C Details, File I/O, and System Calls
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Audrey Seo
Deeksha Vatwani
Derek de Leuw
Katie Gilchrist

Administrivia

- HW0 grades are posted!
  - Regrade requests can be done on gradescope
  - Questions about your grade can go in private edboard messages.
- Exercise 5 due this morning
  - Reminder: there is no exercise 4, we're skipping it this quarter.
- New exercise (ex6) posted today, due Wednesday morning
- HW1 due on **Thursday @ 11pm**

Lecture Outline

- **Final C Details**
- File I/O with the C standard library
- OS Abstraction

C Preprocessor Example

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

```c
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

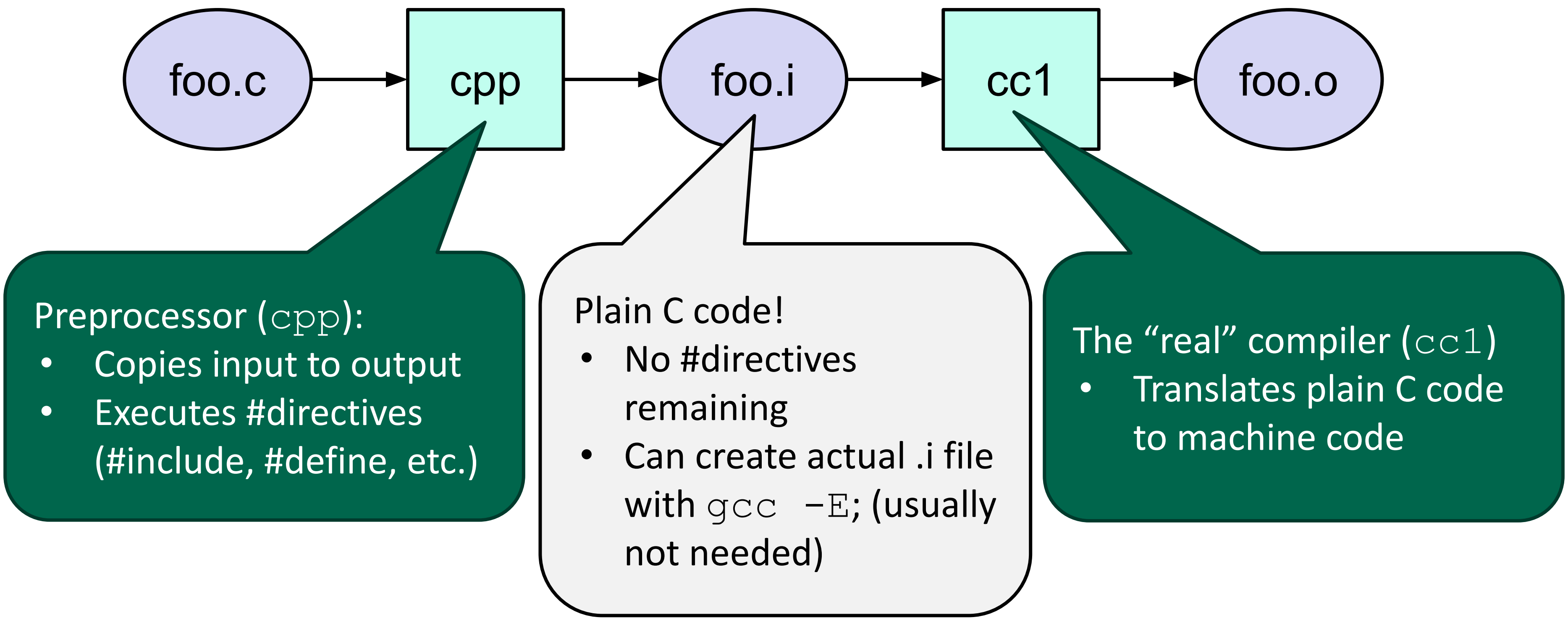
```c
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
  return 0;
}
```

What Is gcc Really Doing?

- `gcc` runs other programs that do the "real work"
- Here's what `gcc` runs to translate `foo.c` to `foo.o`
  - `gcc -c foo.c`



Other Preprocessor Tricks

- A way to deal with "magic numbers" (constants)

