Final C Details, System Calls, and I/O

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Justin Tysdal

Sayuj Shahi

Nicholas Batchelder

Leanna Mi Nguyen

Administrivia

- HW0 grades are posted!
  - Regrade requests can be done on gradescope
  - Questions about your grade can go in private edboard messages.
- New exercise (ex6) posted today, due Wednesday morning
  - There is no exercise 5! We're skipping it this quarter
- HW1 due on **Friday at 11pm**
- No section this week (it's 4th of July)
- Still a lecture on Friday though (5th of July)

Lecture Outline

- **Header Guards and Preprocessor Tricks**
- Visibility of Symbols
  - `extern`, `static`
- File I/O with the C standard library
- System Calls

An `#include` Problem

- What happens when we compile `foo.c`?

```c
struct pair {
  int a, b;
};
```

pair.h

```c
#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);
```

util.h

```c
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
  ...
  return EXIT_SUCCESS;
}
```

foo.c

An `#include` Problem

- What happens when we compile `foo.c`?

```
bash$ gcc -Wall -g -o foo foo.c
In file included from util.h:1:0,
                 from foo.c:2:
pair.h:1:8: error: redefinition of 'struct pair'
 struct pair { int a, b; };
        ^
In file included from foo.c:1:0:
pair.h:1:8: note: originally defined here
 struct pair { int a, b; };
        ^
```

- `foo.c` includes `pair.h` twice!
  - Second time is indirectly via `util.h`
  - Struct definition shows up twice
    - Can see using `cpp`

Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif  // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif  // UTIL_H_
```

util.h

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

```
#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);
```

util.h

```
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
  ...
  return EXIT_SUCCESS;
}
```

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**":

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

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

cpp →

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

- Beware of operator precedence issues!
  - Use parentheses

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

ODD(5 + 1);

WEIRD(5 + 1);
```

cpp →

```c
((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

```
#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;
}
```

Lecture Outline

- Header Guards and Preprocessor Tricks
- **Visibility of Symbols**
  - `extern`, `static`
- File I/O with the C standard library
- System Calls

Headers Aren't Enough

- So far, we've been using header declarations to provide encapsulation (private data/function hiding)
- But code can get around these by re-declaring the variables!
  - The linker will happily link the two variables together
- We need a way to tell the linker which definitions should be only accessible by their module

Namespace Problem

- If we define a global variable named "counter" in one C file, is it visible in a different C file in the same program?
  - Yes, if you use *external linkage* (default)
    - The name "counter" refers to the same variable in both files
    - The variable is *defined* in one file and *declared* in the other(s)
    - When the program is linked, the symbol resolves to one location
  - No, if you use *internal linkage*
    - The name "counter" refers to a different variable in each file
    - The variable must be *defined* in each file
    - When the program is linked, the symbols resolve to two locations

External Linkage

- `extern` makes a *declaration* refer to something externally-visible elsewhere

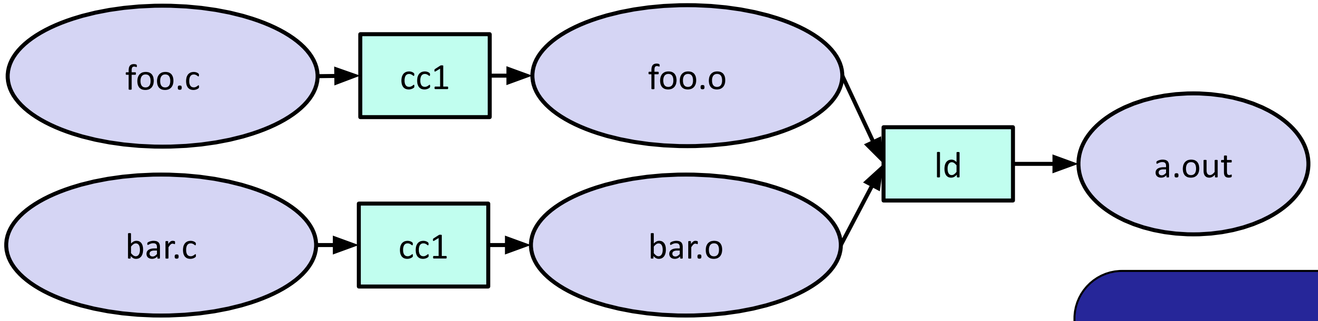
```
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// It has external linkage by
// default.
int counter = 1;

int main(int argc, char** argv) {
  printf("%d\n", counter);
  bar();
  printf("%d\n", counter);
  return EXIT_SUCCESS;
}
```

foo.c

```
#include <stdio.h>

// "counter" is defined and
// initialized in foo.c.
// Here, we declare it, and
// specify external linkage
// by using the extern specifier.
extern int counter;

void bar() {
  counter++;
  printf("(b): counter = %d\n",
         counter);
}
```

bar.c

External Linkage




```c
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// It has external linkage by
// default.
int counter = 1;

int main(int argc, char** argv) {
  printf("%d\n", counter);
  bar();
  printf("%d\n", counter);
  return EXIT_SUCCESS;
}
```

foo.c

```c
#include <stdio

// "counter" i
// initialized in
// Here, we de    re it, and
// specify external linkage
// by using the extern specifier.
extern int counter;

void bar() {
  counter++;
  printf("(b): counter = %d\n",
         counter);
}
```

If you forget to add this, you'll get a linker error!

bar.c

Closing Thoughts on Data Visibility

- Don't do either of these with an unchanged NthPrime.c!

