UNIVERSITY of WASHINGTON

LECTURE 7 · CSE 333 · Summer 2026

System Calls; Intro to C++

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

Reminders

Assessments:

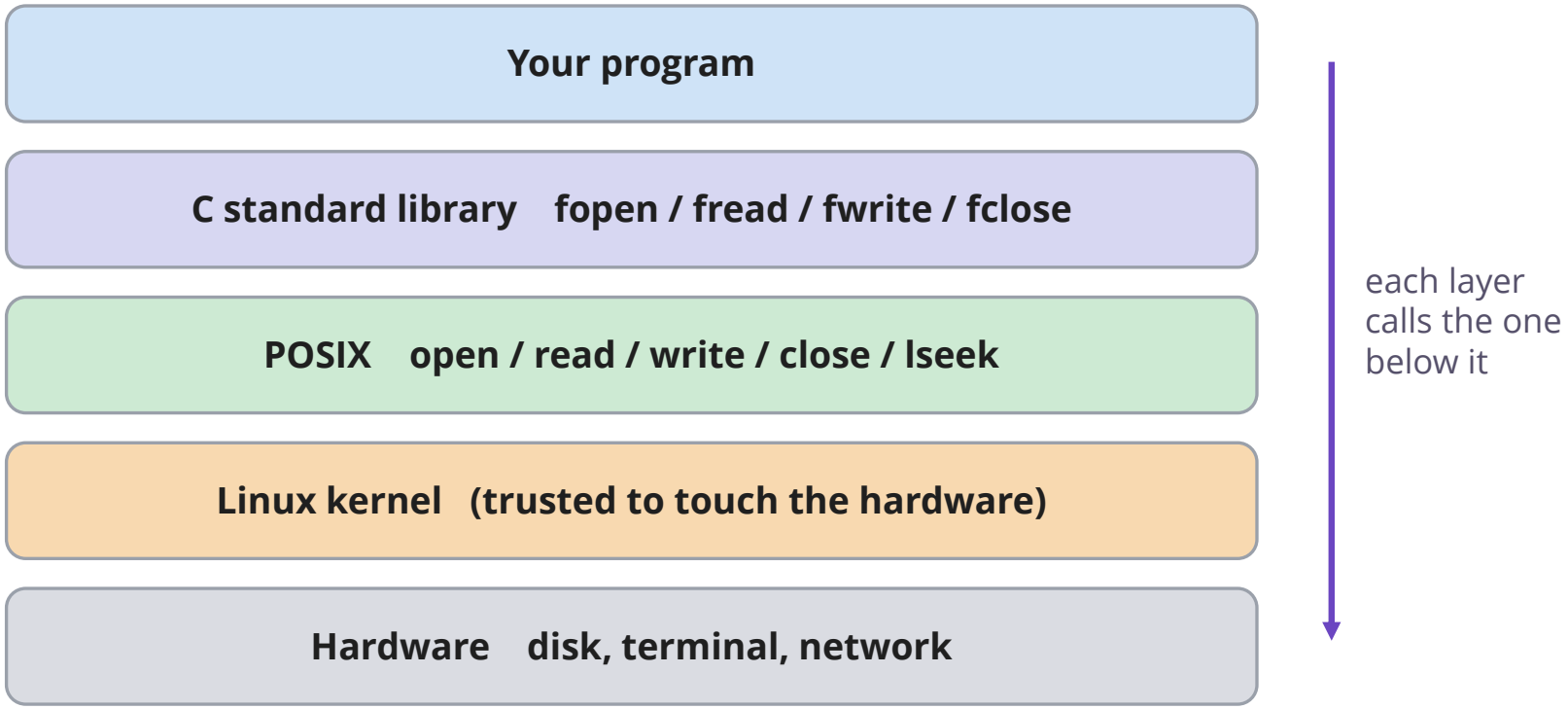
- Exercise 4 will be released today!
  - Due on Thursday at 11:59pm
  - Includes C and C++
- HW2 will be released early next week

General:

- C++
- 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

Review

The I/O software stack



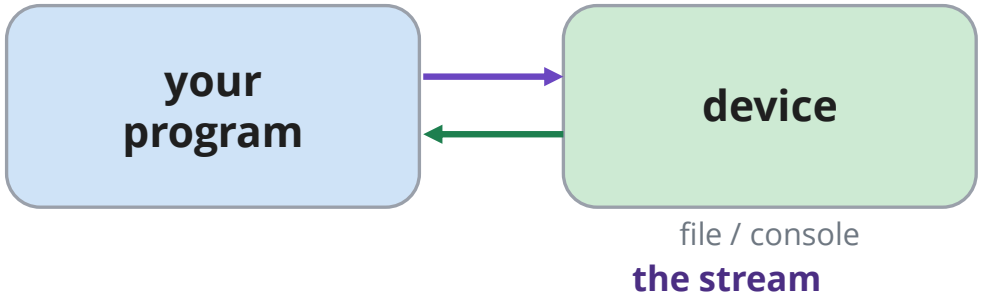
Streams

❖ **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.



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.

REVIEW

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.

REVIEW

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.**

❖ mode "w" creates an empty notes.txt but not all modes will do that.

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

REVIEW

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);
```

count = 4 items

each size = 4 bytes

- 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).

REVIEW

Reading raw bytes: fread

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

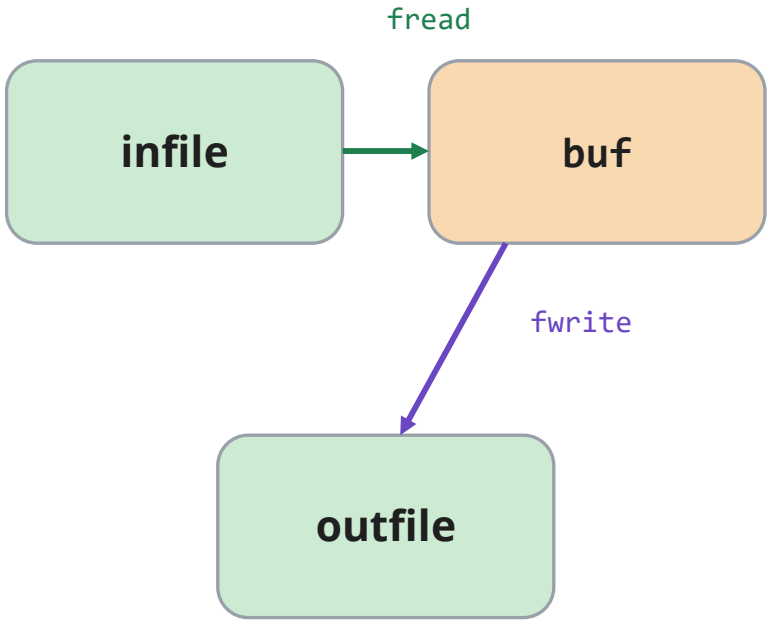
```
int a[4];
size_t got = fread(a, sizeof(int), 4, fp);
```

- **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**.

Example: copying a file

cp_example.c