```
int globalbuffer[1000];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * 3.1415;
  *area = rad * 3.1415 * 3.1415;
}
```

Bad code
(littered with magic constants)

```
#define BUFSIZE 1000
#define PI 3.14159265359

int globalbuffer[BUFSIZE];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * PI;
  *area = rad * PI * PI;
}
```

Better code

Macros

- `#define` definitions can take arguments; these are called "**macros**":

```
#define ODD(x)  ((x) % 2 != 0)

void foo() {
  if ( ODD(5) )
    printf("5 is odd!\n");
}
```

cpp →

```

void foo() {
  if ( ((5) % 2 != 0) )
    printf("5 is odd!\n");
}
```

- Beware of operator precedence issues!
  - Use parentheses

```
#define ODD(x)  ((x) % 2 != 0)
#define WEIRD(x)  x % 2 != 0

ODD(5 + 1);

WEIRD(5 + 1);
```

cpp →

```
((5 + 1) % 2 != 0);

5 +(1 % 2)!= 0;
```

Conditional Compilation

- You can change what gets compiled
  - In this example, `#define TRACE` before `#ifdef` to include debug **`printfs`** in compiled code

```
#ifdef TRACE
#define ENTER(f) printf("Entering %s\n", f)
#define EXIT(f)  printf("Exiting  %s\n", f)
#else
#define ENTER(f)
#define EXIT(f)
#endif

// print n
void pr(int n) {
  ENTER("pr");
  printf("\n = %d\n", n);
  EXIT("pr");
}
```

You can give macros blank definitions to make them do nothing

ifdef.c

Defining Symbols

- Besides `#define`s in the code, preprocessor values can be given as part of the `gcc` command:

```
bash$ gcc -Wall -g -DTRACE -o ifdef ifdef.c
```

- `assert` can be controlled the same way – defining `NDEBUG` causes `assert` to expand to "empty"
  - It's a macro – see `assert.h`

```
bash$ gcc -Wall -g -DNDEBUG -o faster useassert.c
```

```
bash$ gcc -Wall -DBAR -DFOO -o condcomp condcomp.c
bash$ ./condcomp
```

```c
#include <stdio.h>
#ifdef FOO
#define EVEN(x) !((x)%2)
#endif
#ifndef DBAR
#define BAZ 333
#endif

int main(int argc, char** argv) {
  int i = EVEN(42) + BAZ;
  printf("%d\n",i);
  return EXIT_SUCCESS;
}
```

```
bash$ gcc -Wall -DBAR -DFOO -o condcomp condcomp.c
bash$ ./condcomp
```

```c
#include <stdio.h>
#ifdef FOO
#define EVEN(x) !((x)%2)
#endif
#ifndef DBAR
#define BAZ 333
#endif

int main(int argc, char** argv) {
  int i = EVEN(42) + BAZ;
  printf("%d\n",i);
  return EXIT_SUCCESS;
}
```

```
bash$ gcc -Wall -DBAR -DFOO -o condcomp condcomp.c
bash$ ./condcomp
```

```c
#include <stdio.h>

#define EVEN(x) !((x)%2)

#ifndef DBAR
#define BAZ 333
#endif

int main(int argc, char** argv) {
  int i = EVEN(42) + BAZ;
  printf("%d\n",i);
  return EXIT_SUCCESS;
}
```

```
bash$ gcc -Wall -DBAR -DFOO -o condcomp condcomp.c
bash$ ./condcomp
```

42%2 = 0
!0 = 1
1 + 333 = 334
Final Output: 334

```
#include <stdio.h>

#define EVEN(x) !((x)%2)