```
#define NUM_PRECALC 20
static int16_t
kPrecalculated[NUM_PRECALC] =
    {2, 3, 5, 7, /* etc */ };

int16_t NthPrime(int16_t n);
```

NthPrime.static.h

```
extern int16_t kPrecalculated[];



int16_t NthPrime(int16_t n);
```

NthPrime.extern.h

- Static variable in the header: every file that #includes the header will have its own private copy of the variable
  - Unnecessary data duplication!
- Extern variable in the header: the accompanying .c file must define the extern'ed variable for successful linkage

Function Visibility

bar.c

```
// By using the static specifier, we are indicating
// that foo() should have internal linkage.  Other
// .c files cannot see or invoke foo().
static int foo(int x) {
  return x*3 + 1;
}

// Bar is "extern" by default.  Thus, other .c files
// could declare our bar() and invoke it.
int bar(int x) {
  return 2*foo(x);
}
```

main.c

```
#include <stdio.h>

extern int bar(int x); // "extern" is default, usually omit.
                       // should be in .h file, but effect is same

int main(int argc, char** argv) {
  printf("%d\n", bar(5));
  return EXIT_SUCCESS;
}
```

Linkage Issues

- Every global (variables and functions) is `extern` by default
  - Unless you add the `static` specifier, if some other module uses the same name, you'll end up with a collision!
    - Best case: compiler (or linker) error
    - Worst case: stomp all over each other
- It's good practice to:
  - Use `static` to "defend" your globals
    - Hide your private stuff!
  - Place external declarations in a module's header file
    - Header is the public specification

Static Confusion…

- C has a *different* use for the word "`static`": to create a persistent *local* variable
  - The storage for that variable is allocated when the program loads, in either the `.data` or `.bss` segment
  - Retains its value across multiple function invocations
  - Confusing! Don't use!! (But you may see it ☹)

```
void foo() {
  static int count = 1;
  printf("foo has been called %d times\n", count++);
}

void bar() {
  int count = 1;
  printf("bar has been called %d times\n", count++);
}

int main(int argc, char** argv) {
  foo(); foo(); bar(); bar(); return EXIT_SUCCESS;
}
```

static_extent.c

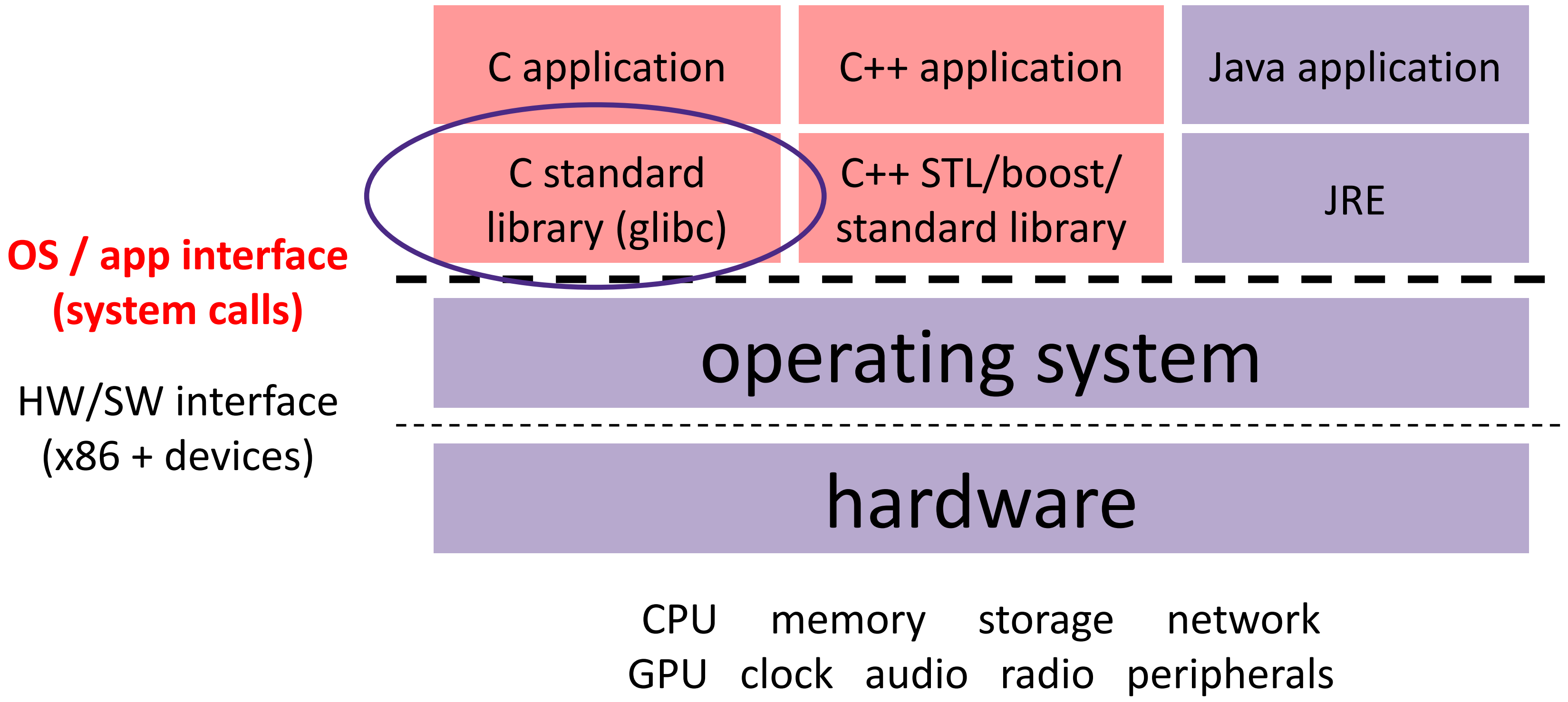
Additional C Topics

- Teach yourself!
  - **man pages** are your friend!
  - 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()
  - How to declare, define, and use a function that accepts a variable-number of arguments (`varargs`)
  - `union`s and what they are good for
  - `enum`s and what they are good for
  - Pre- and post-increment/decrement
  - Harder: the meaning of the "`volatile`" storage class

Lecture Outline

- Header Guards and Preprocessor Tricks
- Visibility of Symbols
  - `extern`, `static`
- **File I/O with the C standard library**
- System Calls

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
  - 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)
  - ```
    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*

