UNIVERSITY of WASHINGTON

LECTURE 7 · CSE 333 · Summer 2026

File I/O, POSIX, System Calls

Discussion: What incentives do students have to cheat? In what ways are they reasonable?

Reminders

Assessments:

- Exercise 3 due tonight at 11:59pm
- Homework 1 due tomorrow at 11:59pm
  - VS Code Debugging tutorial might be helpful

General:

- Sign up for a coffee chat!
- Exams are scheduled
  - Midterm (7/27) from 9:40am to 10:40am in IEB G106
  - Final exam (8/18) from 1:30pm to 3:30pm in CSE2 G10

Cheating

Review

The pattern: #ifndef / #define / #endif

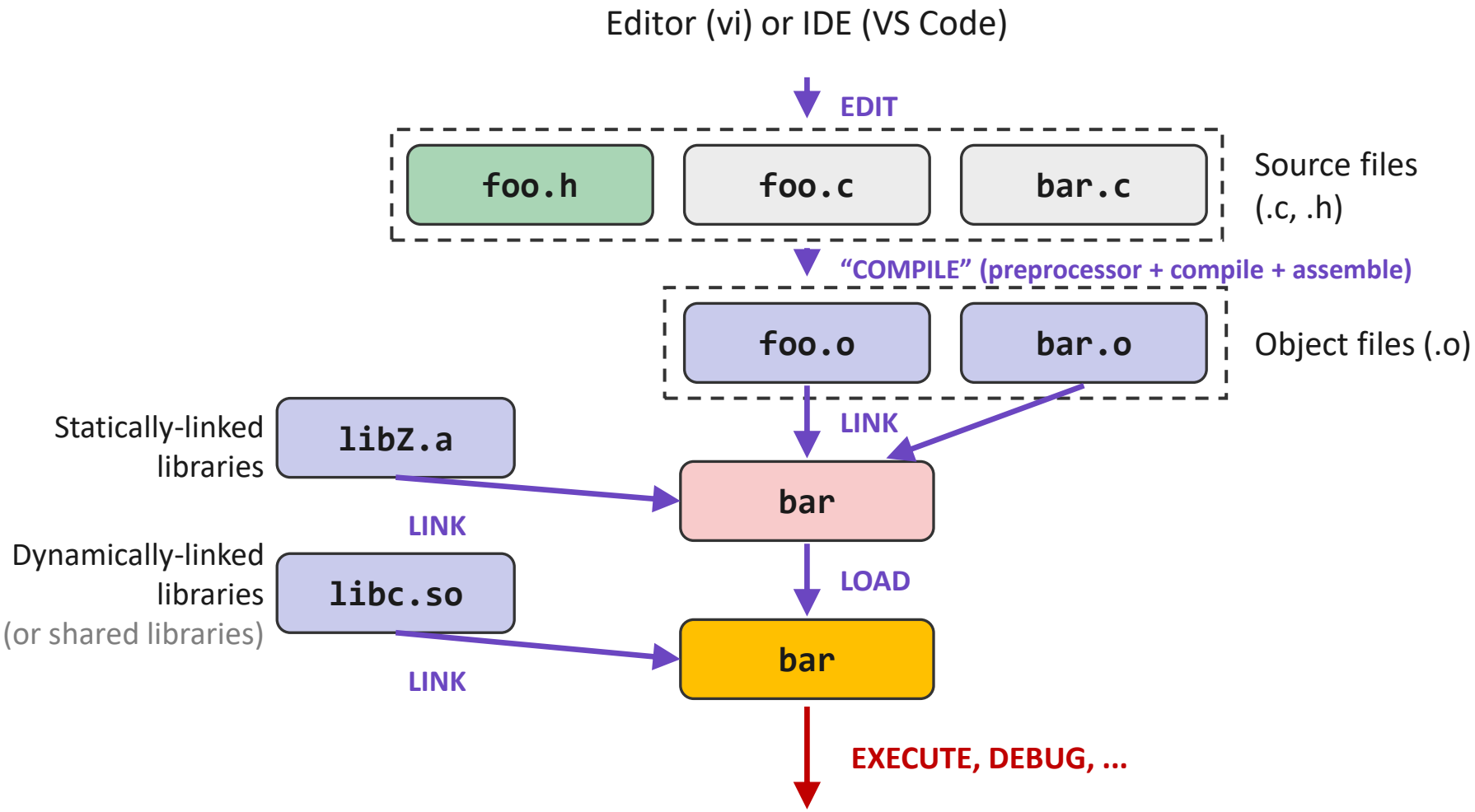
date.h (guarded)

```
1  #ifndef DATE_H_
2  #define DATE_H_
3  struct Date {
4    int y, m, d;
5  };
6  #endif  // DATE_H_
```

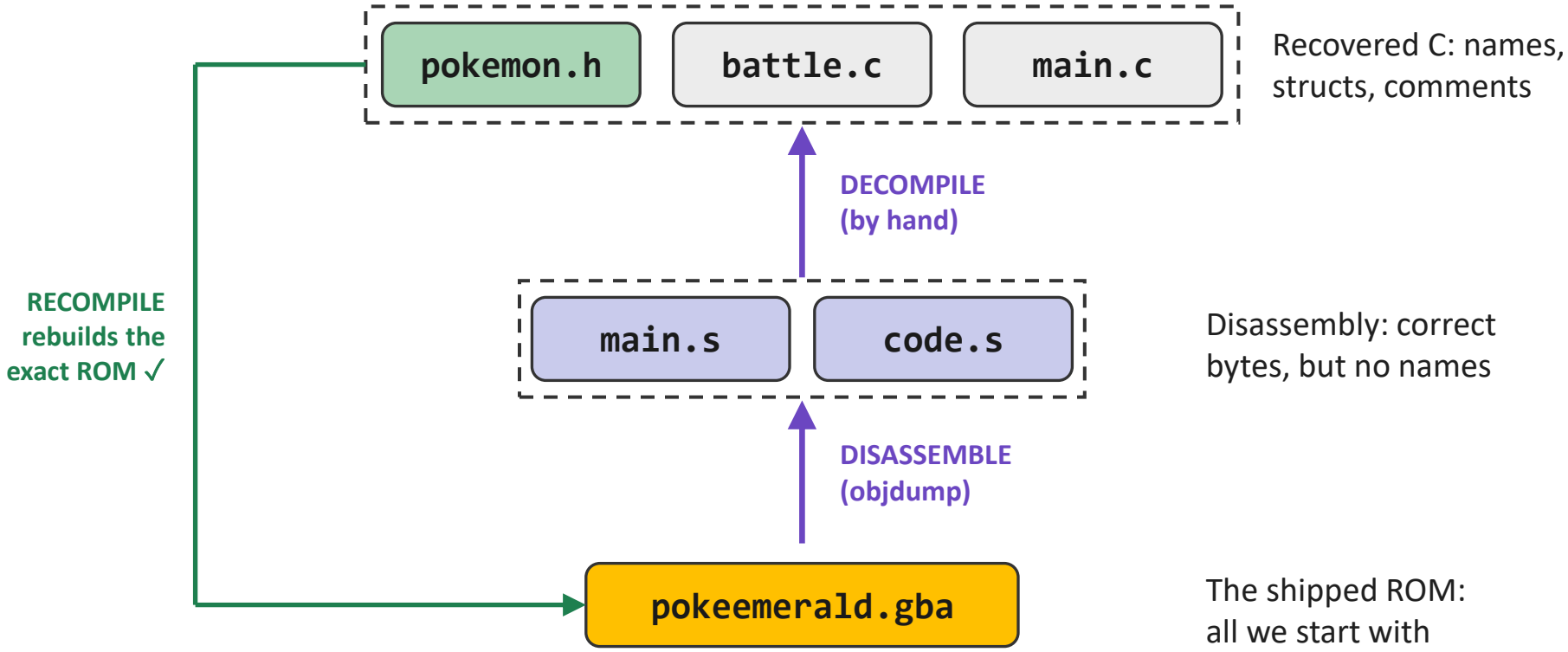
Read it as: "if DATE_H_ has not been defined, keep everything down to #endif; otherwise, delete all of it."

- **#ifndef DATE_H_**
  - "if not defined." The whole block is kept only when DATE_H_ is not yet a macro.
  - The name DATE_H_ is arbitrary,
- **#define DATE_H_**
  - leave the note. Recall a #define just names some text; here the text is empty, so DATE_H_ is only a flag.
- **...the real contents...**
  - your struct, prototypes, and #defines go here.
- **#endif**
  - closes the #ifndef block.

Compilation



Decompilation



Takeaway: Decompilation is the just the compilation pipeline in reverse.

Doing it right, every time

- **Guard every header.** Wrap the entire contents of every .h in the three lines. Make it a habit, not a decision.
- **Name it after the file.** A common convention: the filename in caps with a trailing underscore.
- **Make the name unique.** If two headers share a guard name, the second one gets silently skipped. That bug is nasty to find.
- **It also stops infinite include loops** when two headers include each other.
- **#pragma once** does the same thing in one line. It is widely supported but *technically* not standard C

the standard shape of any .h

```
1  #ifndef DATE_H_
2  #define DATE_H_
3
4  // ... all declarations ...
5
6  #endif  // DATE_H_
```

or, non-standard shorthand (probably fine?)

```
1  #pragma once
2  // ... all declarations ...
```

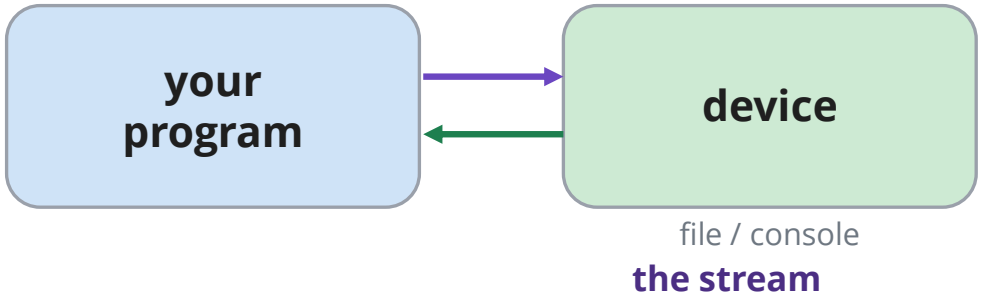
Streams

❖ **A stream is a sequence of bytes** flowing to or from a device.

❖ The device can be anything: a file, the console, or a pipe. Same idea for all of them.

❖ Text or binary: it is all just bytes; Linux does not distinguish.

❖ **You do not read the whole file at once.** Bytes flow through the stream a chunk at a time.