#define BAZ 333


int main(int argc, char** argv) {
  int i = !((42)%2) + 333;
  printf("%d\n",i);
  return EXIT_SUCCESS;
}
```

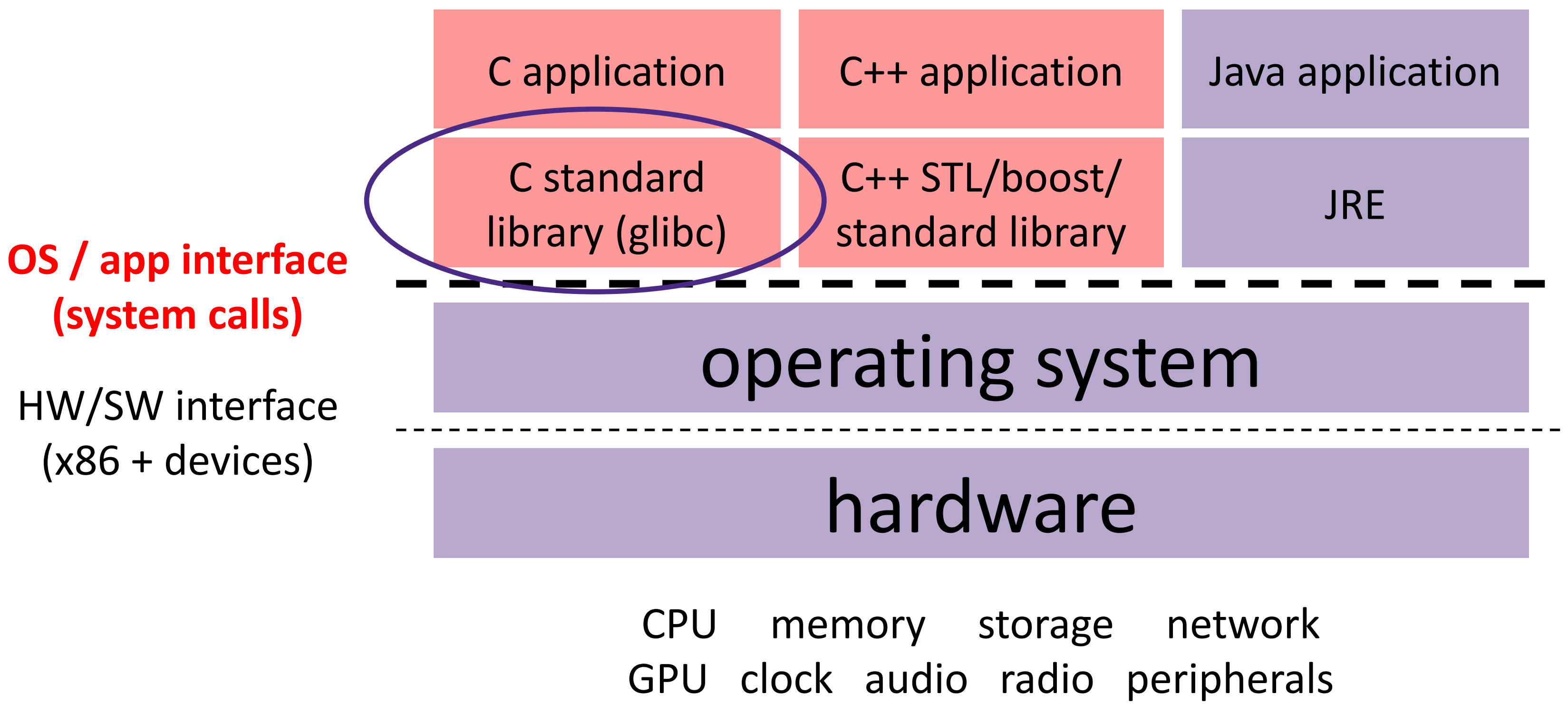
Additional C Topics

- Teach yourself!
  - String library functions in the C standard library
    - `#include <string.h>`
      - strlen(), strcpy(), strdup(), strcat(), strcmp(), strchr(), strstr(), …
    - `#include <stdlib.h>` or `#include <stdio.h>`
      - atoi(), atof(), sprint(), sscanf()
  - `union`s and what they are good for
  - `enum`s and what they are good for
  - Pre- and post-increment/decrement
  - How to declare, define, and use a function that accepts a variable-number of arguments (`varargs`)
  - Harder: the meaning of the "`volatile`" storage class

Lecture Outline

- Final C Details
- **File I/O with the C standard library**
- OS Abstraction

Remember This Picture?



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
  - Three streams provided by default: `stdin`, `stdout`, `stderr`
    - You can open additional streams to read and write to files

C Stream Functions

- In the C Stream API, files are represented by a special pointer FILE*