C Stream Functions

- Formatted I/O stream functions (more in in `stdio.h`):
  - `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

Error Checking/Handling

- Some error functions (complete list in `stdio.h`):
  - `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
  - `void perror(message);`
    - Prints `message` and error message related to `errno` to `stderr`

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

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

}
```

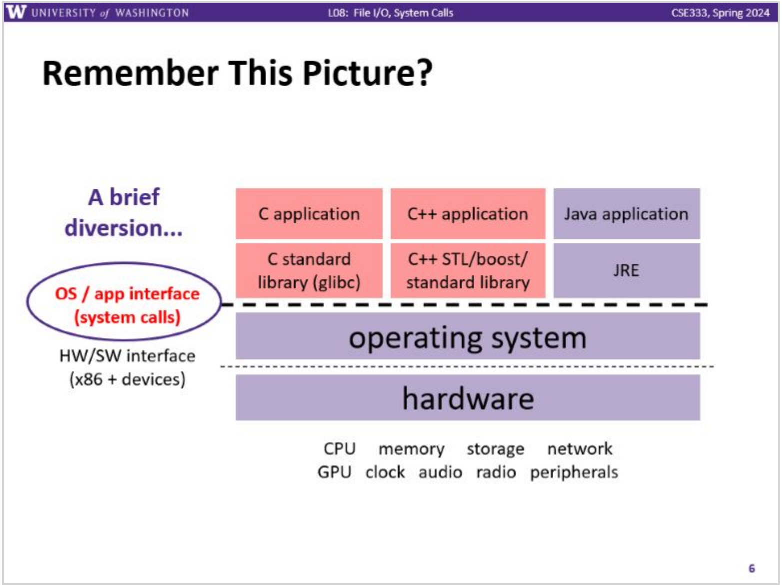
C Streams Example

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"*)
- When is buffering faster? Slower?