Three streams you have already used

`stdin`	**standard input** keyboard by default	`scanf reads it`
`stdout`	**standard output** the console by default	`printf writes it`
`stderr`	**standard error** the console, for messages	`fprintf(stderr,...) writes`

You never opened these: **they are already open when your program starts.**

fopen: open a file, get a stream

```
1  FILE* fopen(const char* filename, const char* mode);
```

```
1  FILE* fp = fopen("notes.txt", "r");
```

- **filename** is the path to the file you want.
- **mode** says what you intend to do: read, write, or append (next slide).
- Returns a **FILE*** you use for everything after, or **NULL** if it could not open.

Access modes: what you plan to do

`"r"`	**read**	open an existing file for reading
`"w"`	**write**	create/truncate a file for writing
`"a"`	**append**	write, but add onto the end
`"r+"`	**read + write**	open existing for both

Add a **b** for binary, like **"rb"** or **"wb"**. On Linux it changes nothing (bytes are bytes), but it is good habit and matters on Windows.

File IO

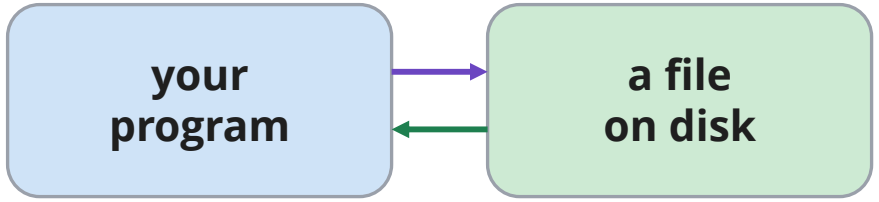
Reading and writing files

❖ **So far, data lived in memory** and vanished when the program ended.

❖ Real programs persist data: save a file, load a config, read input, write a log.

❖ **The C standard library** gives us a simple, portable way to do this.

❖ **Our plan for today:** open a file, write to it, read from it, close it, one function at a time.



read and write, in both directions

The four operations

❖ **Working with a file is like using a physical notebook.** You do the same four things every time:

❖ **Open** (fopen): find the notebook and flip it open.

❖ **Write** (fprintf / fwrite): write bytes where the pen sits.

❖ **Read** (fread): scan bytes back, from where you left off.

❖ **Close** (fclose): shut it so nothing is lost.

open	`fopen`
write	`fprintf / fwrite`
read	`fread`
close	`fclose`

The FILE* handle

- A **FILE*** is your handle to a stream.
- **You do not look inside it.** It bundles the details (the buffer, the position, error flags) that libc manages for you.
- You pass the FILE* to every stream function so it knows which stream you mean.
- Defined in **stdio.h**. stdin, stdout, stderr are FILE*s too.

```
1  #include <stdio.h>
2  FILE* fp;   // a stream handle
```

Think of it like a ticket: fopen hands you a FILE*, you show that ticket to fread / fwrite / fclose, and they act on the right stream. You rarely care what is printed on the ticket.

Opening a file that is not there

```
1  FILE* fp = fopen("nope.txt", "r");
2
3  // fp is now ... what?
```

- nope.txt does not exist, so fopen could not open a stream.
- So what sits in **fp**, **and what happens the moment you use it?**

fopen can fail: always check for NULL

```
1  FILE* fp = fopen("notes.txt", "r");
2  if (fp == NULL) {
3    fprintf(stderr, "cannot open file\n");
4    return EXIT_FAILURE;
5  }
```

- The file may not exist, or you may lack permission.
- On failure fopen returns **NULL**.
- **fprintf(stderr, ...)** writes your message to the error stream.
- **Check every fopen.** Using a NULL FILE* later crashes.

fclose: always close what you open

```
1  int fclose(FILE* stream);
```

- **Every fopen deserves an fclose.** You are done with the stream; hand it back.
- **Closing flushes buffered writes** to the file (more on buffering later), so the last bytes actually land.
- It frees resources. Open files are limited; leaking them is a real bug.
- Returns 0 on success, **EOF** on error.

The smallest complete program

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  FILE* fp = fopen("notes.txt", "w");
  if (fp == NULL) {
    fprintf(stderr, "cannot open\n");
    return EXIT_FAILURE;
  }
  fclose(fp);
  return EXIT_SUCCESS;
}
```

❖ **Open, check, close.** That is the whole skeleton.

❖ This really does something: mode "w" creates an empty notes.txt.

❖ **Everything else** goes between the open and the close.

Writing text: fprintf

```
1  int fprintf(FILE* stream, const char* fmt, ...);
```

```
1  fprintf(fp, "Score: %d\n", 42);
2  fprintf(stderr, "Your program has crashed!\n");
```

- **It is just printf with a stream in front.** Same format string, same %-conversions.
- **In fact: printf(...)** is exactly **fprintf(stdout, ...)**.
- Great for human-readable output: logs, reports, CSV, config.

An fprintf example

```
1  FILE* fp = fopen("log.txt", "w");
2  fprintf(fp, "hello\n");
3  fprintf(fp, "x = %d\n", 7);
4  fclose(fp);
```

produces

log.txt

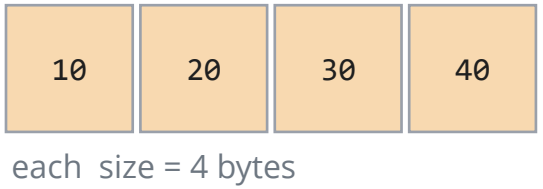
```
hello
x = 7
```

Each fprintf appends where the last one left off. **The stream remembers your position, so line two follows line one, with no need to say where.**

Writing raw bytes: fwrite

```
1  size_t fwrite(ptr, size, count, stream);
```

```
1  int a[4] = {10, 20, 30, 40};
2  fwrite(a, sizeof(int), 4, fp);
```



- Writes **count** items of **size** bytes each: here 4 ints, 16 bytes total.
- Use it for binary data: arrays, structs, whole buffers, in one call.
- Returns the number of items written (should be count; less means trouble).

Reading raw bytes: fread

```
size_t fread(ptr, size, count, stream);
```

```
int a[4];
size_t got = fread(a, sizeof(int), 4, fp);
```

- **The exact mirror of fwrite:** same ptr, size, count, stream.
- Copies bytes from the stream into your buffer at ptr.
- **Returns how many items it actually read.** Fewer than count means end-of-file (or an error), so always check **got**.

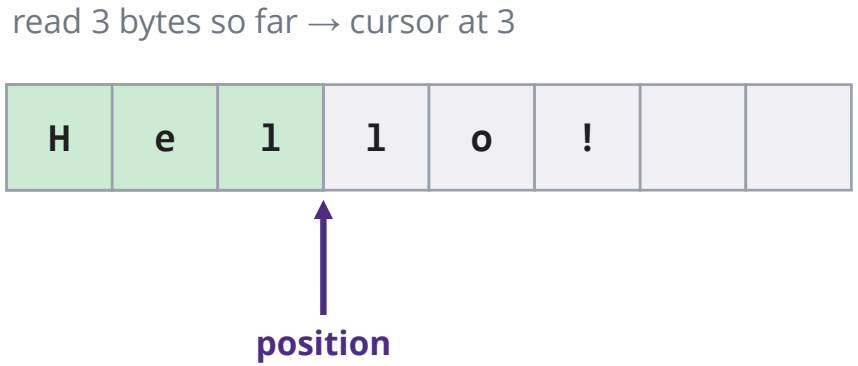
The file position

❖ **Every open stream has a position,** a cursor into the file.

❖ Read or write, and it advances by that many bytes, automatically.

❖ **So repeated calls march forward:** read the first chunk, then the next, then the next.

❖ That is why a loop works: each fread picks up exactly where the last one stopped.



Calling fread twice

```
1  fread(a, sizeof(int), 4, fp);   // first four ints
2  fread(a, sizeof(int), 4, fp);   // ... and now?
```

- **Same call, run twice.** The second fread does NOT re-read the same four ints. It reads the next four.

Return values and errors

- **Every call reports success** through its return value: **NULL** from fopen, a short count from fread/fwrite.
- **errno** is a global the library sets to say what went wrong.
- **fprintf(stderr, ...)** sends your error message to the error stream, so it stays separate from normal output.
- **Check as you go.** Do not wait until the end to discover a write failed.

```
int ferror(stream);
```

did an error happen on this stream?

```
void clearerr(stream);
```

reset the error / EOF flags

```
fprintf(stderr, fmt, ...);
```

print an error message

error output (to stderr):

```
cannot open file
```

ferror and clearerr

```c
#include <stdio.h>
int  ferror(FILE* stream);
void clearerr(FILE* stream);
```

```c
size_t n = fread(buf, 1, SIZE, fp);
if (n < SIZE && ferror(fp)) {
  fprintf(stderr, "read error\n");
  clearerr(fp);   // reset error/EOF flags
}
```

❖ **A short count is ambiguous.** Did fread stop at end-of-file, or hit a real error?

❖ **ferror(fp)** tells them apart: nonzero means the stream's error flag is set.

❖ **clearerr(fp)** clears the error and EOF flags so the stream is usable again.

❖ Undefined behavior on failure!

Step 1 of 4

Example: copying a file

cp_example.c (the core loop)

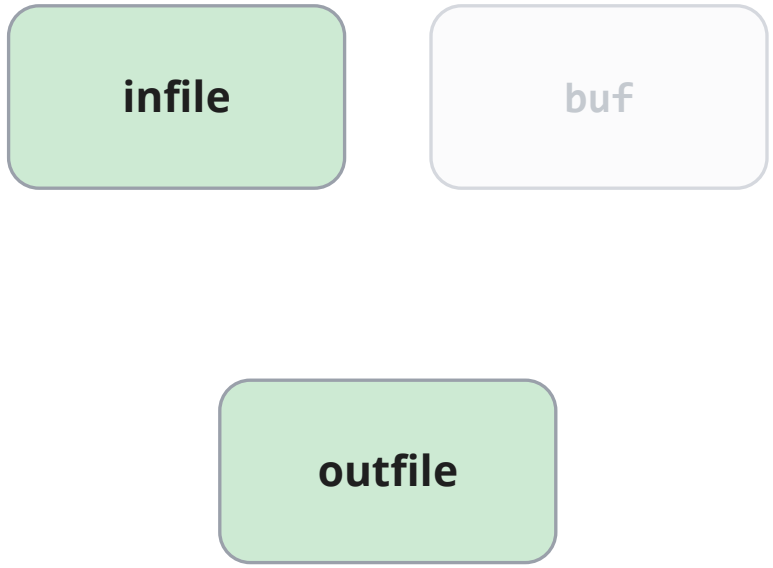
```
1  FILE* fin = fopen(argv[1], "rb");
2  if (fin == NULL) { return EXIT_FAILURE; }
3
4
5
6
7
8
9
10
11
12
13
```

Open the input for reading. If fopen returns **NULL**, stop right away with **EXIT_FAILURE**.

Example: copying a file (oops!)

cp_example.c (the core loop)