- ```
  FILE* fopen(char* filename, char* mode);
  ```
  - Opens a stream to the specified file in the specified access mode
  - Returns NULL if it fails

- ```
  int fclose(FILE* stream);
  ```
  - Closes the specified stream (file)
  - Returns non-zero if it fails
  - But you can often assume it succeeds

C Stream Access Modes

- File access modes in the C Stream API are represented by strings (`char*`)
- Three main modes you should know about:
  - "`r`" - reading from the beginning of the file
  - "`w`" - writing - if the file already exists, overwrite it completely
  - "`a`" - appending - create the file if it doesn't exist, then write to the end.

Printing Errors

- If your file operations fail, use **`perror`** to print the exact error message

A global variable that some library functions set to indicate an error

  - `void perror(message);`
    - Prints `message` and error message related to `errno` to `stderr`

C Streams Example, Part 1

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];   // space for input data
  size_t readlen;

  // We'll handle wrong-number-of-arguments in a second


  // Open the input file
  fin = fopen(argv[1], "r");  // "r" -> read
  if (fin == NULL) {
    perror("fopen for read failed");
    return EXIT_FAILURE;
  }
  ...
```

Using C Streams: Reading and Writing

- ```
  size_t fwrite(void* ptr, size_t size, size_t count, FILE* stream);
  ```
  - Write an array of *count* elements of *size* bytes from *ptr* to *stream*
  - *size* is only a request; returns the number of elements ***actually*** written

- ```
  size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
  ```
  - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*
  - Returns the number of elements actually read

- In this class, we'll just be writing text data, so:
  - `ptr` is always a `char*`
  - `size` is always 1 (same as `sizeof(char)`)
  - `count` in `fwrite` is always `strlen(ptr)`

Using C Streams: Reading and Writing

- Stream objects *change* when you read **or** write them
- Each file stream holds a file position that it exists at (except stdin/stdout/stderr)
- When you read or write, you move the position
- If you open a new stream, the position resets (except with ‘a’ mode)

C Stream Functions

- Formatted I/O stream functions:
  - `int fprintf(FILE* stream, char* format, ...);`
    - Writes a formatted C string
      - `printf(...);` is equivalent to `fprintf(stdout, ...);`
  - `int fscanf(FILE* stream, char* format, ...);`
    - Reads data and stores data matching the format string
    - The inverse of **fprintf**

Error Checking/Handling

- Each stream has its own error indicator
  - ```
    int ferror(FILE* stream);
    ```
    - Checks if the error indicator associated with the specified stream is set
  - ```
    int clearerr(FILE* stream);
    ```
    - Resets error and eof indicators for the specified stream

More I/O functions in stdio.h, see cppreference.com

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];   // space for input data
  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

}
```

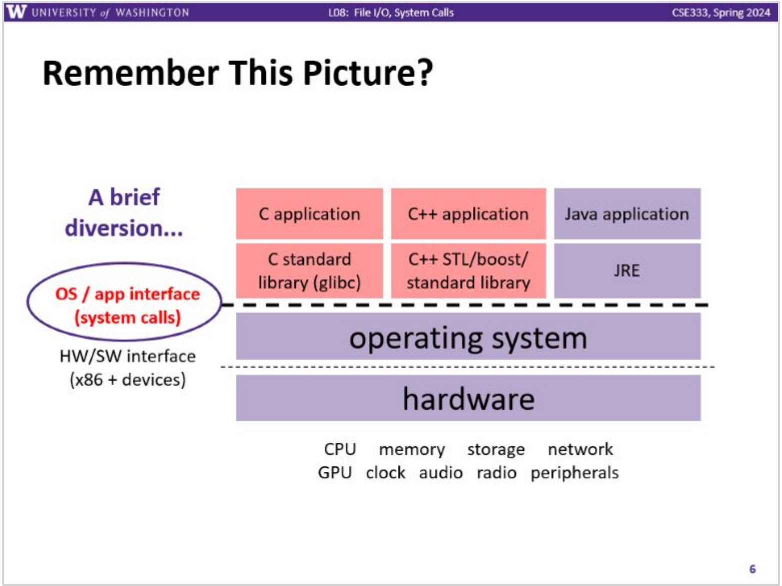
C Streams Example

cp_example.c

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

  ...   // code from previous 2 slides

  // 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;
}
```

