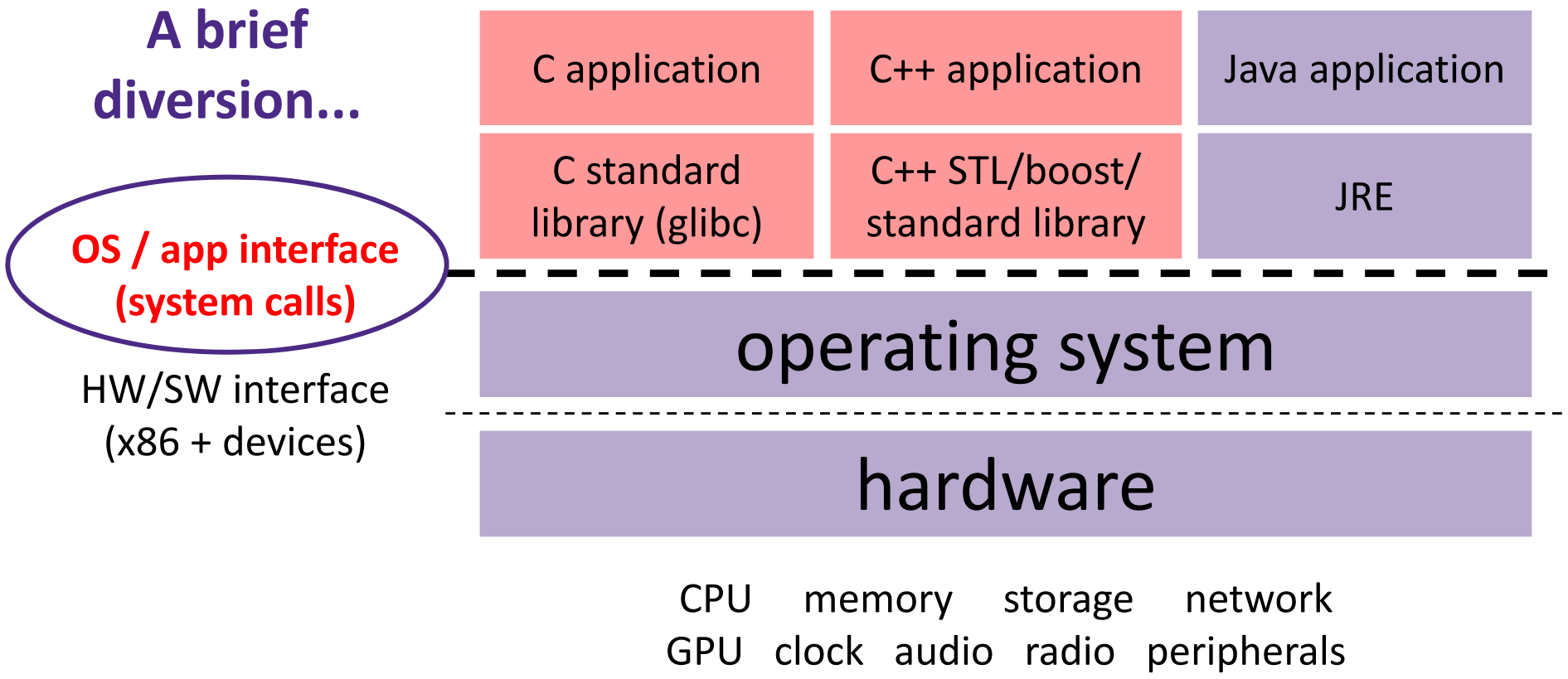
Lecture Outline

- Makefile Basics
- **File I/O with the C standard library**
- System Calls

These are essential material for the next part of the project (hw2)!

Remember This Picture?

A brief diversion...