```
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;
```



REVIEW

From C streams to POSIX calls

C standard library		POSIX	
`fopen()`	→	`open()`	get a handle to a file
`fread()`	→	`read()`	read bytes in
`fwrite()`	→	`write()`	write bytes out
`fclose()`	→	`close()`	release the handle
`fseek()`	→	`lseek()`	move within the file

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

REVIEW

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**

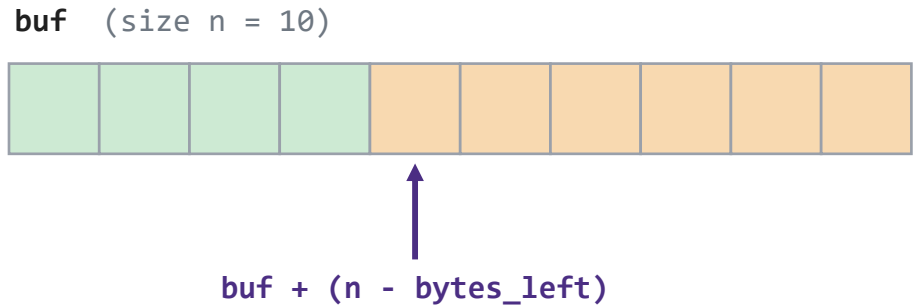
0, 1, 2 are already open for every process: stdin, stdout, stderr. Your first open gets the lowest free number, 3.

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**

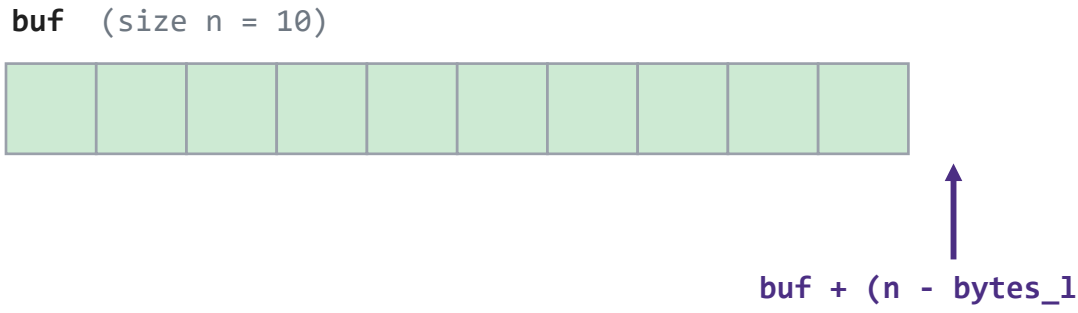
First read returns 4 (a short read). Those 4 bytes land at the front, and the cursor moves to buf + 4. bytes_left drops from 10 to 6.

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 = **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).

REVIEW

What's the difference?

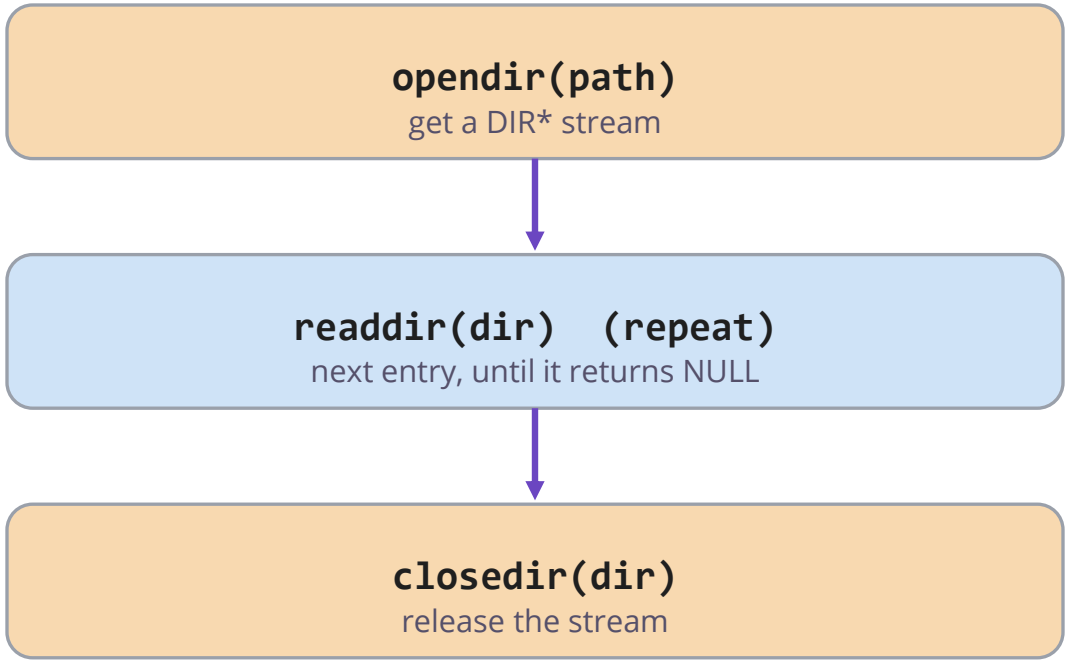
```c
FILE* fp = fopen("out.txt", "w");
fwrite("hi", 1, 2, fp);   // returns 2
// the power cuts out right here
```

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

Section Review

A directory is just a special file

- A directory is a file whose contents are a list of entries (names plus some metadata)
- So POSIX gives you directory calls that mirror open / read / close
- **You do not read the bytes yourself:** the library hands you one entry at a time



opendir(): get a directory stream

```
1  DIR* opendir(const char* name);
```

- **name** is the directory to open; relative or absolute paths both work
  - A trailing '/' is allowed but not required
- **Returns a DIR***, a handle to the directory stream (like FILE* for a file)
- **On error returns NULL** and sets errno

readdir(): one entry at a time

```
1  struct dirent* readdir(DIR* dirp);
```

- **dirp** is the stream to read from
- **Returns a struct dirent*** for the next entry in the directory
- **Returns NULL** at the end of the stream, or on error
  - So the usual loop is: call readdir until it returns NULL
- Each call advances to the next entry; you do not index into it

struct dirent

```
1  struct dirent {
2    ino_t          d_ino;     // inode number
3    off_t          d_off;     // not really an offset
4    unsigned short d_reclen;  // length of this record
5    unsigned char  d_type;    // file type (see note)
6    char           d_name[NAME_MAX + 1];  // the name
7  };
```

- **d_name** is what you almost always want
- The rest is bookkeeping the kernel uses

d_type is probably not what you think (not a file extension)

- **It is the entry's file type** (DT_REG, DT_DIR, DT_LNK, ...), not a general field. Not every filesystem fills it in; it can be DT_UNKNOWN, in which case use stat() to find the type.

closedir(): release the stream