```
1  FILE* fin = fopen(argv[1], "rb");
2  if (fin == NULL) { return EXIT_FAILURE; }
3  FILE* fout = fopen(argv[2], "wb");
4  if (fout == NULL) { return EXIT_FAILURE; }
5
6
7
8
9
10
11
12
13
```

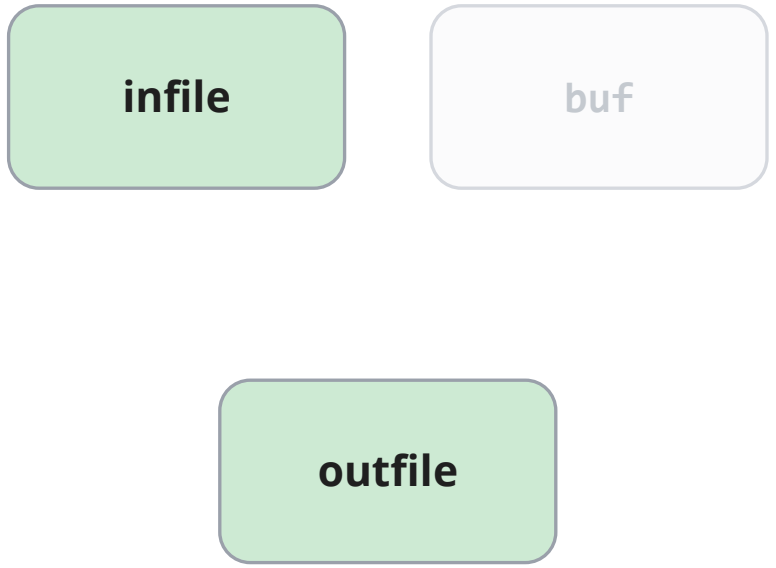


Open the output for writing. What's wrong with this picture

Example: copying a file

cp_example.c (the core loop)

```
1  FILE* fin = fopen(argv[1], "rb");
2  if (fin == NULL) { return EXIT_FAILURE; }
3  FILE* fout = fopen(argv[2], "wb");
4  if (fout == NULL) {
5    fclose(fin);
6    return EXIT_FAILURE;
7  }
8
9
10
11
12
13
```

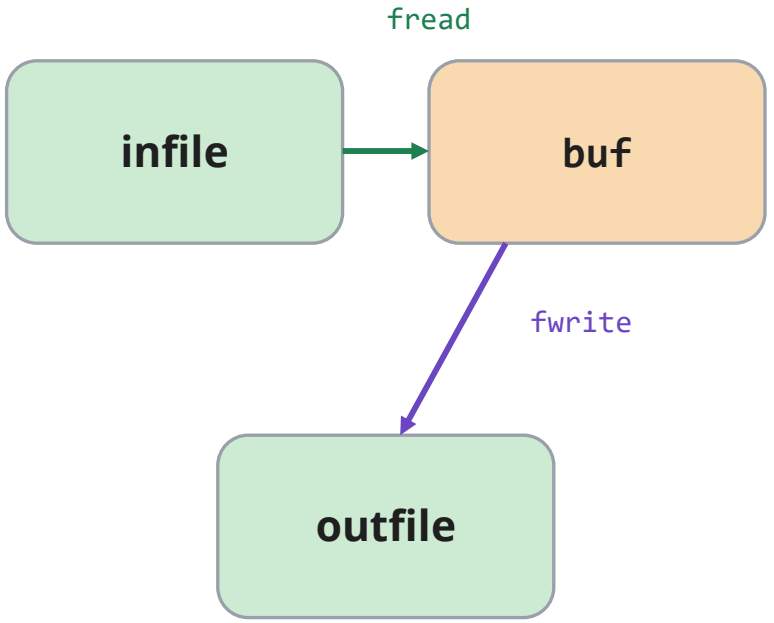


Open the output for writing. Same check, but now we also **fclose(fin)** before leaving, so we do not leak the stream we already opened.

Example: copying a file

cp_example.c (the core loop)

```
1  FILE* fin = fopen(argv[1], "rb");
2  if (fin == NULL) { return EXIT_FAILURE; }
3  FILE* fout = fopen(argv[2], "wb");
4  if (fout == NULL) {
5    fclose(fin);
6    return EXIT_FAILURE;
7  }
8  while ((n = fread(buf, 1, SIZE, fin)) > 0) {
9    fwrite(buf, 1, n, fout);
10 }
11
12
13
```

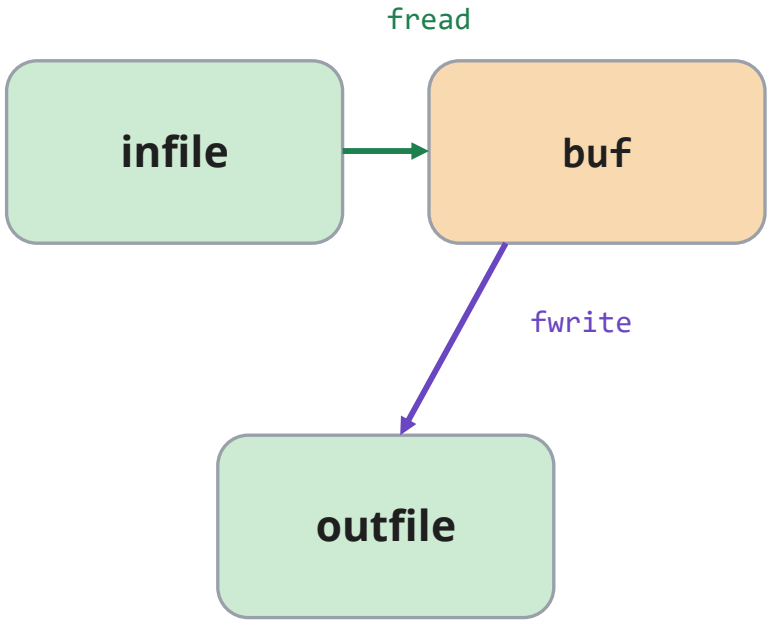


The copy loop: **fread** a chunk, then **fwrite** exactly the **n** bytes we got. Repeat until fread returns 0 (end of file).

Example: copying a file

cp_example.c (the core loop)