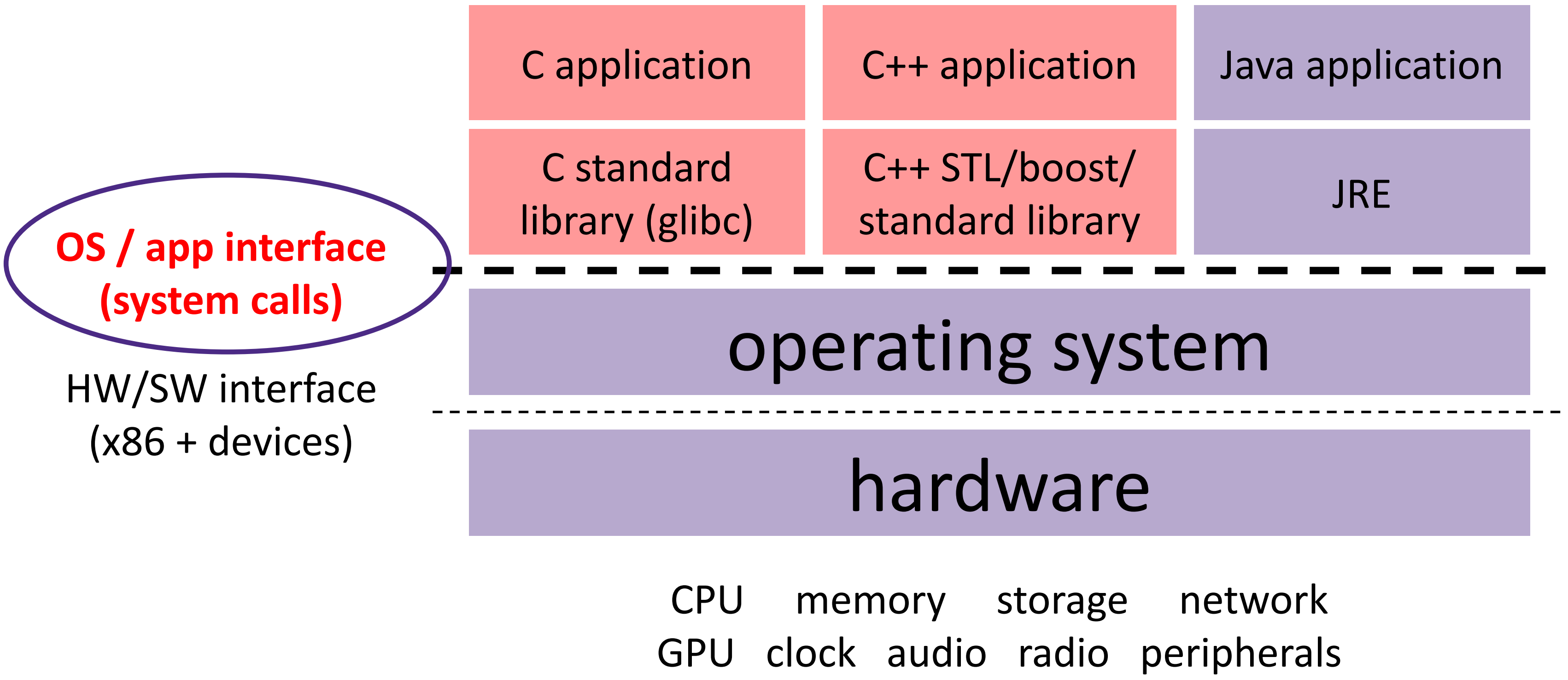
Disabling C's Buffering

- 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
- But… what about the layers below?
  - The OS caches disk reads and writes in the file system *buffer* cache
  - Disk controllers have caches too!

Lecture Outline

- Header Guards and Preprocessor Tricks
- Visibility of Symbols
  - `extern`, `static`
- File I/O with the C standard library
- **System Calls**

What's an OS?

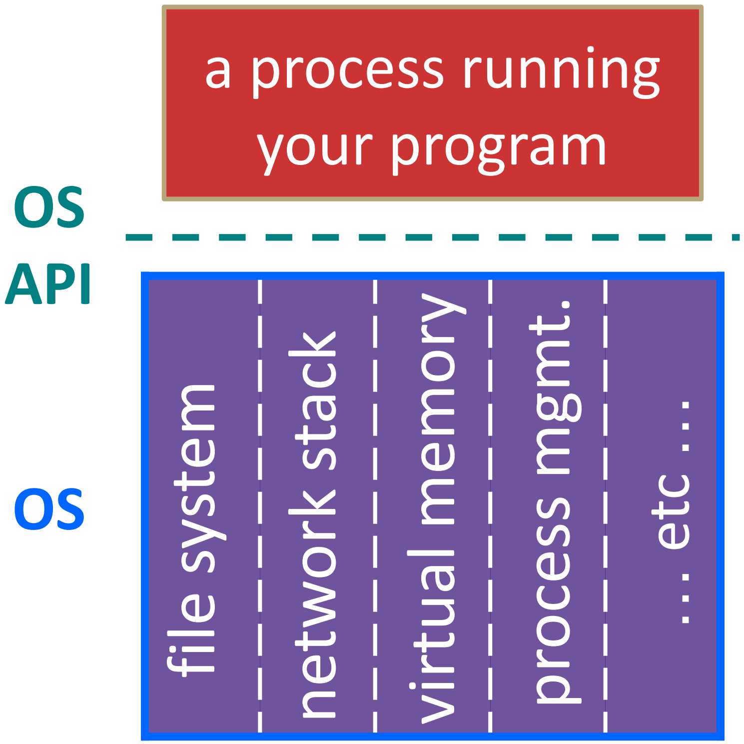

C application
C++ application
Java application
C standard library (glibc)
C++ STL/boost/ standard library
JRE
OS / app interface (system calls)
operating system