```
1  int closedir(DIR* dirp);
```

- **dirp** is the directory stream to close
- **Returns 0** on success, **-1** on error (errno set)
- **Always close what you opened,** just like fclose

Example: list a directory

```
1  #include <dirent.h>    // opendir, readdir, closedir
2  #include <...>
3
4  int main(int argc, char** argv) {
5    DIR* dir = opendir(argv[1]);
6    if (dir == NULL) {
7      perror("opendir");
8      return EXIT_FAILURE;
9    }
10
11   struct dirent* entry;
12   while ((entry = readdir(dir)) != NULL) {
13     printf("%s\n", entry->d_name);
14   }
15   closedir(dir);
16   return EXIT_SUCCESS;
17 }
```

- Open the directory named on the command line
- **Guard the NULL:** report with perror and bail out
- Next: loop over the entries and print each name

Example: list a directory

```c
#include <dirent.h>    // opendir, readdir, closedir
#include <...>

int main(int argc, char** argv) {
  DIR* dir = opendir(argv[1]);
  if (dir == NULL) {
    perror("opendir");
    return EXIT_FAILURE;
  }

  struct dirent* entry;
  while ((entry = readdir(dir)) != NULL) {
    printf("%s\n", entry->d_name);
  }
  closedir(dir);
  return EXIT_SUCCESS;
}
```

- readdir in the loop condition: stop when it returns NULL
- **Print entry->d_name** each time
- Then closedir to release the stream

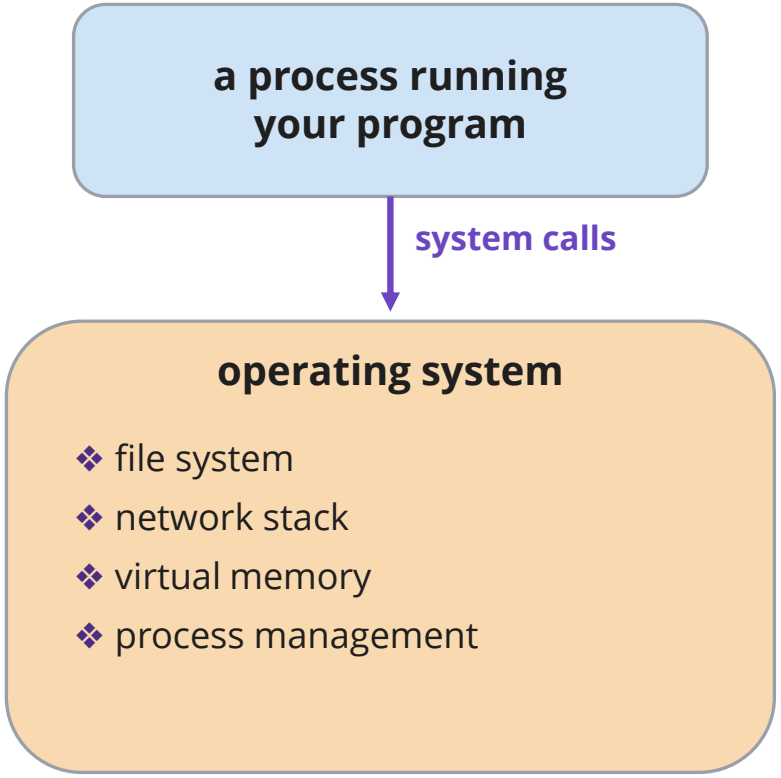
```
$ ./list .
$ .
$ ..
$ list.c
$ Makefile
```

System Calls

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

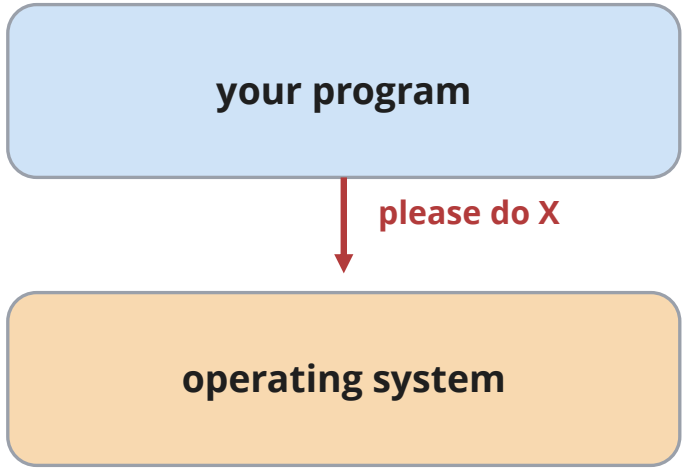
`brk()`, `mmap()`, `shm_open()`, ...

Process mgmt.

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

What is a system call?

- **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)               = ?
```

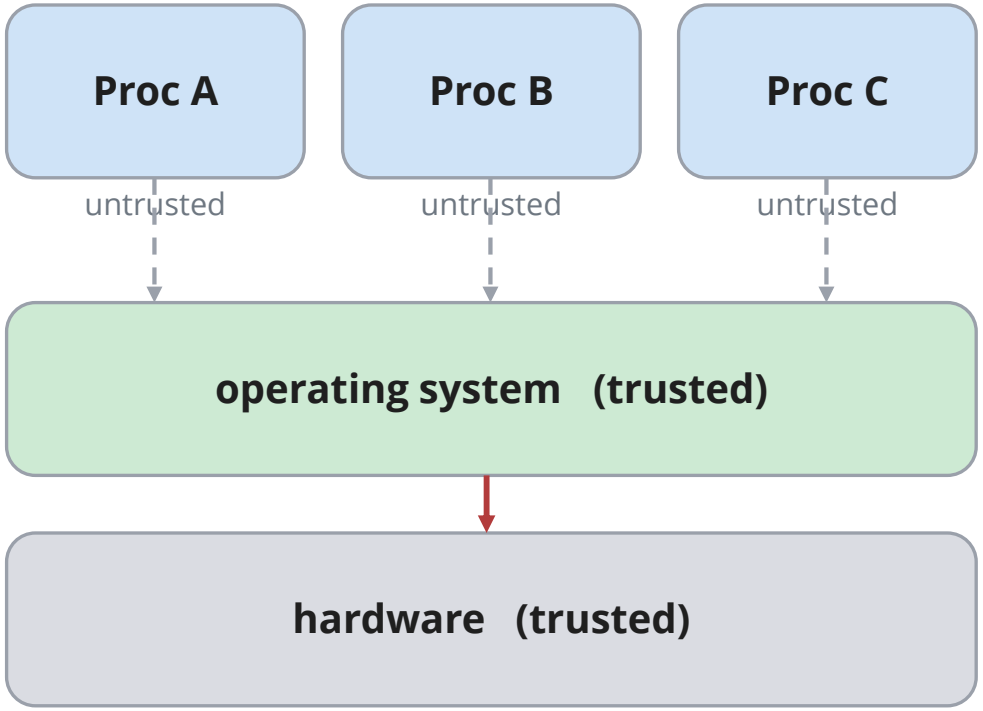
- **strace** lists every system call a process makes.
- Your one printf becomes a single **write(1, ...), the highlighted line.**
- All the noise above it is startup: mapping memory and loading the C library, itself done with syscalls.
- **Every interaction with the outside world** (files, console, network,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

Why not touch the hardware directly?