```
1  FILE* fin = fopen(argv[1], "rb");
2  if (fin == NULL) { return EXIT_FAILURE; }
3  FILE* fout = fopen(argv[2], "wb");
4  if (fout == NULL) {
5    fclose(fin);
6    return EXIT_FAILURE;
7  }
8  while ((n = fread(buf, 1, SIZE, fin)) > 0) {
9    fwrite(buf, 1, n, fout);
10 }
11 fclose(fin);
12 fclose(fout);
13 return EXIT_SUCCESS;
```

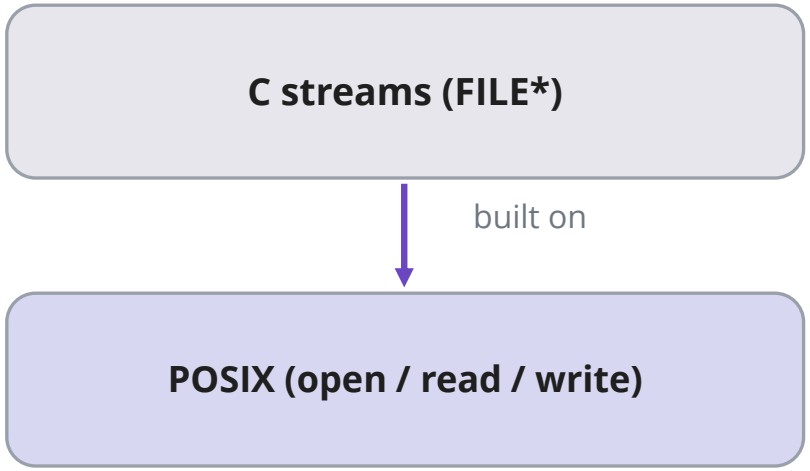


Close both streams, then **return EXIT_SUCCESS**. **fclose** flushes buffered output so the final bytes truly reach the file.

POSIX

Why go below the C stream library

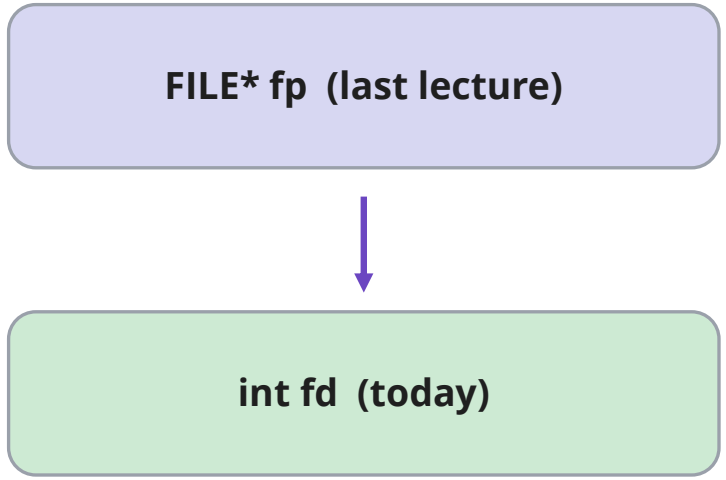
- **You already have fopen, fread, fwrite.** Why learn another way to read and write?
- Streams cannot do everything. They cannot list a directory or open a network socket.
- Sometimes you need control the buffer hides: deciding exactly when bytes leave your program.
- **POSIX is the raw layer the OS provides.** The C streams are built on top of it.



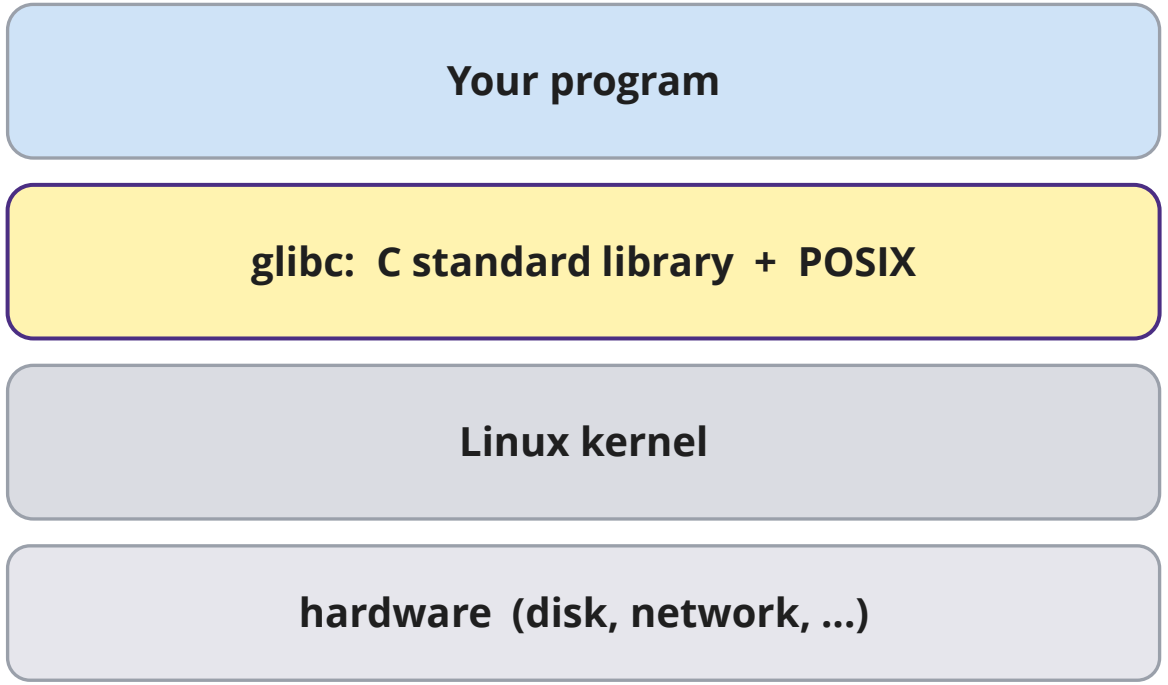
directories and sockets: only POSIX reaches them

The same operations, lower level

- **You already know this story.** Last lecture: fopen, fread, fwrite, fclose on a FILE*.
- **Today is the same four jobs,** just the raw versions the OS provides: open, read, write, close.
- **The handle changes.** Instead of a FILE*, you hold a plain int called a file descriptor.
- You handle two things yourself now: there is no buffer, and a read or write can come up short.

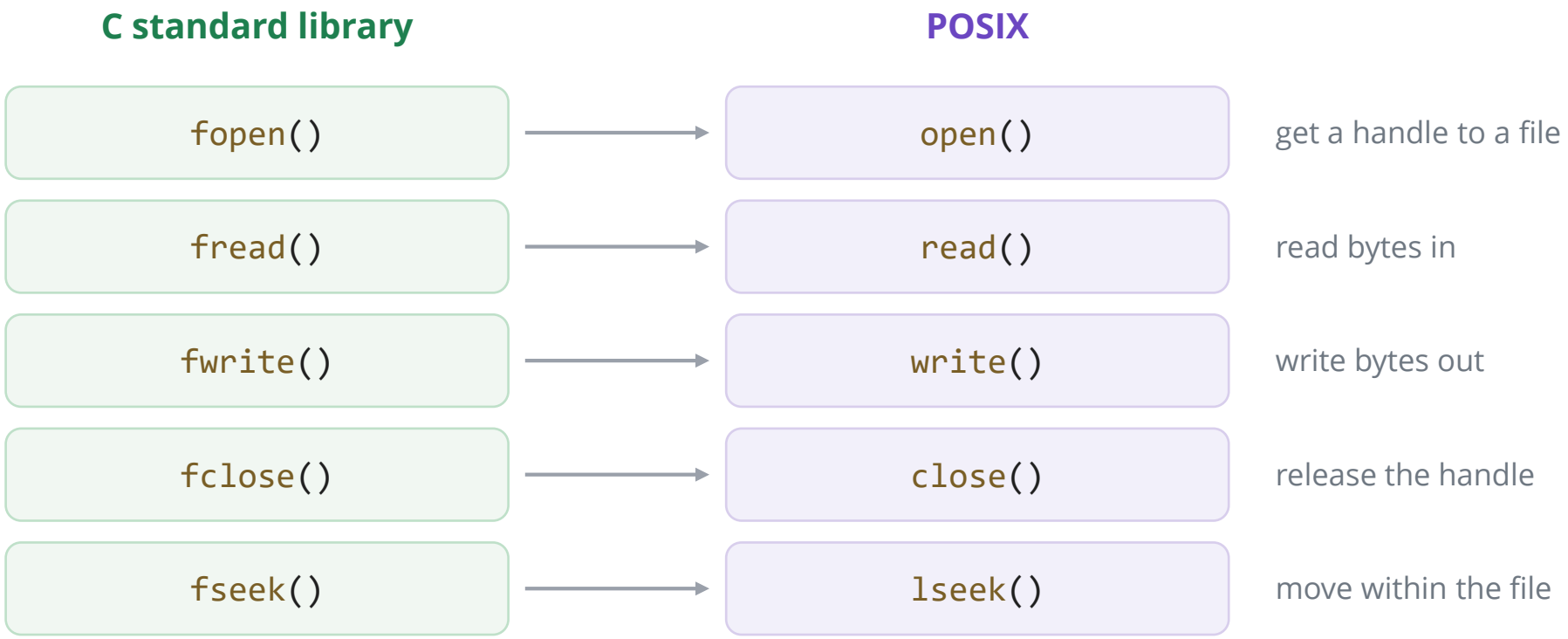


Where POSIX sits



- **So far: the C stream API.** Convenient, portable, buffered.
- Those f-functions are not magic. They are implemented on top of lower-level OS calls.
- **POSIX** is the common set of those lower-level calls on UNIX-like systems.
- **Today we call POSIX directly** so we can do things streams cannot: read directories, do network I/O.

From C streams to POSIX calls



Same jobs, but POSIX is **lower level, unbuffered, and less convenient**. No FILE*, no automatic buffer, and you handle more edge cases yourself.

open() gives you back ... what?

```
1  int fd = open("foo.txt", O_RDONLY);
2  // fd is an int, not a FILE*. What is it?
```

- **It is a small integer.** Your first open often hands back 3, never 0, 1, or 2. Why those numbers?

open / close and file descriptors

opening a file

```
1  #include <fcntl.h>    // open()
2  #include <unistd.h>   // close()
3
4  int fd = open("foo.txt", O_RDONLY);
5  if (fd == -1) {
6    perror("open failed");
7    exit(EXIT_FAILURE);
8  }
9  // ... use fd ...
10 close(fd);
```

- You get back a **file descriptor**: a small **int**, not a FILE*.
- **-1** means the open failed (check errno).

file descriptor table (per process)

0	stdin
1	stdout
2	stderr
3	foo.txt

← **open returned 3**

0, 1, 2 are already open for every process: stdin, stdout, stderr. Your first open gets the lowest free number, 3.

read()

```
1  ssize_t read(int fd, void* buf, size_t count);
```

- Reads up to **count** bytes from **fd** into **buf**, and advances the file position by that many.
- Returns the number of bytes actually read, **which may be fewer than you asked for.**
  - **0** means end-of-file: there is nothing left.
  - **-1** means error, and sets **errno**.
- **Surprising error you must handle: EINTR**, the call was interrupted before reading anything, so just try again.

You asked for 1000 bytes...

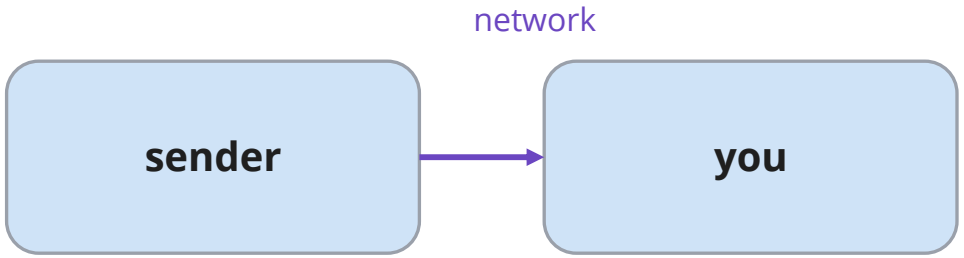
```
1  ssize_t got = read(fd, buf, 1000);
2  // can got ever be less than 1000?
```

- You requested 1000 bytes from an open fd. **Is a single read guaranteed to hand you all 1000?**

Predict first: when might one read() give back fewer bytes than you asked for?

Why would a read come up short?

- **You asked for 1000 bytes** and read returned 30. Why?
- The data is not all there yet. For a pipe or a network socket, the sender may have only sent 30 bytes so far.
- read does not wait for the rest. It gives you what is available now and returns.
- Regular files rarely do this, **which is exactly why the bug hides until you touch a socket.**




```
read(fd, buf, 1000) ...
```

returns **30**, because only 30 bytes have arrived so far. The other 970 are still in flight.

Example: complete the read() call

```
1  char* buf = ...;   // buffer of size n
2  int bytes_left = n;
3  int result;        // result of read()
4  while (bytes_left > 0) {
5    result = read(fd, ______, bytes_left);
6    if (result == -1) {
7      if (errno != EINTR) { /* real error */ }
8      continue;      // EINTR: try again
9    }
10   bytes_left -= result;
11 }
```

A `buf`

B `buf + bytes_left`

C `buf + (bytes_left - n)`

D `buf + (n - bytes_left)`

E `We're lost...`

Think about where the next chunk should land: **after the bytes you already have.**

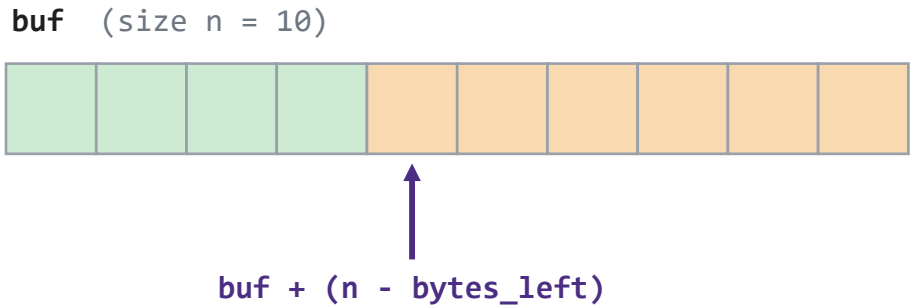
Step 1 of 3

Reading exactly n bytes

readN.c

```
1  int bytes_left = n;
2  while (bytes_left > 0) {
3    result = read(fd, buf + (n - bytes_left), bytes_left);
4    if (result == -1) {
5      if (errno != EINTR) break;    // a real error
6      continue;                     // EINTR: just retry
7    } else if (result == 0) {
8      break;                        // EOF
9    }
10   bytes_left -= result;
11 }
```

buf as the loop runs (n = 10):



bytes_left = **6** **green = read**, **orange = todo**

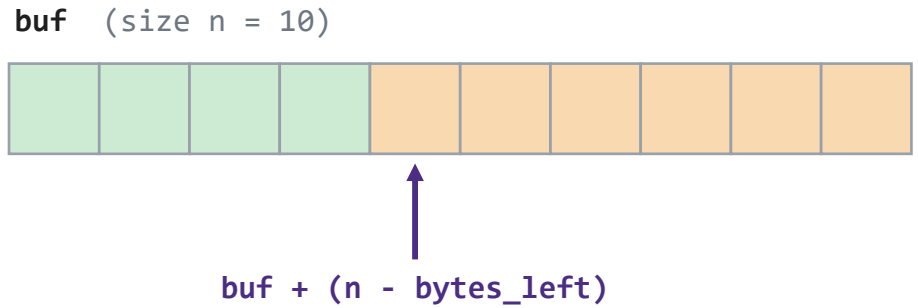
First read returns 4 (a short read). Those 4 bytes land at the front, and the cursor moves to buf + 4. bytes_left drops from 10 to 6.

Reading exactly n bytes

readN.c