Buffering

- By default, stdio uses buffering for streams:
  - Data written by `fwrite()` is copied into a buffer allocated by `stdio` inside your process' address space
  - As some point, the buffer will be "drained" into the destination:



Buffering

- By default, stdio uses buffering for streams:
  - Data written by **`fwrite()`** is copied into a buffer allocated by `stdio` inside your process' address space
  - As some point, the buffer will be "drained" into the destination:
    - When you explicitly call **`fflush()`** on the stream
    - When the buffer size is exceeded (often 1024 or 4096 bytes)
    - For `stdout` to console, when a newline is written (*"line buffered"*) or when some other function tries to read from the console
    - When you call **`fclose()`** on the stream
    - When your process exits gracefully (**`exit()`** or `return` from **`main()`**)

Why Buffer?

- Performance – avoid disk accesses
  - Group many small writes into a single larger write
  - Why minimize the number of writes? Disk Latency = 😱😱😱

- Convenience – nicer API
  - We'll compare C's `fread()` with POSIX's `read()` shortly

Why Buffer?

- Disk Latency = 😱😱😱 (Jeff Dean from LADIS '09)

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Why NOT Buffer?

- Reliability – the buffer needs to be flushed
  - Loss of computer power = loss of data
  - "Completion" of a write (*i.e.* return from **`fwrite`**`()`) does not mean the data has actually been written
    - What if you signal another process to read the file you just wrote to?
- Performance – buffering takes time
  - Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
  - Can potentially slow down high-performance applications, like a web server or database (*"zero-copy"*)

Disabling C's Buffering

- Explicitly turn off with `setbuf(stream, NULL)`
  - But potential performance problems: lots of small writes triggers lots of slower system calls instead of a single system call that writes a large chunk
- Use POSIX APIs instead of C's
  - No buffering is done at the user level
  - We'll see these soon

Poll Everywhere

pollev.com/uwcse33343aF

- Can you think of any other places where buffering might occur?

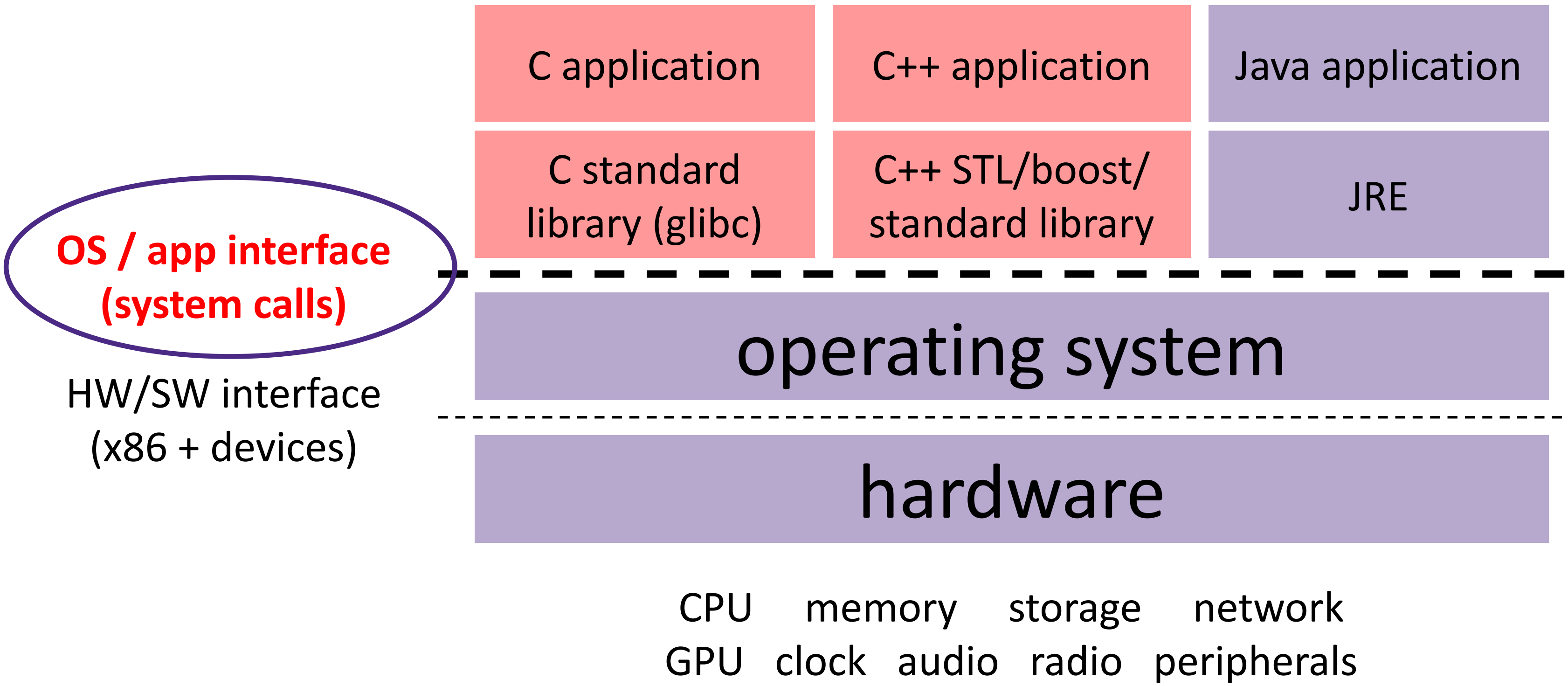
- Can you think of any other places where buffering might occur?