HW/SW interface (x86 + devices)
hardware
CPU memory storage network
GPU clock audio radio peripherals

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

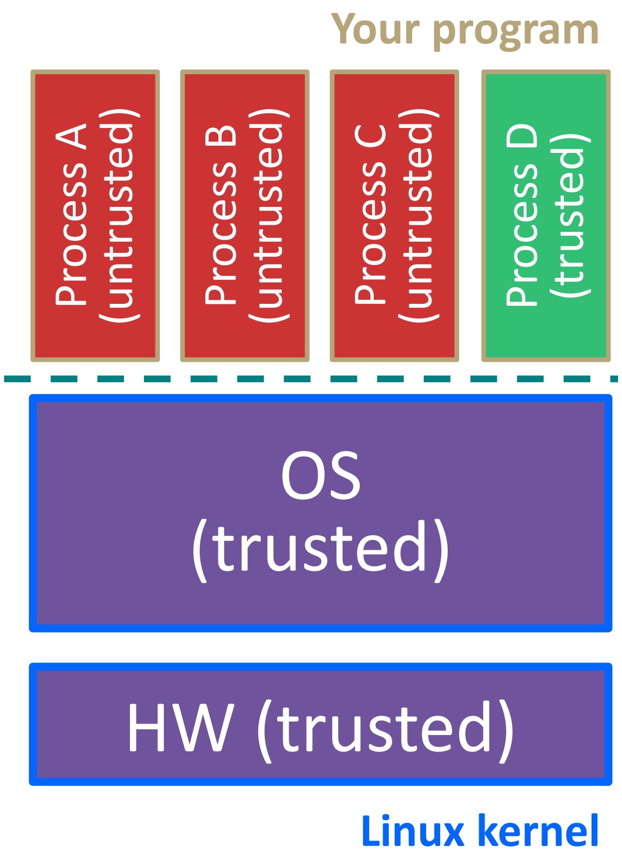
- brk(), shm_open(), …

Process Management

- fork(), wait(), nice(), …

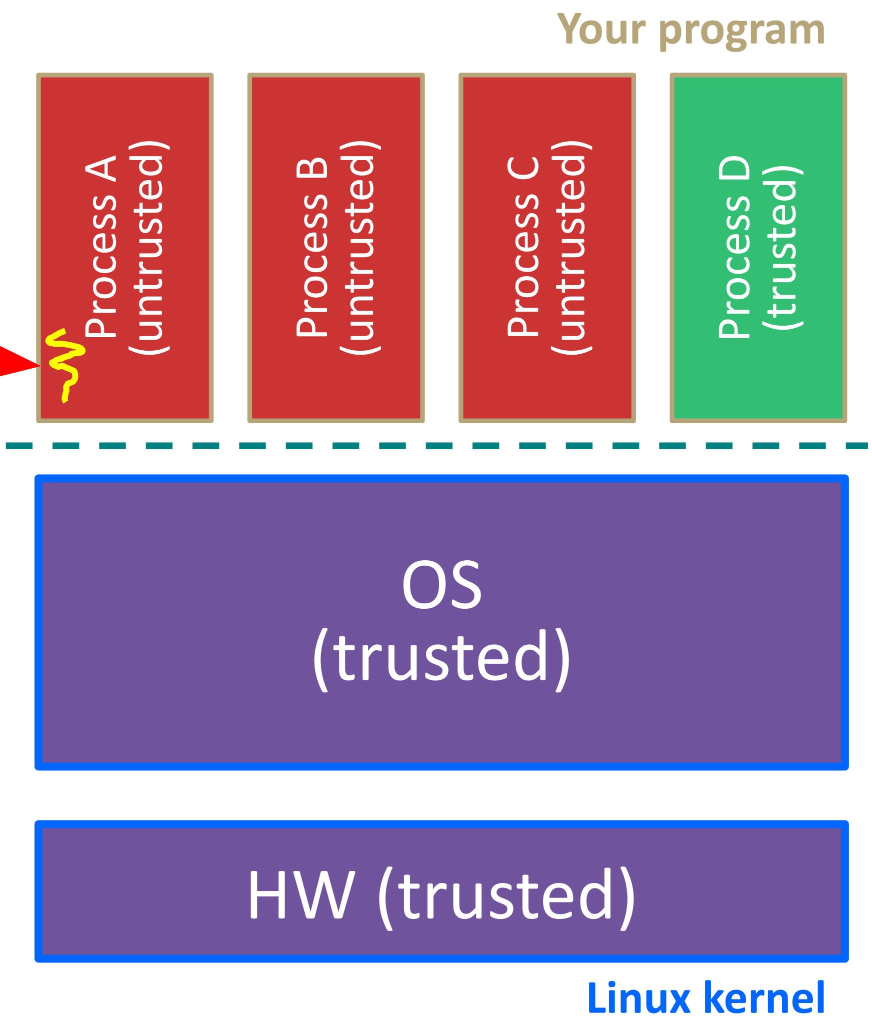
OS: Protection System

- OS isolates process from each other
  - But permits controlled sharing between them
    - Through shared name spaces (*e.g.* file names)
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly
- OS is allowed to access the hardware
  - User-level processes run with the CPU (processor) in unprivileged mode