```
1  int bytes_left = n;
2  while (bytes_left > 0) {
3    result = read(fd, buf + (n - bytes_left), bytes_left);
4    if (result == -1) {
5      if (errno != EINTR) break;   // a real error
6      continue;                    // EINTR: just retry
7    } else if (result == 0) {
8      break;                       // EOF
9    }
10   bytes_left -= result;
11 }
```

buf as the loop runs (n = 10):



bytes_left = **6** **green = read**, **orange = todo**

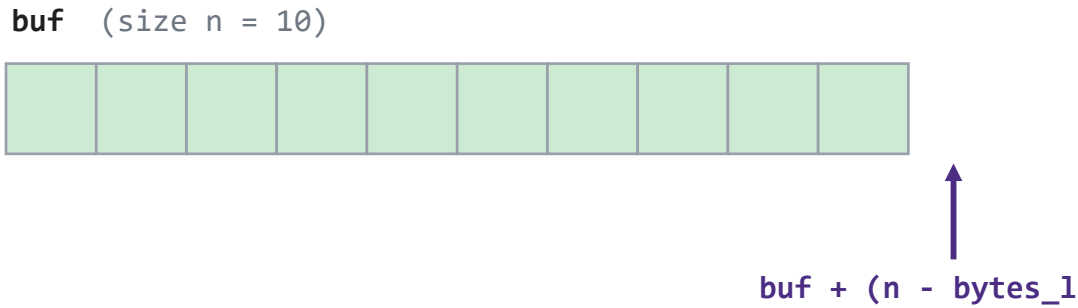
Next read is interrupted: it returns -1 with errno EINTR. No bytes moved, so we make no progress and just continue the loop to try again.

Reading exactly n bytes

readN.c

```
1  int bytes_left = n;
2  while (bytes_left > 0) {
3    result = read(fd, buf + (n - bytes_left), bytes_left);
4    if (result == -1) {
5      if (errno != EINTR) break;   // a real error
6      continue;                    // EINTR: just retry
7    } else if (result == 0) {
8      break;                       // EOF
9    }
10   bytes_left -= result;
11 }
```

buf as the loop runs (n = 10):



bytes_left = **0** **green = read**, **orange = todo**

Later reads fill the rest. When bytes_left hits 0 the loop ends and buf holds all n bytes, assembled from however many reads it took.

write()

```
1  ssize_t write(int fd, const void* buf, size_t count);
```

WriteAll: write all n bytes

```
1  int bytes_left = n;
2  while (bytes_left > 0) {
3    result = write(fd, buf + (n - bytes_left), bytes_left);
4    if (result == -1) {
5      if (errno != EINTR) break;    // a real error
6      continue;                     // EINTR: just retry
7    }
8    bytes_left -= result;
9  }
```

❖ **write can be short too.** It may accept fewer bytes than you handed it, especially to pipes and sockets.

❖ **Same approach as reading:** loop, advance by what actually got written, retry on EINTR.

❖ No EOF case. Writing does not hit end-of-file, so the loop is a touch simpler than read's.

❖ **Learn this pattern once.** ReadAll and WriteAll show up all over systems code (and HW2).

errno and perror

```
#include <errno.h>    // errno
#include <stdio.h>    // perror
#include <string.h>   // strerror
```

```
ssize_t n = read(fd, buf, count);
if (n == -1) {
  perror("read");   // read: <reason>
  return EXIT_FAILURE;
}
```

❖ **A -1 return means error,** and the call sets the global **errno** to say why.

❖ **perror(msg)** prints your message plus a readable version of errno to stderr.

❖ **strerror(errno)** gives that text as a string if you want to format it yourself.

❖ **Read errno right away.** The next library call may overwrite it.

fread vs read

C stream: fread

```
1  size_t got = fread(buf, 1, n, fp);
2  // got == n unless EOF or error
```

❖ **Buffered.** The library keeps reading under the hood until it has your n bytes (or hits EOF).

❖ One call is usually enough. The loop is hidden inside libc.

❖ Convenient, less control.

POSIX: read

```
1  ssize_t got = read(fd, buf, n);
2  // got may be < n at any time
```

❖ **Unbuffered.** You get exactly what one system call returned, no more.

❖ **You write the loop.** Short reads are yours to handle, as we just saw.

❖ Raw control, and the only option for directories and sockets.

Other low-level calls

`write()` `<unistd.h>`	write bytes to an fd
`fsync()` `<unistd.h>`	force buffered data out to the disk
`opendir()` / `readdir()` / `closedir()` `<dirent.h>`	walk a directory's entries
`lseek()` `<unistd.h>`	jump to a byte offset in the file

The man pages are the reference; read them, especially the right section:

What's the difference?

```c
FILE* fp = fopen("out.txt", "w");
fwrite("hi", 1, 2, fp);   // returns 2
// the power cuts out right here
```

```c
int fd = open("foo.txt", O_RDONLY);
char* data = "hi";
write(fd, "hi", strlen(data));   // returns 2
// the power cuts out right here
```

File Buffering

Intuition: it is like a video buffering

❖ **Watch a video on YouTube or Netflix** and the player quietly downloads ahead of where you are, into a buffer.

❖ Playback stays smooth because it reads from that local buffer, not straight from the slow network.

❖ Network hiccup? You keep watching from the buffer. Only when it runs dry do you see “buffering…”

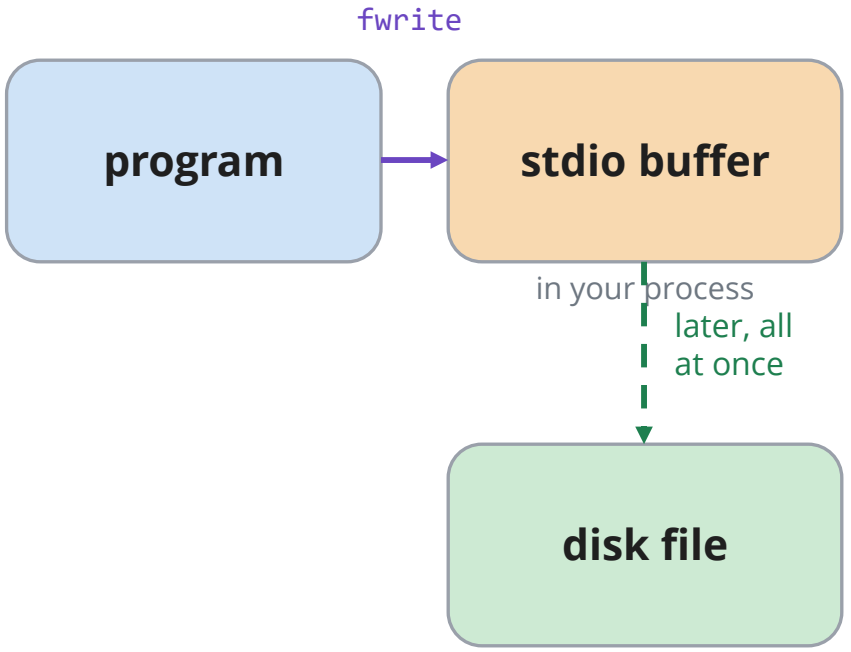
❖ **A file buffer plays the same role:** it sits between your program and the slow disk, so you are not waiting on the disk for every byte.

Same idea, two worlds:

the video player	↔	**your program**
the network (slow)	↔	**the disk (slow)**
the download buffer	↔	**the stdio buffer**

Your writes do not go straight to disk

- **fwrite copies into a buffer** that stdio keeps inside your process, not onto the disk.
- **Later, the buffer is drained** to the real file in one larger write.
- **So "the write returned"** does not mean "the bytes are on disk."



When does the buffer actually drain?

- You call **fflush(stream)** explicitly.
- The buffer fills up (often 1024 or 4096 bytes).
- Line-buffered to a console: a **'\n'** is written.
- You call **fclose(stream)**.
- The process exits gracefully (**exit()** or return from **main**).

Same program, two behaviors:

```
$ ./prog        # to a terminal
```

line buffered: you see each line as it prints.

```
$ ./prog | cat   # into a pipe
```

fully buffered: nothing appears until the buffer fills or the program exits. Same code, different device.

Buffered

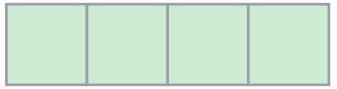
buffered (default)

```
1  FILE *fp = fopen("test.txt", "wb");
2  fwrite("h", 1, 1, fp);   // write 'h'
3  fwrite("i", 1, 1, fp);   // write 'i'
4  // fclose(fp);
```

after both fwrite calls:

stdio buffer

test.txt (disk)



still empty!

Both chars sit in the buffer. The disk file is empty until **fclose drains it. A crash here would lose both.**

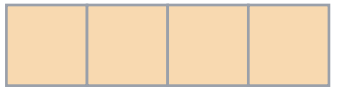
Unbuffered

unbuffered (setbuf NULL)

```
1  FILE *fp = fopen("test.txt", "wb");
2  setbuf(fp, NULL);          // buffering OFF
3  fwrite("h", 1, 1, fp);     // write 'h'
4  fwrite("i", 1, 1, fp);     // write 'i'
5  // fclose(fp);
```

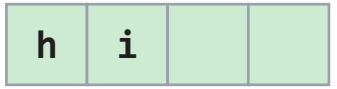
after both fwrite calls:

stdio buffer



stays empty

test.txt (disk)



Super slow!

setbuf(fp, NULL) turns buffering off, so each **fwrite** goes straight to the file. Safer, but one trip to disk per character.

Losing logs during a crash!

buggy.c

```
1  printf("hi");    // no newline!
2  int* p = NULL;
3  *p = 5;     // purposefully segfault here
```

the fix