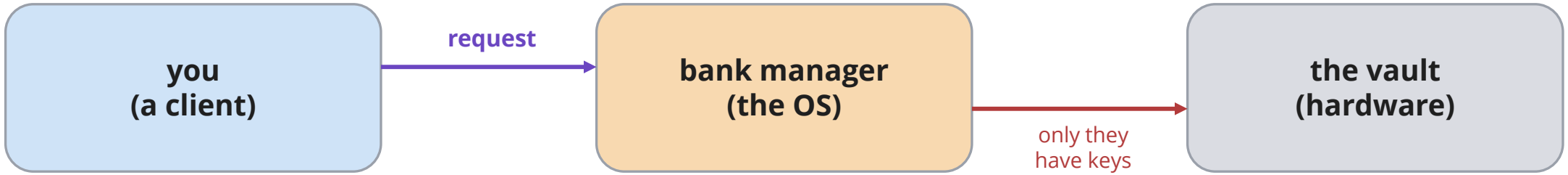
- Your program knows exactly what it wants: write these bytes to that disk.
- **So why can't it just run the instruction** that talks to the disk itself, and skip asking the OS?
- **What would go wrong** if every program could drive the hardware directly?

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.



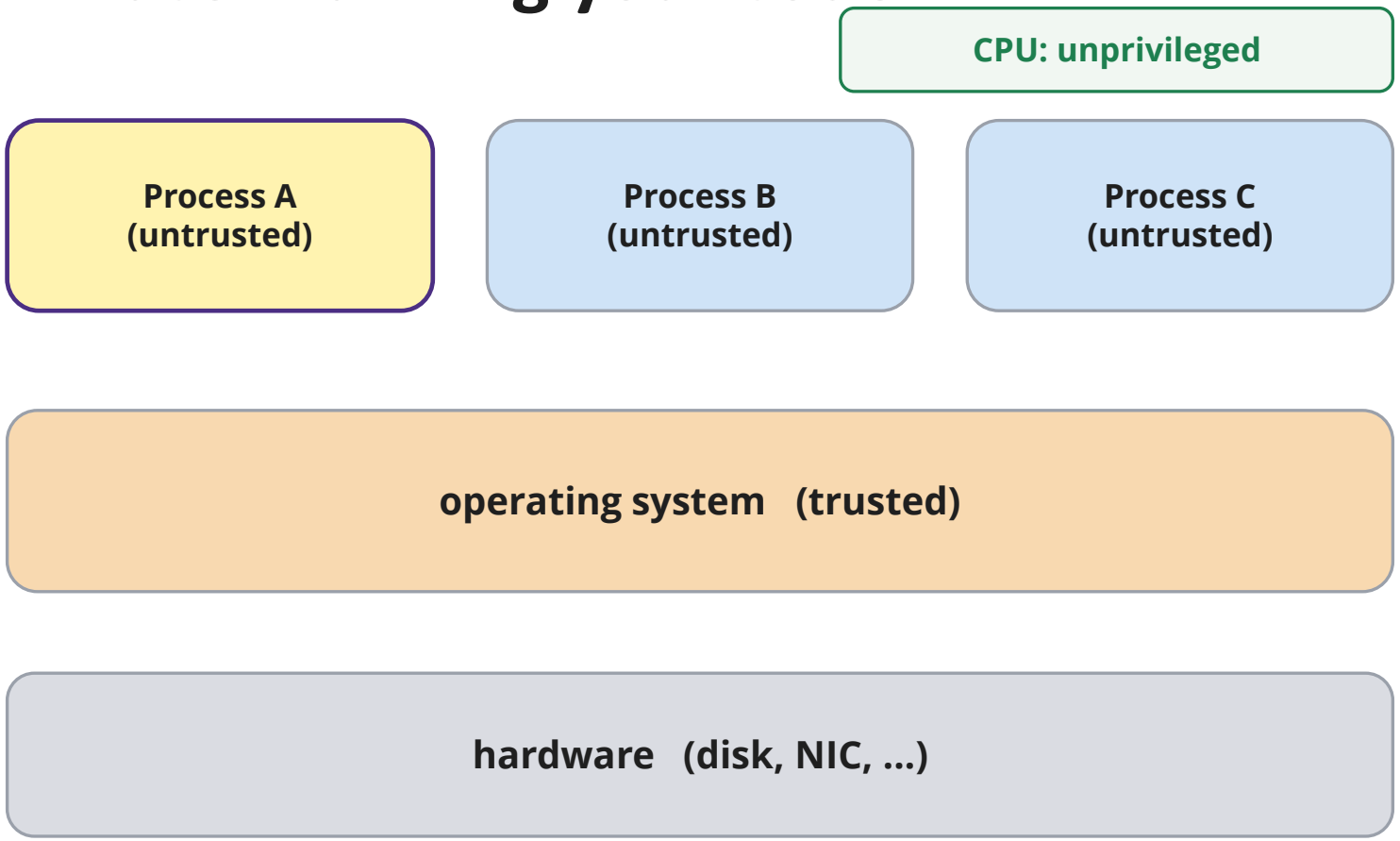
Analogy: the bank vault



- **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.