- The OS caches disk reads and writes in the file system *buffer* cache
- Disk controllers have caches too!
- Input from the user is buffered by the shell

Lecture Outline

- Final C Details
- File I/O with the C standard library
- **OS Abstraction**

What's an OS?

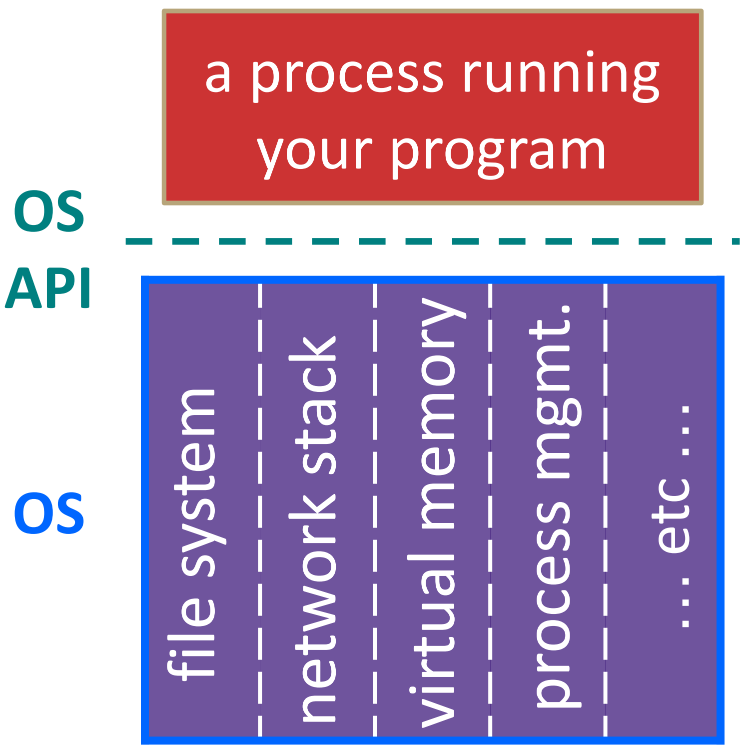

C application
C++ application
Java application
C standard library (glibc)
C++ STL/boost/ standard library
JRE
OS / app interface (system calls)
operating system
HW/SW interface (x86 + devices)
hardware
CPU memory storage network
GPU clock audio radio peripherals

What's an OS?

- Software that:
  - Abstracts away messy hardware devices
    - Provides high-level, convenient, portable abstractions (*e.g.* files, disk blocks)
  - Directly interacts with the hardware
    - OS is trusted to do so; user-level programs are not
    - OS must be ported to new hardware; user-level programs are portable
  - Manages (allocates, schedules, protects) hardware resources
    - Decides which programs can access which files, memory locations, pixels on the screen, etc. and when

OS: Abstraction Provider

- The OS is the "layer below"
  - A module that your program can call (with system calls)
  - Provides a powerful OS API – POSIX, Windows, etc.