```
1  printf("hi\n")    // newline flushes
2  fflush(stdout);   // or force it yourself
```

What you expect vs. what you get

"hi" went into stdout's buffer, not the screen. Then the segfault kills the process **before the buffer is flushed**, so the message never prints. It looks like the crash came earlier than it did.

Debugging lesson: do not trust the last printed line to mark where a program died. Add **\n** or **fflush**, or print progress to unbuffered **stderr**.

Why buffer, and why not

Why buffer

- **Performance.** Many tiny writes become one big write, so you touch the slow disk far less often.
- **Disk is slow.** A disk access is millions of times slower than a memory copy; batching hides that.
- **Convenience.** A nicer API than raw system calls (next lecture).

Why not

- **Reliability.** A crash or power loss drops whatever was still in the buffer.
- **Latency.** If you need a byte out now (a log, a prompt), buffering delays it.
- **Copy cost.** Copying into the buffer burns CPU and memory bandwidth; servers chase "zero-copy".

System Calls

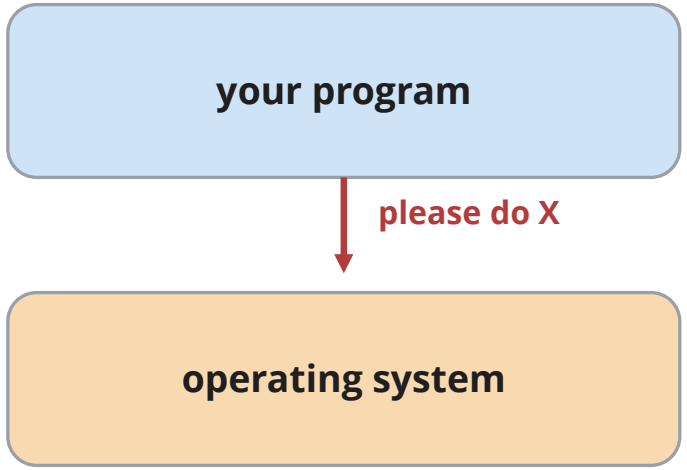
What a system call is

❖ **Your program cannot touch the hardware directly.** Not the disk, not the network card, not the screen.

❖ When it needs one of those, it asks the operating system to do it. That request is a **system call**.

❖ Think of a bank teller: you cannot walk into the vault, so you hand over a slip and wait while they fetch what you asked for.

❖ **read and write are exactly this.** Today we open the hood and watch one request happen.



Syscalls are everywhere