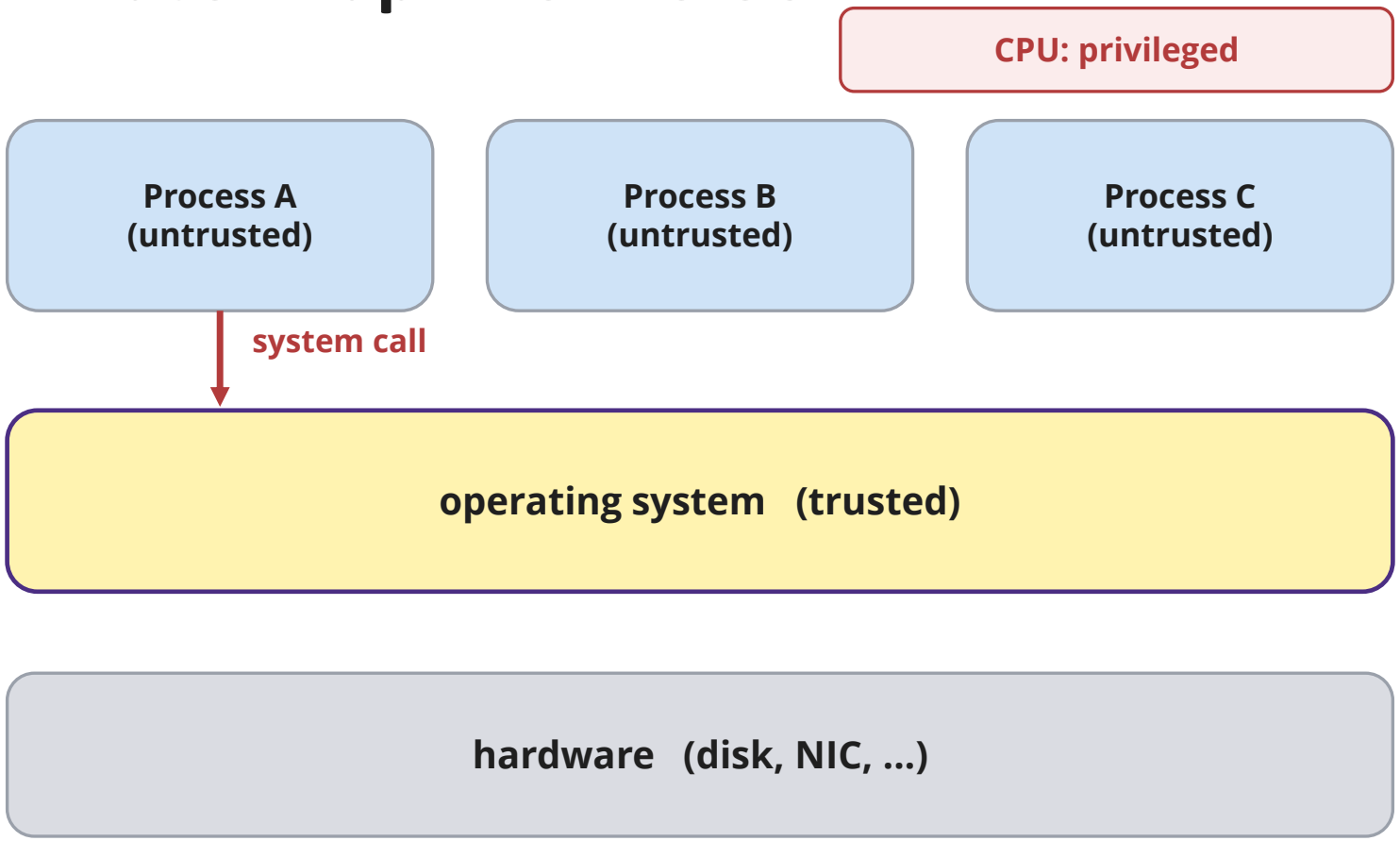
Step 1 of 5

System call trace: Running your code



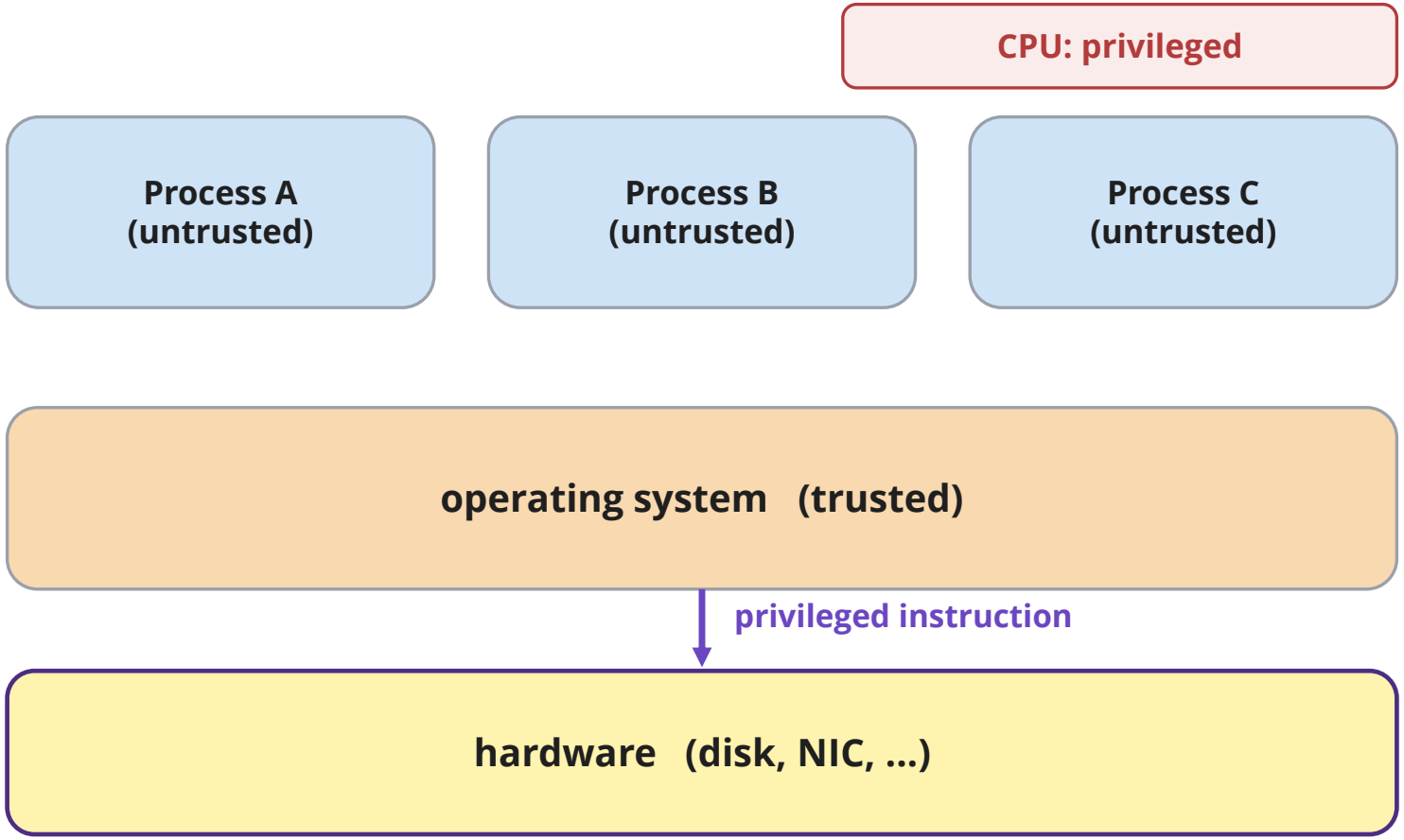
A CPU is running user-level code in Process A. The CPU is in unprivileged mode, so it cannot touch hardware directly.

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.