File I/O

- We'll start by using C's standard library
  - These functions are part of `glibc` on Linux
  - They are implemented using Linux system calls

- C's `stdio` defines the notion of a stream
  - A way of reading or writing a sequence of characters to and from a device
  - Can be either *text* or *binary*; Linux does not distinguish
  - Is *buffered* by default; `libc` reads ahead of your program
  - Three streams provided by default: `stdin`, `stdout`, `stderr`
    - You can open additional streams to read and write to files
  - C streams are manipulated with a `FILE*` pointer, which is defined in `stdio.h`

C Stream Functions

- Some stream functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`
    - Closes the specified stream (and file)
  - `int fprintf(stream, format, ...);`
    - Writes a formatted C string
      - `printf(...);` is equivalent to `fprintf(stdout, ...);`
  - `int fscanf(stream, format, ...);`
    - Reads data and stores data matching the format string

C Stream Functions

- Some stream functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`
    - Closes the specified stream (and file)
  - `size_t fwrite(ptr, size, count, stream);`
    - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*
  - `size_t fread(ptr, size, count, stream);`
    - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

Error Checking/Handling

- Some error functions (complete list in `stdio.h`):
  - `void perror(message);`
    - Prints `message` and error message related to `errno` to `stderr`
  - `int ferror(stream);`
    - Checks if the error indicator associated with the specified stream is set
  - `int clearerr(stream);`
    - Resets error and eof indicators for the specified stream

C Streams Example

cp_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define READBUFSIZE 128

int main(int argc, char** argv) {
  FILE *fin, *fout;
  char readbuf[READBUFSIZE];
  size_t readlen;

  if (argc != 3) {
    fprintf(stderr, "usage: ./cp_example infile outfile\n");
    return EXIT_FAILURE;        // defined in stdlib.h
  }

  // Open the input file
  fin = fopen(argv[1], "rb");  // "rb" -> read, binary mode
  if (fin == NULL) {
    fprintf(stderr, "%s -- ", argv[1]);
    perror("fopen for read failed");
    return EXIT_FAILURE;
  }
  ...
```

C Streams Example

cp_example.c

```
int main(int argc, char** argv) {

  ...   // previous slide's code

  // Open the output file
  fout = fopen(argv[2], "wb");  // "wb" -> write, binary mode
  if (fout == NULL) {
    fprintf(stderr, "%s -- ", argv[2]);
    perror("fopen for write failed");
    return EXIT_FAILURE;
  }

  // Read from the file, write to fout
  while ((readlen = fread(readbuf, 1, READBUFSIZE, fin)) > 0) {
    if (fwrite(readbuf, 1, readlen, fout) < readlen) {
      perror("fwrite failed");
      return EXIT_FAILURE;
    }
  }

  ...   // next slide's code

}
```

C Streams Example

cp_example.c

```
int main(int argc, char** argv) {

  ...   // two slides ago's code

  ...   // previous slide's code

  // Test to see if we encountered an error while reading
  if (ferror(fin)) {
    perror("fread failed");
    return EXIT_FAILURE;
  }

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

Extra Exercise #1

- Write a program that:
  - Uses `argc`/`argv` to receive the name of a text file
  - Reads the contents of the file a line at a time
  - Parses each line, converting text into a `uint32_t`
  - Builds an array of the parsed `uint32_t`'s
  - Sorts the array
  - Prints the sorted array to `stdout`

- Hint: use `man` to read about `getline`, `sscanf`, `realloc`, and `qsort`

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ./extra1 in.txt
5
52
1213
3231
bash$
```

Extra Exercise #2

- Write a program that:
  - Loops forever; in each loop:
    - Prompt the user to input a filename
    - Reads a filename from `stdin`
    - Opens and reads the file
    - Prints its contents to `stdout` in the format shown:

```
00000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
00000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
00000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
00000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
00000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
00000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
00000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
00000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
00000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
00000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
000000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
... etc ...
```

- Hints:
  - Use `man` to read about `fgets`
  - Or, if you're more courageous, try `man 3 readline` to learn about `libreadline.a` and Google to learn how to link to it