```c
int main() {
  printf("hello\n");
  return EXIT_SUCCESS;
}
```

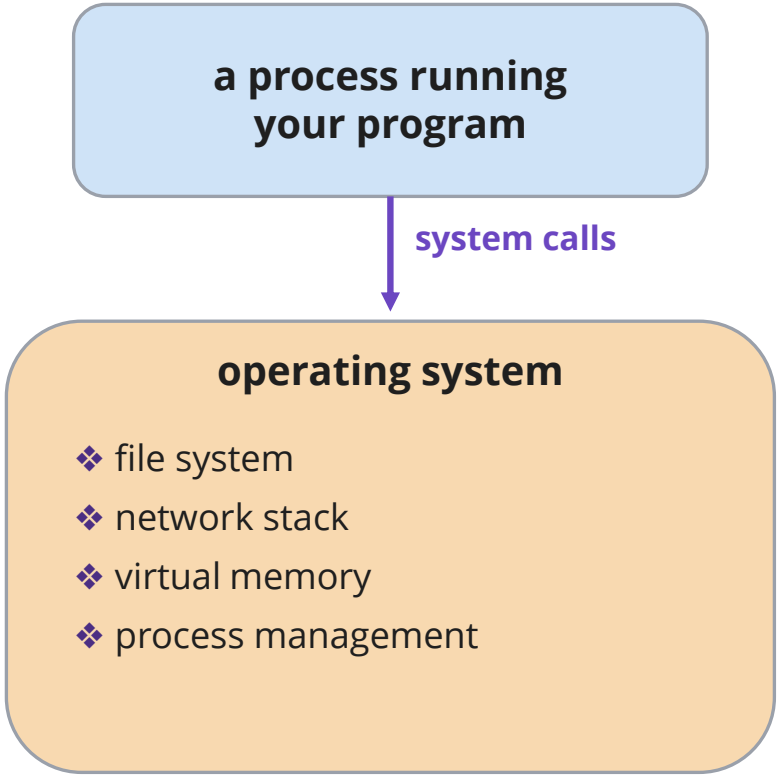
```
$ strace ./hello
$ execve("./hello", ...)      = 0
$ brk(NULL)                   = ...
$ mmap(..., MAP_ANON, ...)    = ...
$ openat(..., "libc.so.6", ...) = 3
$ read(3, "\177ELF...", 832)   = 832
$ write(1, "hello\n", 6)       = 6
$ exit_group(0)               = ?
```

- **strace** lists every system call a process makes.
- Your one printf becomes a single **write(1, ...), the highlighted line.**
- All the noise above it is startup: mapping memory and loading the C library, itself done with syscalls.
- **Every interaction with the outside world** (files, console, network, memory, processes) goes through a system call.

What is an operating system?

- **Talks to the hardware.** The OS is trusted to drive devices directly; your program is not. It is also what gets ported to new hardware, so your program stays portable.
- **Manages the resources.** It allocates, schedules, and protects the CPU, memory, disk, and screen, and decides which program gets what and when.
- **Abstracts the mess.** Raw devices are ugly and all different. The OS hides them behind clean, portable ideas like files and disk blocks.

The OS offers an API



File system

`open()`, `read()`, `write()`, `close()`, ...

Network stack

`connect()`, `listen()`, `read()`, `write()`, ...

Virtual memory

`brk()`, `mmap()`, `shm_open()`, ...

Process mgmt.

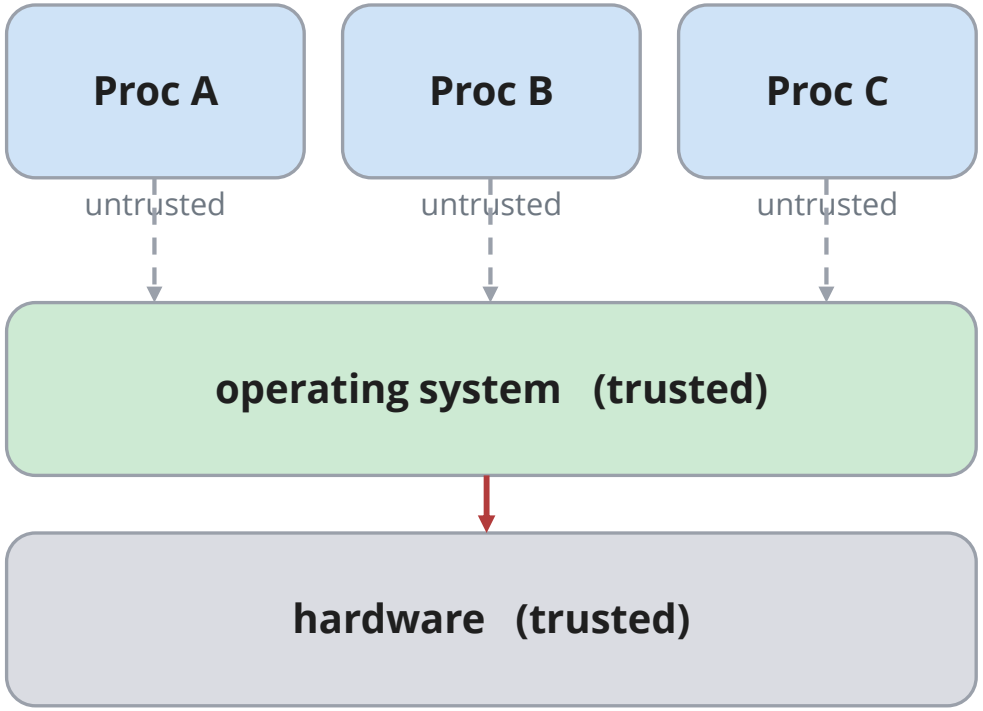
`fork()`, `wait()`, `exec()`, `nice()`, ...

Why not touch the hardware directly?

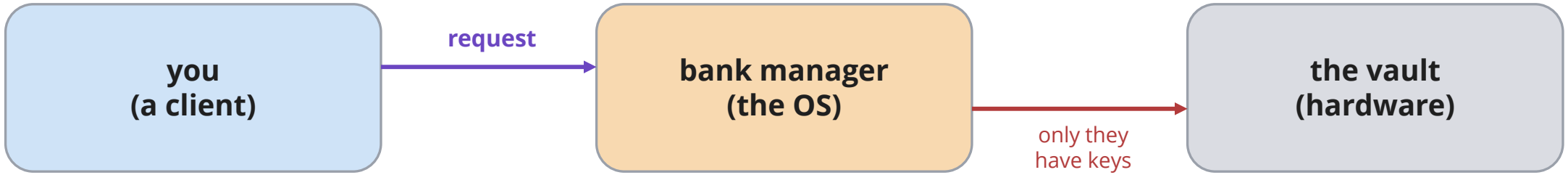
- Your program knows exactly what it wants: write these bytes to that disk.
- **So why can't it just run the instruction** that talks to the disk itself, and skip asking the OS?
- **What would go wrong** if every program could drive the hardware directly?

Two CPU modes: user and kernel

- **User code runs unprivileged.** The CPU is in a restricted mode; hardware-touching instructions are forbidden.
- **The OS runs privileged.** Only it may talk to devices directly.
- **Isolation.** The OS keeps processes out of each other's memory, with controlled sharing through names like files.
- **The only door in** is a system call, which safely switches the CPU into privileged mode inside the OS.



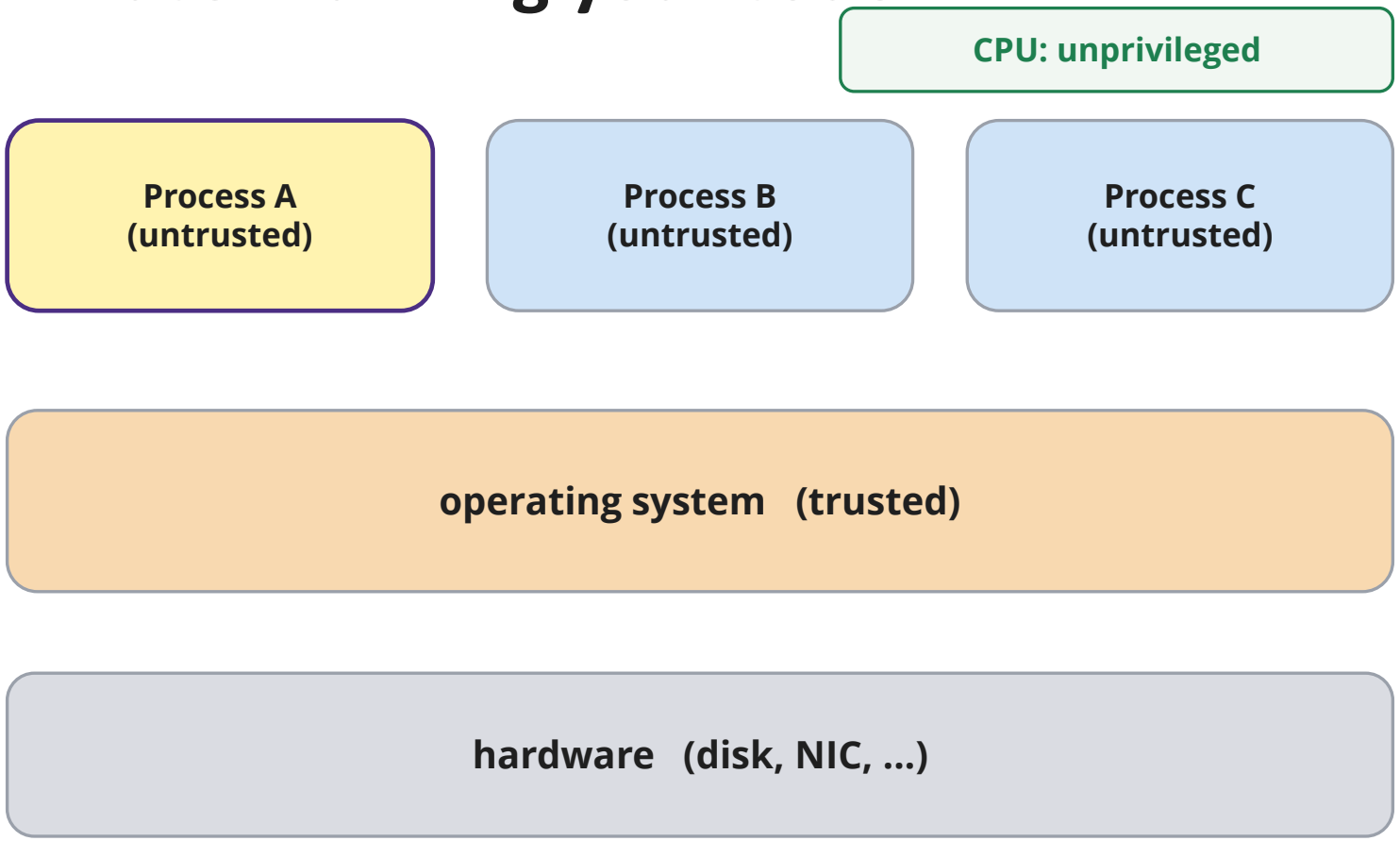
Analogy: the bank vault



- **You cannot enter the vault.** Only the manager has the keys, just as only the OS may touch the hardware.
- You ask; they fetch. You wait in the lobby while the manager goes to your box, which keeps you from messing with other boxes.
- **It takes time.** Walking to the vault, finding the key, and coming back is overhead, exactly like the cost of a system call.

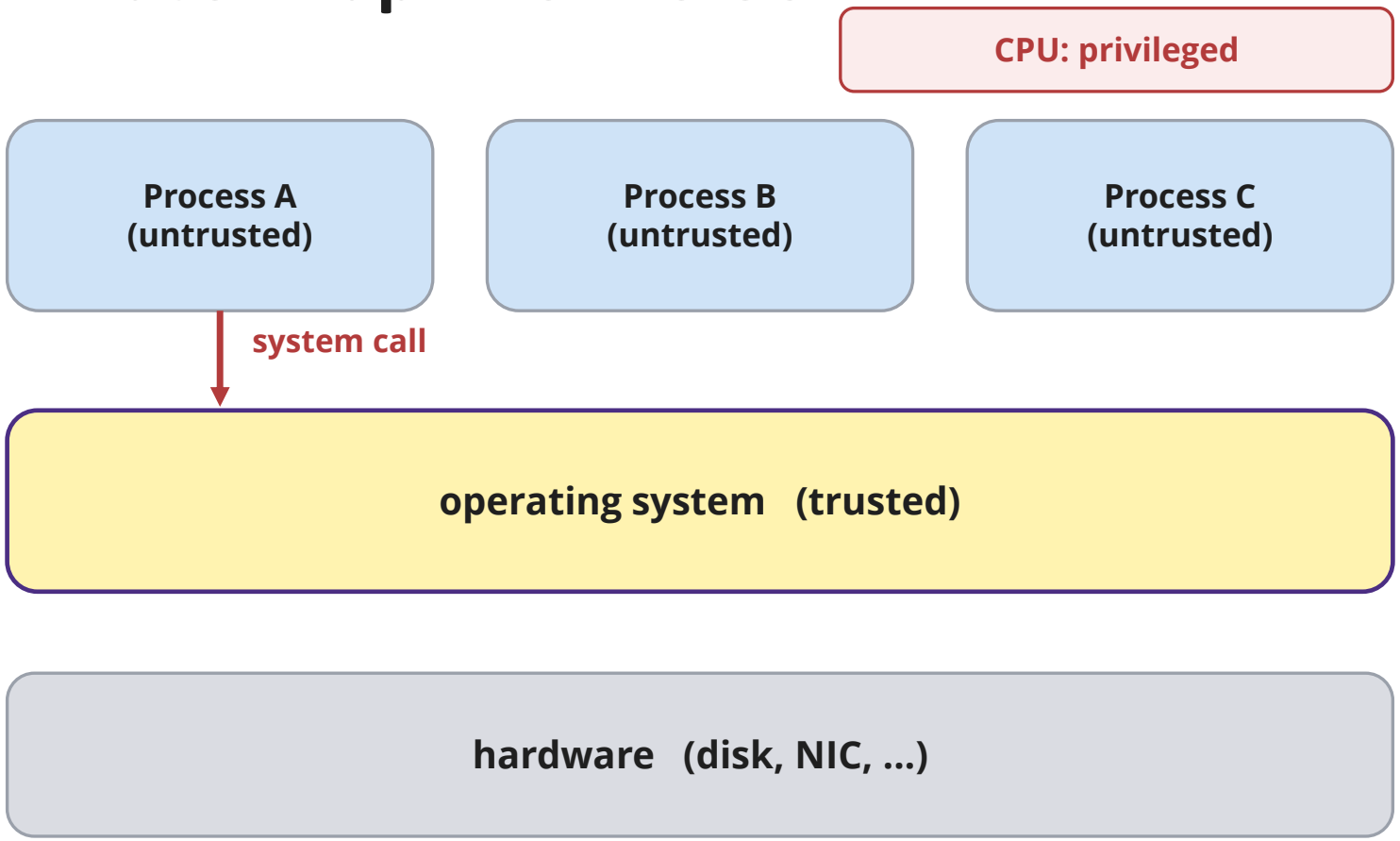
Step 1 of 5

System call trace: Running your code



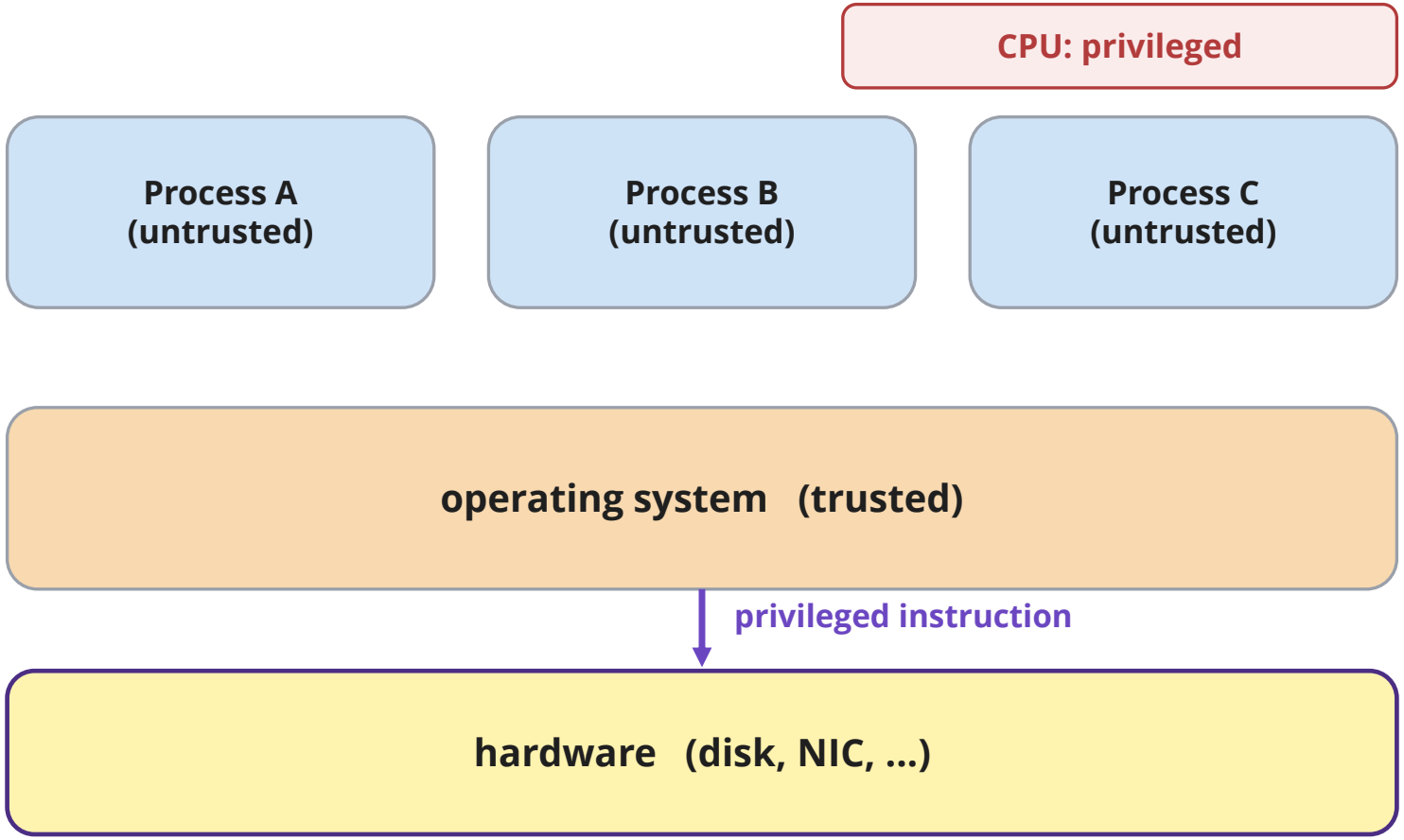
A CPU is running user-level code in Process A. The CPU is in unprivileged mode, so it cannot touch hardware directly.

System call trace: Trap into the OS



Process A invokes a system call. The hardware switches the CPU to privileged mode and traps into the OS, which runs the matching system-call handler.

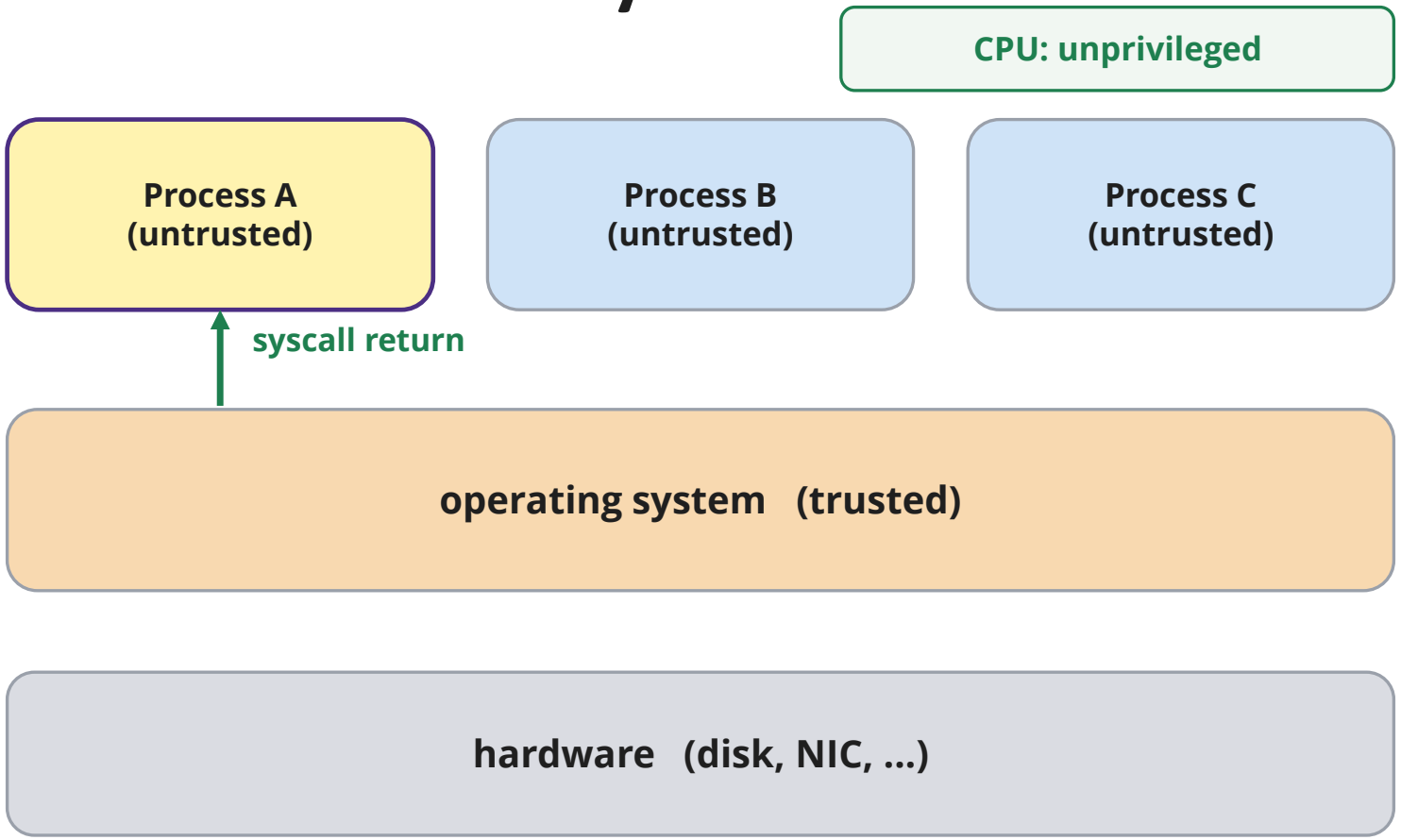
System call trace: The OS drives the hardware



Because the CPU is now privileged, the OS can run privileged instructions that talk directly to hardware, like the disk.

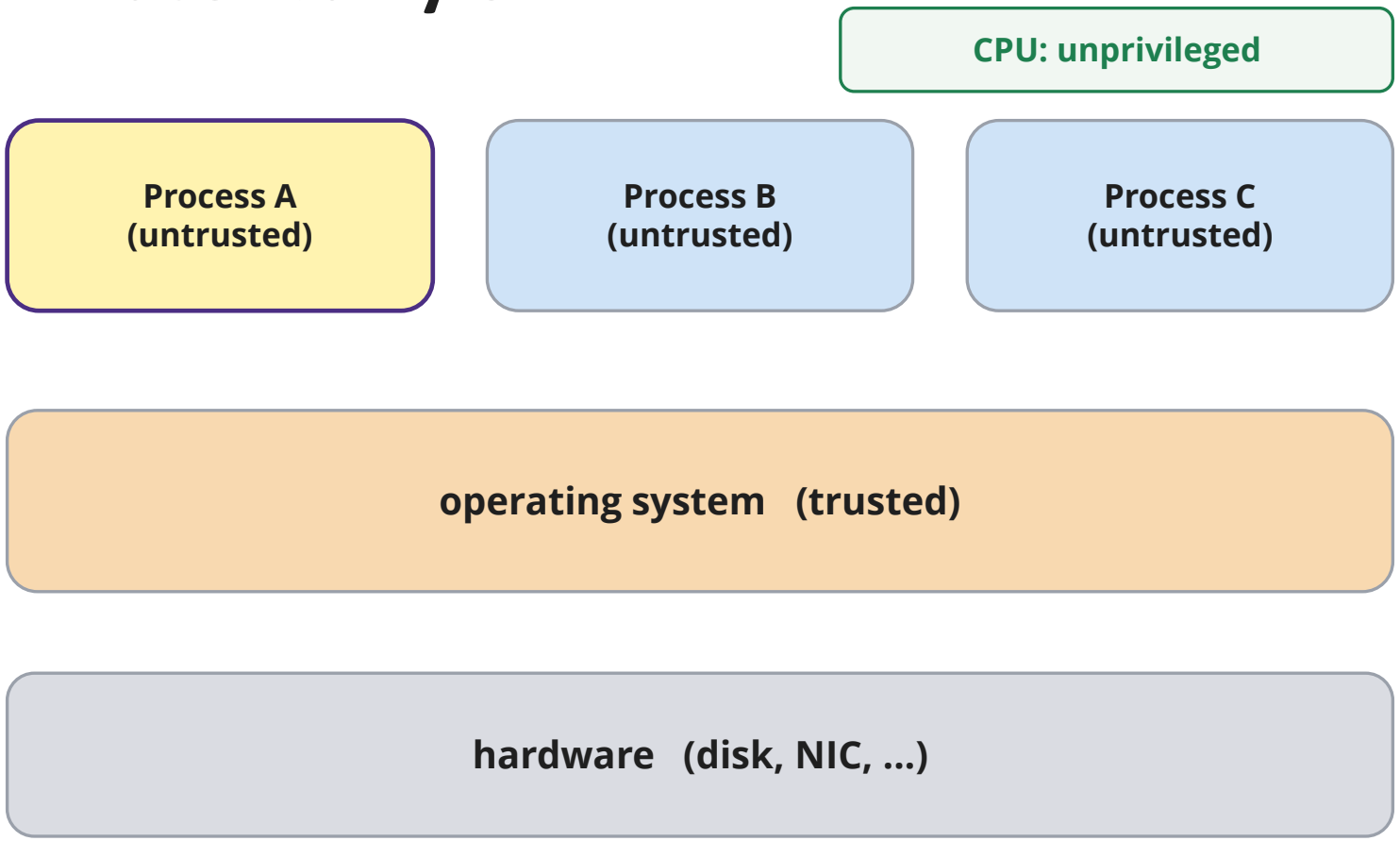
This is the slow part: hardware waits can be long, so the OS may run another process meanwhile.

System call trace: Return to your code



When the handler finishes (possibly after a long hardware wait), the OS sets the CPU back to unprivileged mode and returns into Process A's code.

System call trace: Carry on



Process A carries on with whatever code came after the system call, none the wiser about the mode switches that happened.

Three ways a library call resolves

1. Stays in glibc