  - The OS runs with the CPU in privileged mode
  - User-level processes invoke system calls to safely enter the OS



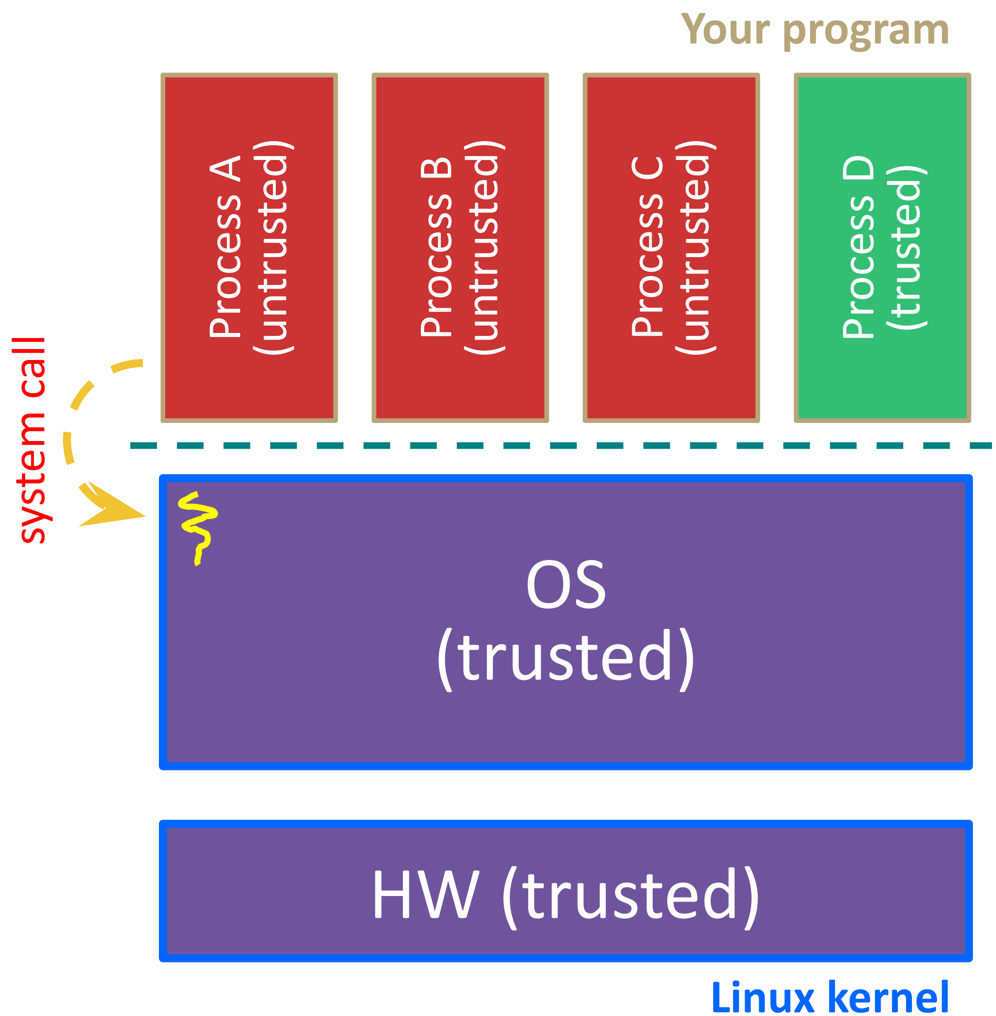
System Call Trace

A CPU (thread of execution) is running user-level code in Process A; the CPU is set to *unprivileged mode*.



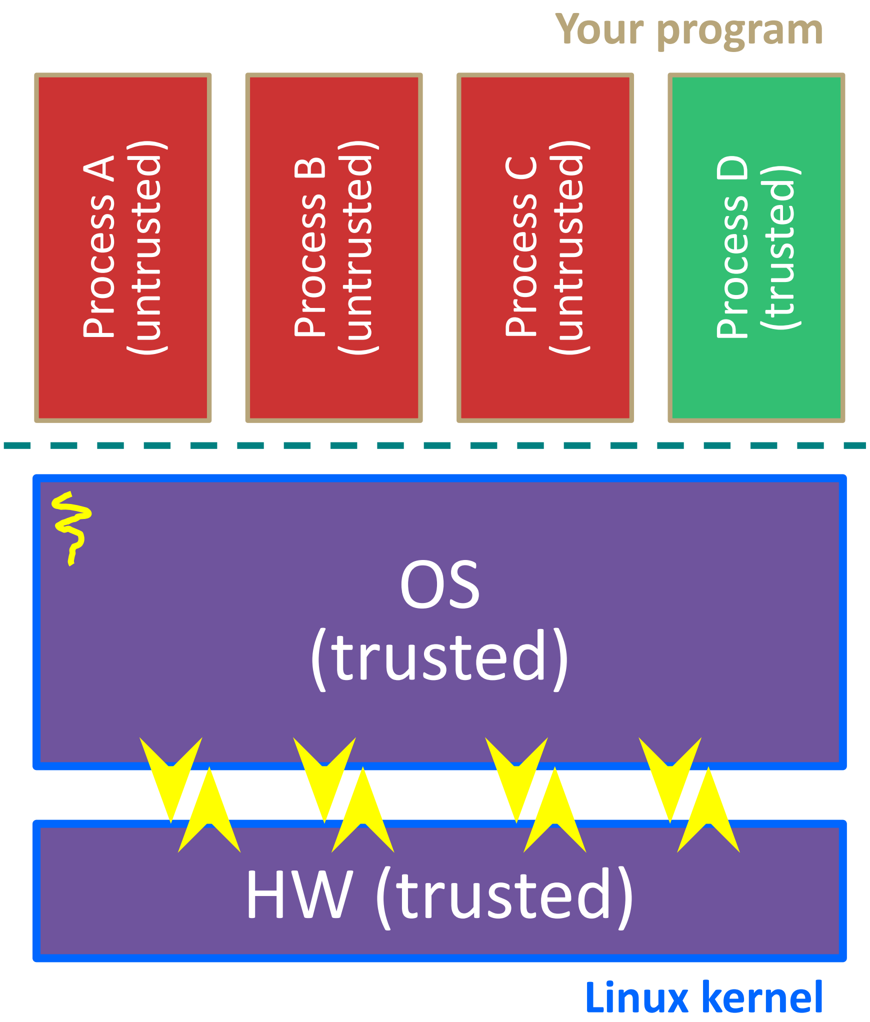
System Call Trace

Code in Process A invokes a system call; the hardware then sets the CPU to *privileged mode* and traps into the OS, which invokes the appropriate system call handler.



System Call Trace

Because the CPU executing the thread that's in the OS is in privileged mode, it is able to use *privileged instructions* that interact directly with hardware devices like disks.