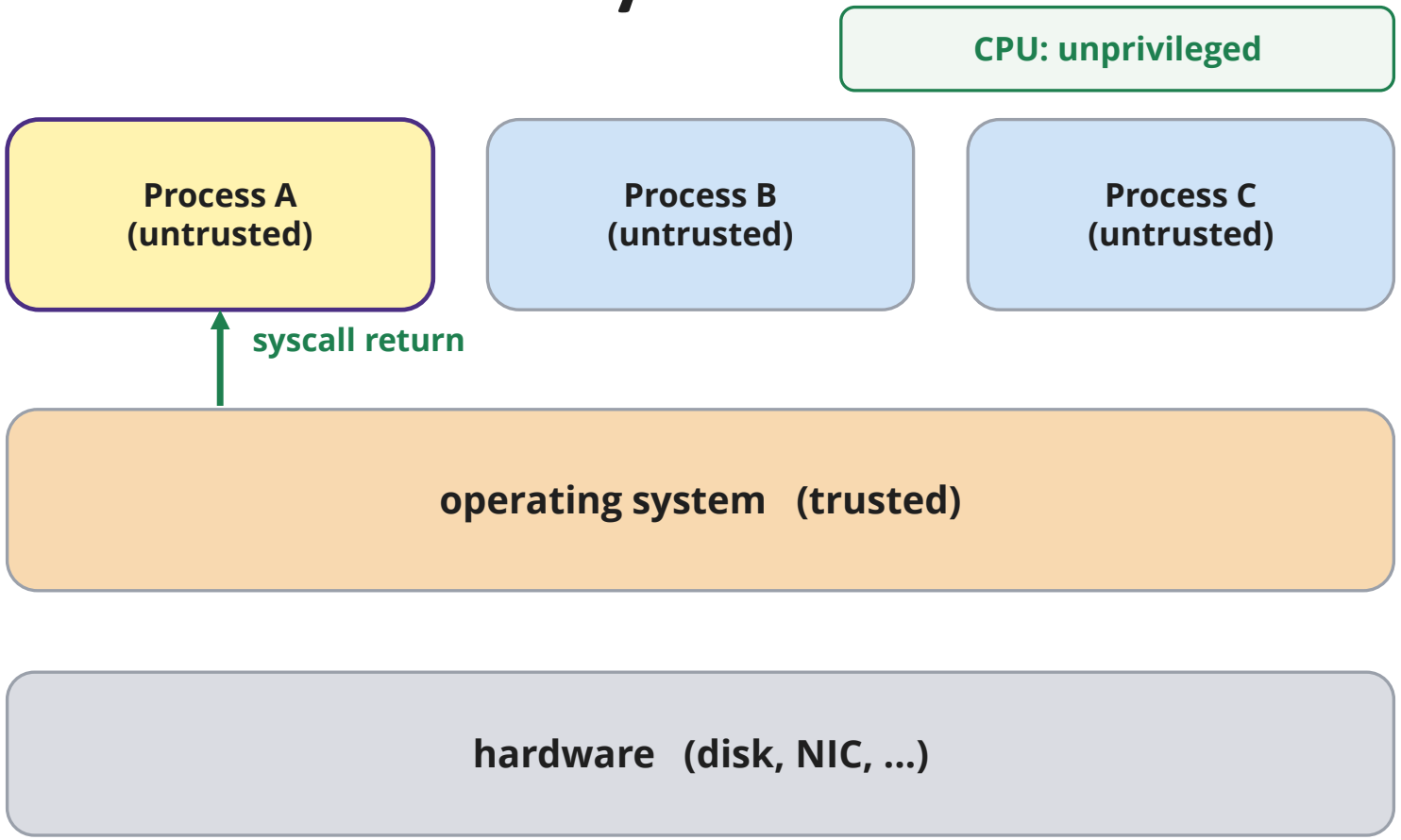
System call trace: The OS drives the hardware



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

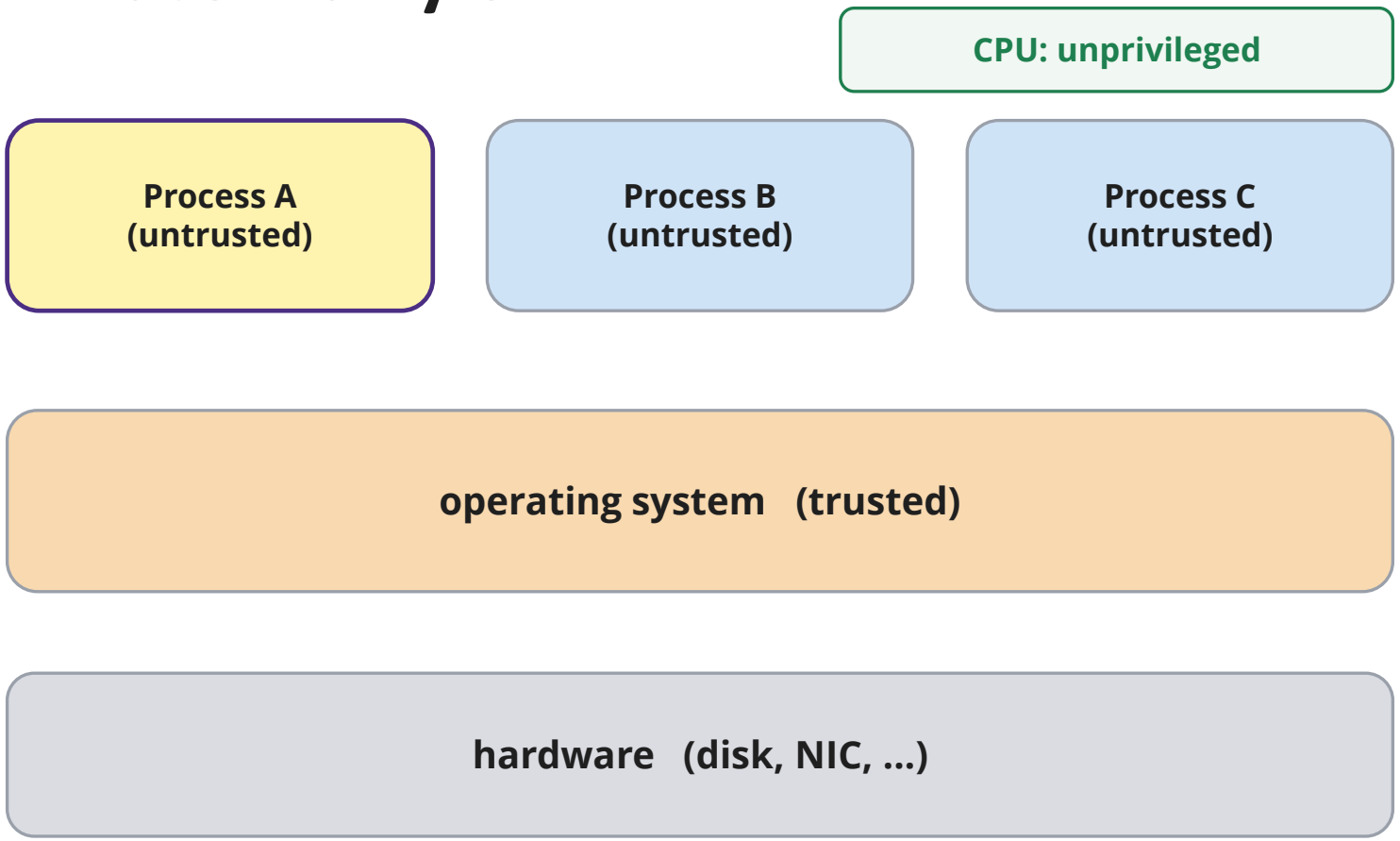
This is the slow part: hardware waits can be long, so the OS may run another process meanwhile.