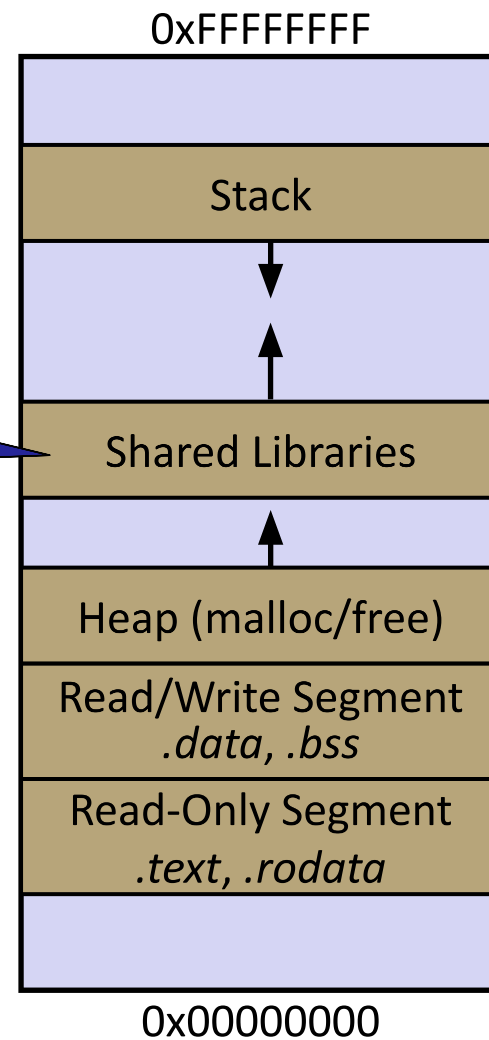
File System
- open(), read(), write(), close(), …

Network Stack
- connect(), listen(), read(), write(), ...

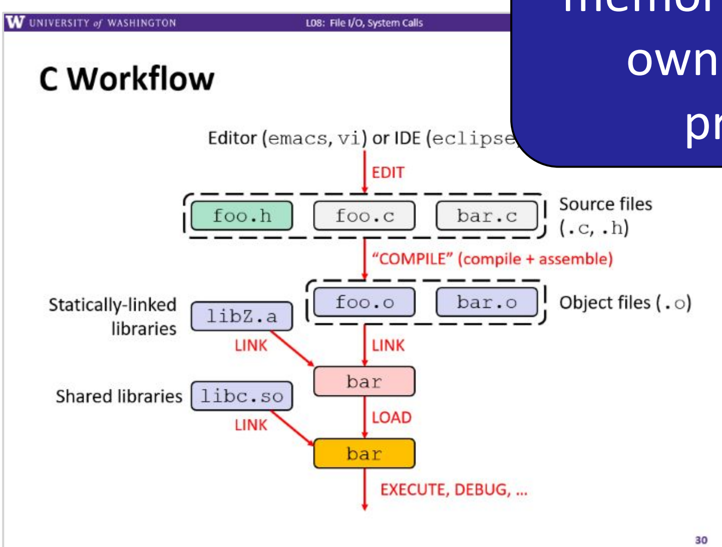
Virtual Memory
- brk(), shm_open(), …

Process Management
- fork(), wait(), nice(), …

- Where does shared code, such as `strcmp()`, live in memory?



But not in physical memory exclusively owned by the process!



To do:

- New exercise (ex6) posted today, due Wednesday morning
- HW1 due on **Thursday @ 11pm**

Extra Exercise #1

- Write a program that:
  - Prompts the user to input a string (use `fgets()`)
    - Assume the string is a sequence of whitespace-separated integers (*e.g.* `"5555 1234 4 5543"`)
  - Converts the string into an array of integers
  - Converts an array of integers into an array of strings
    - Where each element of the string array is the binary representation of the associated integer
  - Prints out the array of strings

Extra Exercise #2

- 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`

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

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