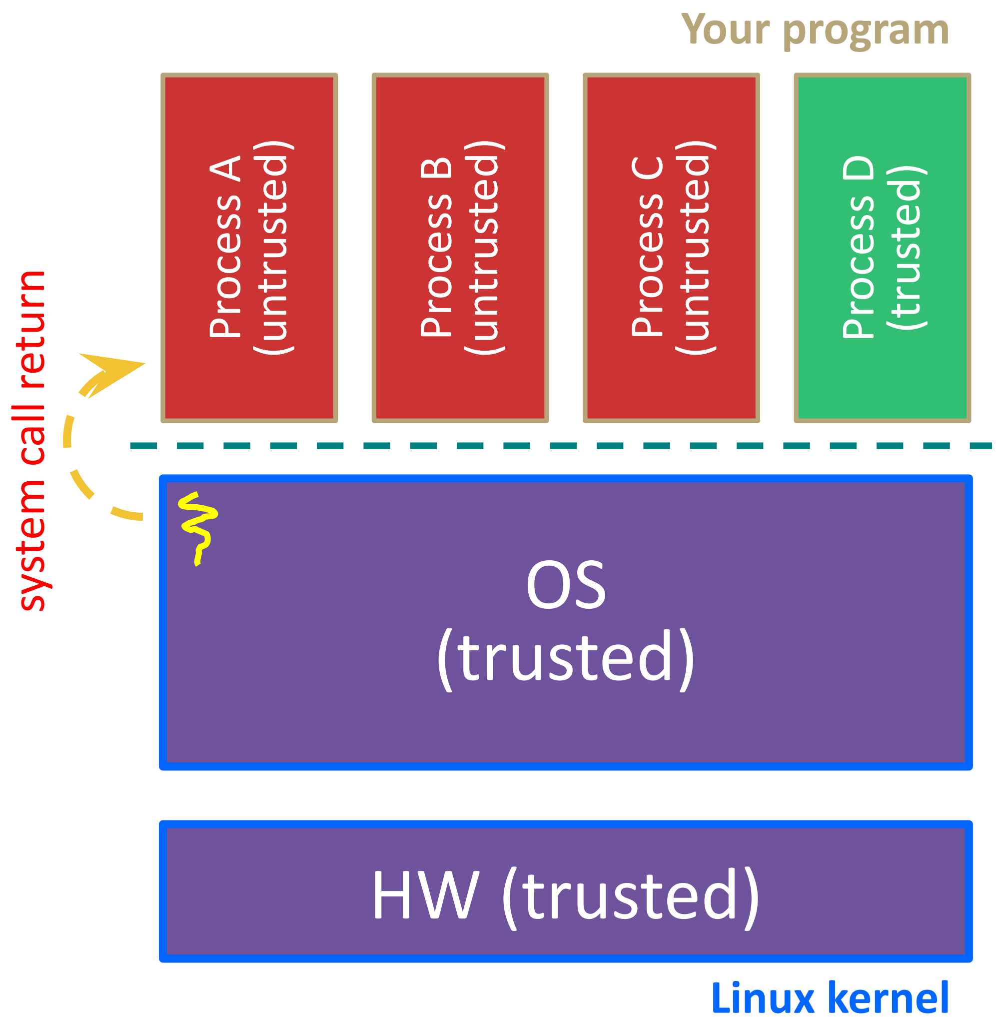


System Call Trace

Once the OS has finished servicing the system call, which might involve long waits as it interacts with HW, it:

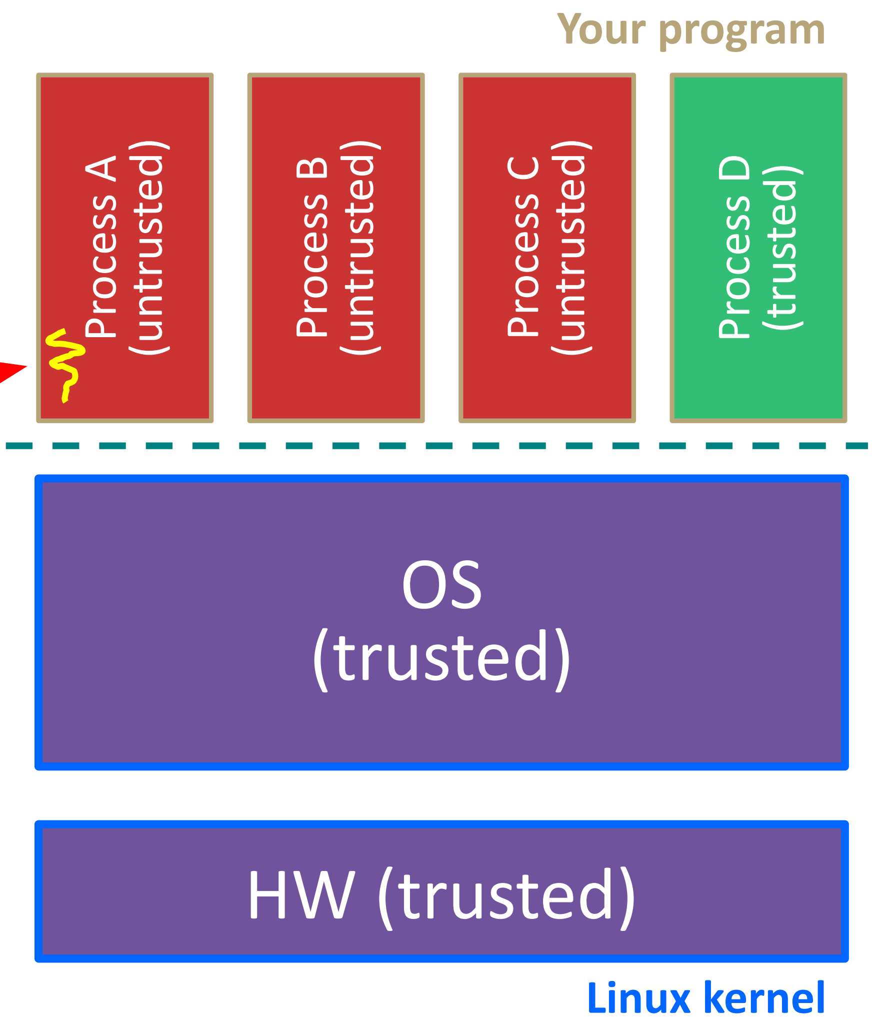
(1) Sets the CPU back to unprivileged mode and

(2) Returns out of the system call back to the user-level code in Process A.



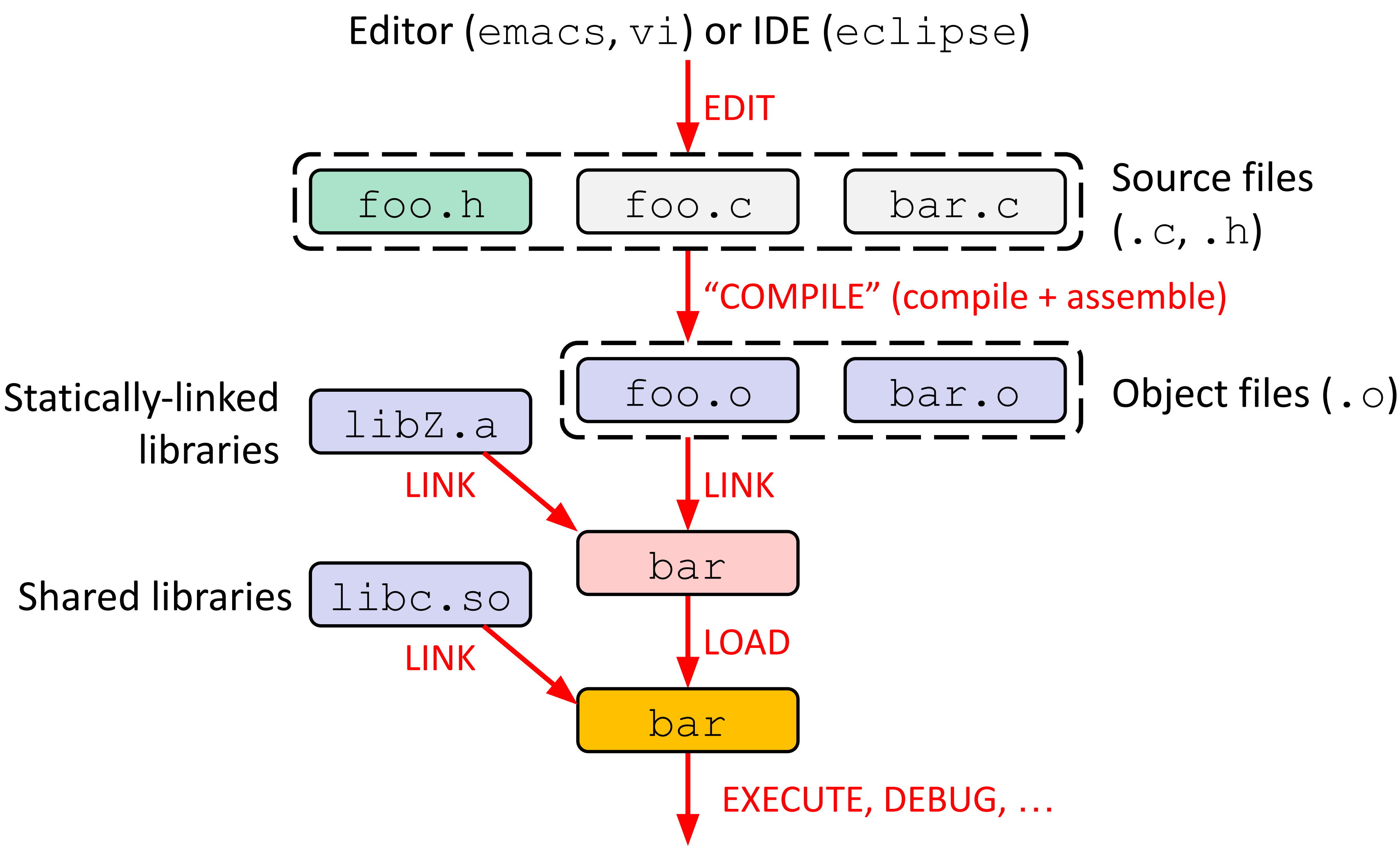
System Call Trace

The process continues executing whatever code is next after the system call invocation.



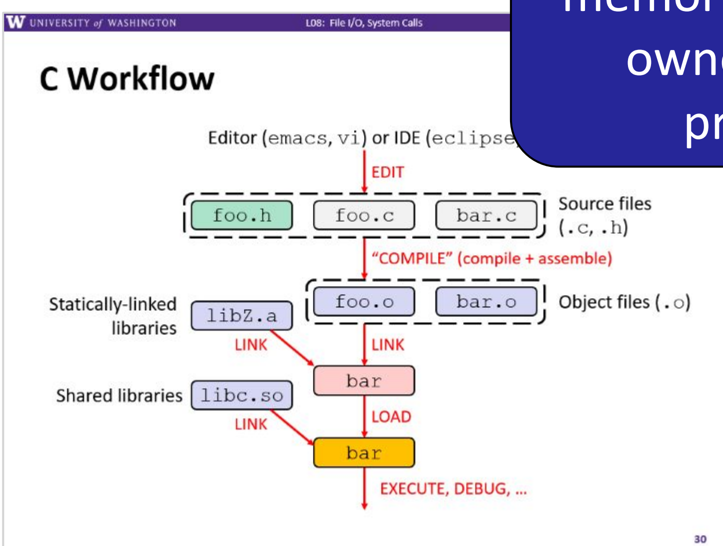
Useful reference:
CSPP § 8.1–8.3
(the 351 book)