System call trace: Return to your code



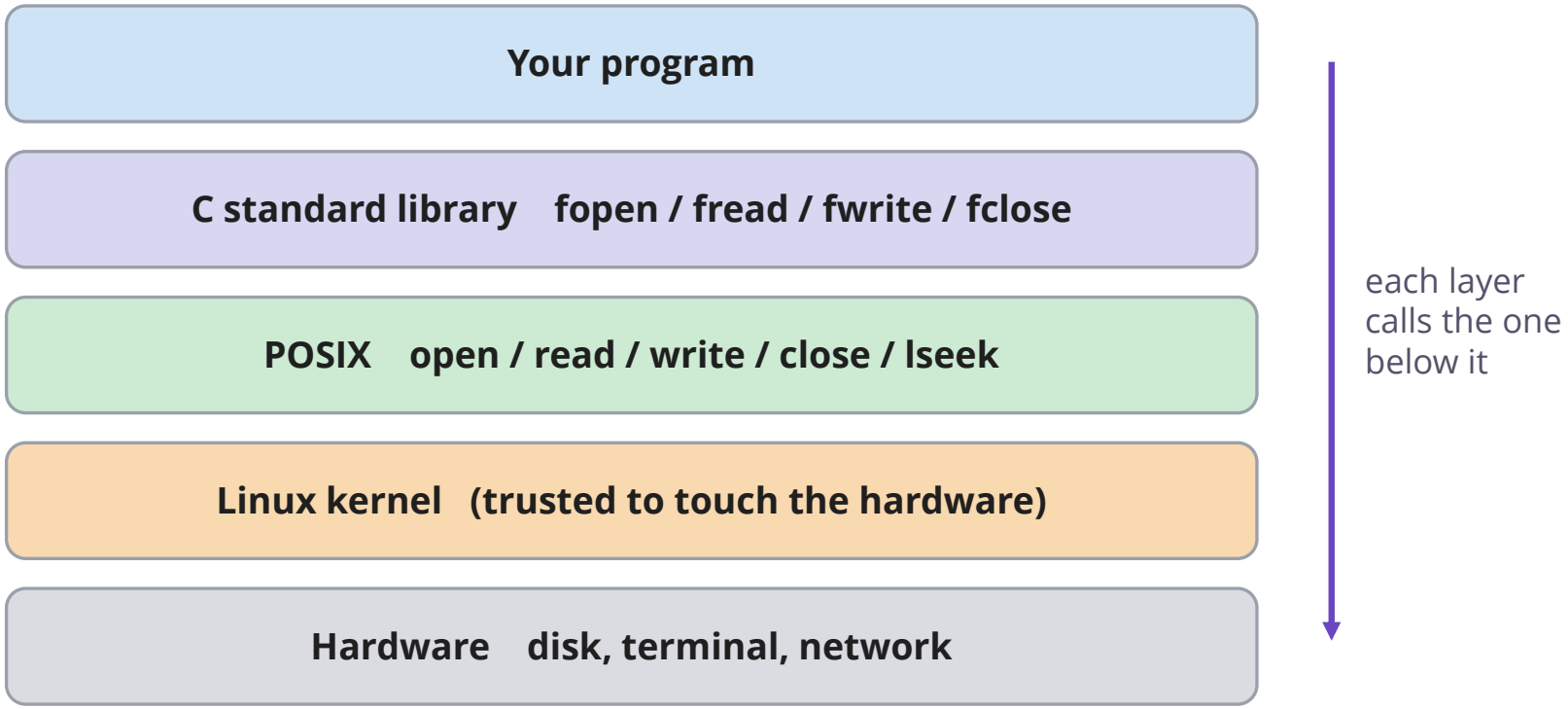
When the handler finishes (possibly after a long hardware wait), the OS sets the CPU back to unprivileged mode and returns into Process A's code.

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.

The I/O software stack



Intro to C++

Today: get to hello world, and contrast with C

- **Goal:** a first, practical look at C++, enough to read and write it. Not mastery, and not objects yet
- We write the smallest C++ program and line it up against the C you already know
- Next week, we will dive deeper into the tricker semantics of C++

Hello World in C

```c
#include <stdio.h>    // for printf
#include <stdlib.h>   // for EXIT_SUCCESS

int main(int argc, char** argv) {
  printf("Hello, World!\n");
  return EXIT_SUCCESS;
}
```

```
$ gcc -Wall -g -std=c17 -o hello hello.c
```

- The C you already know, compiled with gcc
- **Reminder:** that printf becomes a write system call
- Now let's write the exact same thing in C++

Hello World in C++

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

```
$ g++ -Wall -g -std=c++17 -o hello hello.cc
```

❖ **What changed?**

Hello World in C++

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

```
$ g++ -Wall -g -std=c++17 -o hello hello.cc
```

- **Compile with g++**, and the file is **.cc**
- **Different include** (iostream), and a new-looking output line
- Same program, so let's compare the two side by side

C vs C++ at a glance

	C	C++
compiler	`gcc`	`g++`
source file	`hello.c`	`hello.cc`
I/O header	`#include <stdio.h>`	`#include <iostream>`
print	`printf("%d\n", x);`	`cout << x << endl;`
read	`scanf("%d", &x);`	`cin >> x;`
strings	`char* + <string.h>`	`std::string`
booleans	`int flags / macros`	`bool  (built in)`

cout: printing in C++

```
1  int x = 5;
2  std::cout << "x is " << x << std::endl;
3  // chain values with <<, left to right
4  // endl adds a newline and flushes
```

- **cout** is C++'s stdout, in namespace **std**
- Chain as many values as you like with <<
- No % format string to get wrong

In C, you had to...

- **printf("%d %s", n, name);**
- Pick the right %-specifier for each type
- **Wrong specifier =** garbage or a crash

In C++, you get...

- **cout << n << " " << name;**
- No format string at all
- **Type-safe:** the right output for each type

std::string: a real string type

```
1  #include <string>
2  string s("Hello");   // construct from a C-string
3  s += ", World!";     // append; it grows for you
4  cout << s.length();   // 13: it knows its own size
```

- **Include <string>**
- Grows automatically, manages its own memory
- Supports +, +=, ==, and .length()

In C, you had to...