```
strcmp(), memcpy()
```

Pure computation. Handled entirely in the library; the kernel is never involved. As fast as any function call.

never leaves glibc

2. glibc → syscall

```
fopen() → open()
```

The library does a syscall under the hood. C stdio and POSIX wrappers both end in a trap into the kernel.

reaches the kernel

3. direct syscall

```
syscall(SYS_write, ...)
```

You invoke the kernel yourself, skipping glibc. Possible, but less portable across UNIX flavors.

reaches the kernel

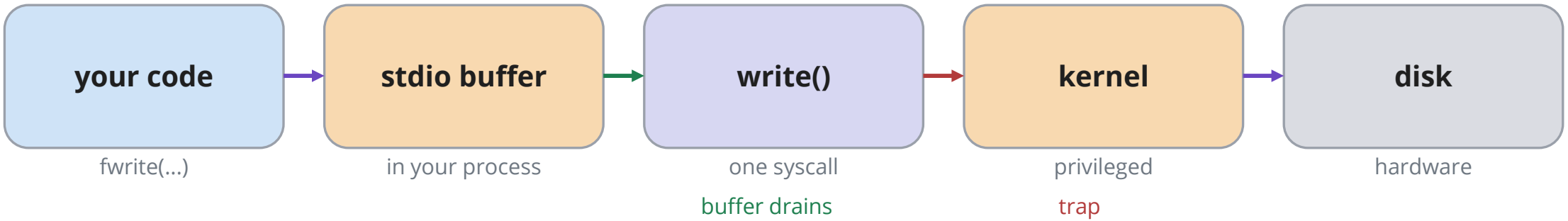
Tracing one fwrite

```
1  fwrite(buf, 1, 4, fp);   // just 4 bytes
```

- **Follow those four bytes all the way down.** Which layers do they pass through, in order?
- **And the sharp one:** how many times does the CPU trap into the kernel for this single fwrite?

Predict first: stdio buffer, syscall, trap, kernel, disk: put them in order, then count the traps.

How the three ideas connect



- **File Buffering (deck 1)** exists to make step 3 rare: many fwrite calls collapse into one write() syscall.
- **POSIX read/write (deck 2)** are those syscalls, the actual requests to the OS.
- **The trap (deck 3)** is why each syscall is expensive, which is exactly why batching them with a buffer pays off.

Takeaway: Buffer to avoid syscalls, use read/write to make them, and trap into the kernel to run them. The three ideas are one story.

Reminders

Assessments:

- Exercise 3 due tonight at 11:59pm
- Homework 1 due tomorrow at 11:59pm
  - VS Code Debugging tutorial might be helpful

General:

- Sign up for a coffee chat!
- Exams are scheduled
  - Midterm (7/27) from 9:40am to 10:40am in IEB G106
  - Final exam (8/18) from 1:30pm to 3:30pm in CSE2 G10

Questions?

Thanks for coming - see you next lecture!