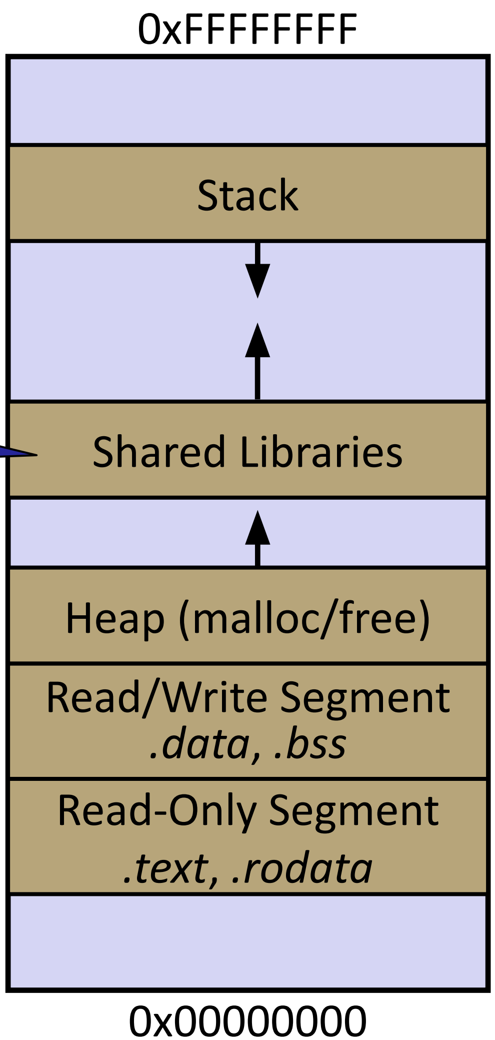
C Workflow


Editor (emacs, vi) or IDE (eclipse)
EDIT
foo.h
foo.c
bar.c
Source files
(.c, .h)
"COMPILE" (compile + assemble)
Statically-linked
libraries
libZ.a
foo.o
bar.o
Object files (.o)
LINK
LINK
bar
Shared libraries
libc.so
LOAD
LINK
bar
EXECUTE, DEBUG, …

- 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
  - There is no exercise 5! We're skipping it this quarter
- HW1 due on **Friday at 11pm**
- Bring your laptop to class on Wednesday! We're going to be doing some in-class exercises

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`

- 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$
```