- **char buf[64];**
- strcpy / strcat into a fixed buffer
- **Overflow risk,** and you track the '\0' yourself

In C++, you get...

- **string s = "Hello";**
- s += ", World!"; just works
- **No overflow,** no manual terminator

Reading input, and a real bool

```
1  int num;
2  cin >> num;                // read an int
3  if (cin >> num) { ... }    // true if it worked
4  string line;
5  getline(cin, line);        // read a whole line
```

- **cin** with **>>** reads input
- Use (cin >> x) as a condition to loop safely
- **Useful for Exercise 8;** bool is a real type now, use it

In C, you had to...

- **scanf("%d", &num);**
- Remember the &, and a format string
- **int flags** for true/false

In C++, you get...

- **cin >> num;**
- No &, and it can be a loop condition
- **bool** with true / false built in

Namespaces and using

```
1  std::cout << "Hello" << std::endl;   // fully qualified
2
3  using namespace std;                // import ALL of std
4  using std::cout;                    // import just cout
5  cout << "Hello" << endl;             // now shorter
```

- **The standard library lives in namespace std**
- **using** lets you drop the std:: prefix
- **Style tip:** in headers, import only what you use, not the whole namespace

Stream manipulators (formatting)

```
1  #include <iomanip>    // for setw, hex, dec
2
3  cout << setw(4) << 5 << endl;   // "   5"  (width 4)
4  cout << hex << 255 << endl;     // "ff"  (base stays hex)
5  cout << dec << 255 << endl;     // "255" (back to base 10)
```

- **setw(x):** width of the NEXT field only (not persistent)
- **hex / dec / oct:** number base, sticks until you change it
- From <iomanip>

Surprisingly complex!

```
1  #include <iostream>   // for cout, endl
2  #include <cstdlib>    // for EXIT_SUCCESS
3
4  int main(int argc, char** argv) {
5    std::cout << "Hello, World!" << std::endl;
6    return EXIT_SUCCESS;
7  }
```

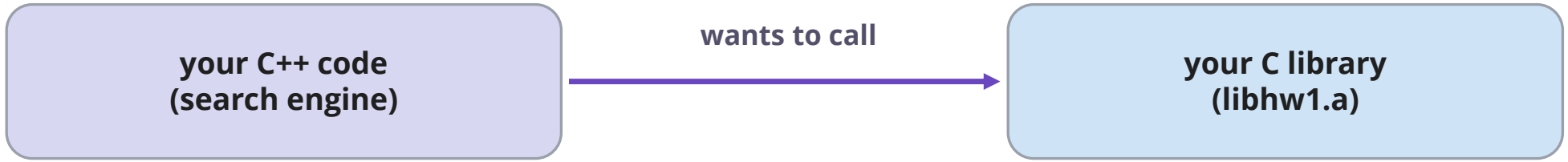
- **That one cout line is not simple:** cout is an object, and << is an overloaded operator
- C++ hides a lot behind friendly syntax
- **We learn how it works** in the classes lectures. For now, just use it

Takeaway: When C++ looks simple, remember how much is going on under one line

Interoperability

You already have C you want to reuse

- **C is roughly a subset of C++,** so it is tempting to just call your C code from C++
- **Homework 2:** your C++ search engine links against libhw1.a, the LinkedList and HashTable you wrote in C
- So: can we take a C header, include it in a C++ file, and have it just work?



A tiny C library

arith.h

```
1  // two plain C functions
2  int Add(int a, int b);
3  int Max(int a, int b);
```

arith.c

```
1  #include "arith.h"
2
3  int Add(int a, int b) { return a + b; }
4  int Max(int a, int b) {
5    return a > b ? a : b;
6  }
```

- Nothing fancy: two ordinary C functions
- **Header arith.h** declares them; **arith.c** defines them
- This is exactly the shape of your HW1 code: a .h of declarations, a .c of definitions
- We will hold onto this header and try to use it two ways

Baseline: use it from C

use_c.c

```c
#include <stdio.h>
#include <stdlib.h>   // for EXIT_SUCCESS
#include "arith.h"

int main(int argc, char** argv) {
  printf("%d\n", Add(2, 3));
  return EXIT_SUCCESS;
}
```

```
$ gcc -o use_c use_c.c arith.c
$ ./use_c
  5
```

- Include the header, call Add, compile both .c files together
- **Works perfectly.** This is the world you already live in
- Now let's include that same header from a C++ file instead

Include the same header from C++

use.cc

```cpp
#include <iostream>
#include <cstdlib>
#include "arith.h"    // the SAME header
using namespace std;
int main(int argc, char** argv) {
  cout << Add(2, 3) << endl;
  return EXIT_SUCCESS;
}
```

- **It compiles, but fails to link**
- Same header, same call… why?
- **Short version:** C++ looked for a different symbol name than the one in arith.o

```
$ g++ -c use.cc
$ gcc -c arith.c
$ g++ -o use use.o arith.o
  use.o: undefined reference to `Add(int, int)'
```

The fix: extern "C" in the header

arith.h (guarded)

```
1  #ifndef ARITH_H_
2  #define ARITH_H_
3
4  #ifdef __cplusplus
5  extern "C" {          // C++ only
6  #endif
7
8  int Add(int a, int b);
9  int Max(int a, int b);
10
11 #ifdef __cplusplus
12 }
13 #endif
14
15 #endif  // ARITH_H_
```

- **extern "C"** tells C++: link these with plain C names
- **The __cplusplus** guard means:
  - a C++ compiler sees the extern "C" wrapper
  - a C compiler skips it (it is not valid C)
- **One header,** used unchanged by both gcc and g++

Rebuild: now it links

```
$ g++ -c use.cc      # sees extern "C", wants plain 'Add'
$ gcc -c arith.c     # defines plain 'Add'
$ g++ -o use use.o arith.o
$ ./use
  5
```

- **Each file is compiled by its own compiler:** arith.c with gcc, use.cc with g++
- Thanks to the guard, both agree on the symbol name Add, so the linker matches them
- **Link the object files together with g++** (let the C++ driver do the final link)

Takeaway: Guard the header once and the same C library links cleanly into both C and C++ programs.

Can we wrap every function?

No: only C-compatible functions

- No overloaded functions (two would collide on one C name)
- No function templates
- No class member functions

extern "C" fixes linkage, not types

- One unmangled symbol per name, that is all it gives you
- You still cannot hand a C function a std::string, a reference, or a class
- Parameters must be things C understands

- **Rule of thumb:** wrap plain C-style functions with C-style parameters. Like, your HW1!
- Going the other way (calling true C++ features from C) does not work, and that is fine for us

Reminders

Assessments:

- Exercise 4 released shortly today! Due on Thursday at 11:59pm

General:

- C++
- 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!