CS 241: Systems Programming
Lecture 33. Course Overview

Fall 2025

Stephen Checkoway

Announcements

Group project Final Report due Dec 19 at 9 a.m.

Group assessment Google form due Dec. 19 at 9 a.m.

Group project Presentation given May 19 from 9 a.m. to 11 a.m.

- Room: TBA (waiting on the registrar to assign a room)

Today’s Class

A look back at what we covered this semester, with clicker questions

Why do we use Rust?

A. Faster than languages like Python and Java that aren’t compiled to machine language

B. Offers memory safety guarantees that compiled languages like C and C++ don’t

C. Easier to write code in than scripting languages like Python

D. A and B

E. A, B and C

Why Rust?

- Systems language: Designed to interface with Operating System and networks and provide low-level access to memory
- Designed for memory-safety
- Has lots of cool packages called “crates”
- Jason Orendorff, a GitHub engineer who wrote a book on Rust:
    - “To me, what’s great about Rust is that it’s both fast AND reliable. It lets me write multi-headed programs that run on 16 cores and keep them readable, maintainable, and crash-free. It also lets me write very low-level algorithms requiring control over memory layout and pull in a crate that makes HTTPS requests super simple. It’s the combination of these features that makes Rust so unique.”

Review: Rust & Memory Safety

Rust ensures that program are memory safe, e.g.,

- It’s impossible to confuse a pointer with an integer
- It’s impossible to access out-of-bounds data in an array/Vec

Most modern languages (Python, Java, Go, Haskell, Ruby, etc.) are memory-safe

Most systems languages (C and C++) are not!

- Memory safety errors are common and lead to real harm

Ownership

Rust ensures memory safety through a concept of ownership

These are rules that the rust compiler enforces to prevent **undefined behavior**

What is true after this line of code?

```
let v = vec![1,2,3]
```

A. The stack contains memory holding the values 1, 2, 3

B. The stack contains memory holding the values 1, 2, 3, and a pointer to that memory, v

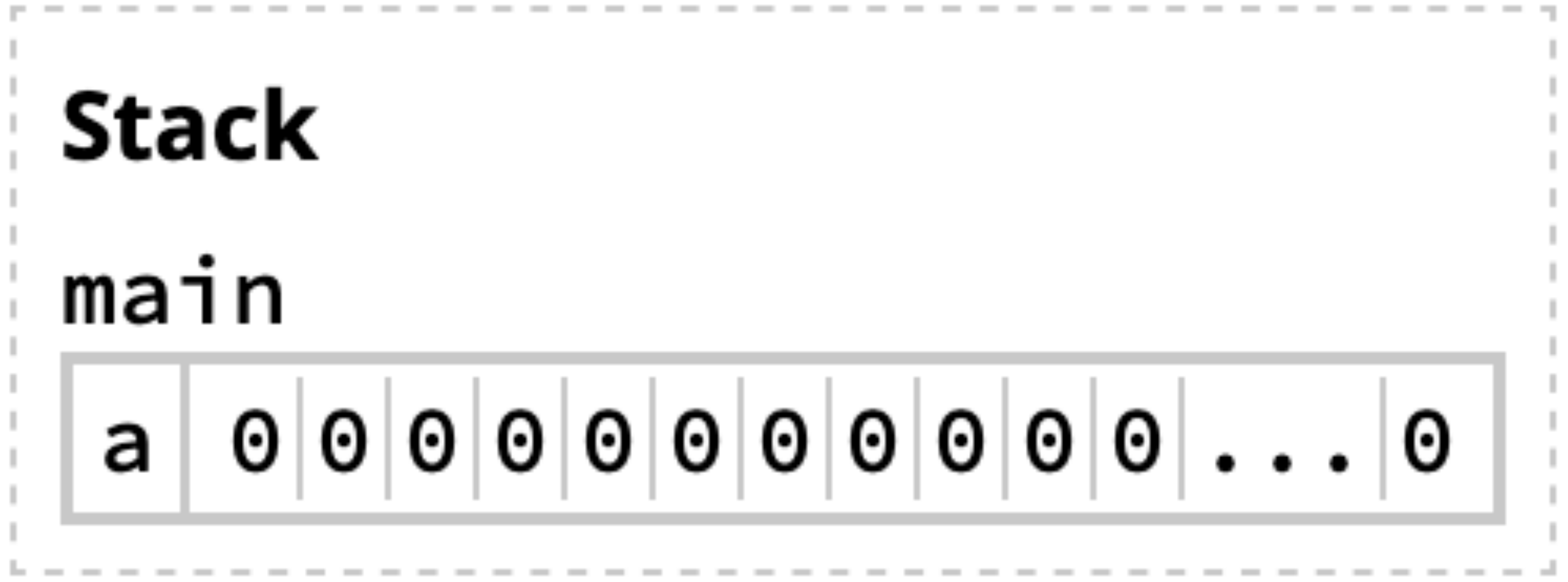
C. The heap contains memory holding the values 1, 2, 3, and a pointer to that memory, v

D. The heap contains memory holding the values 1, 2, 3, and the stack contains a pointer to that memory, v

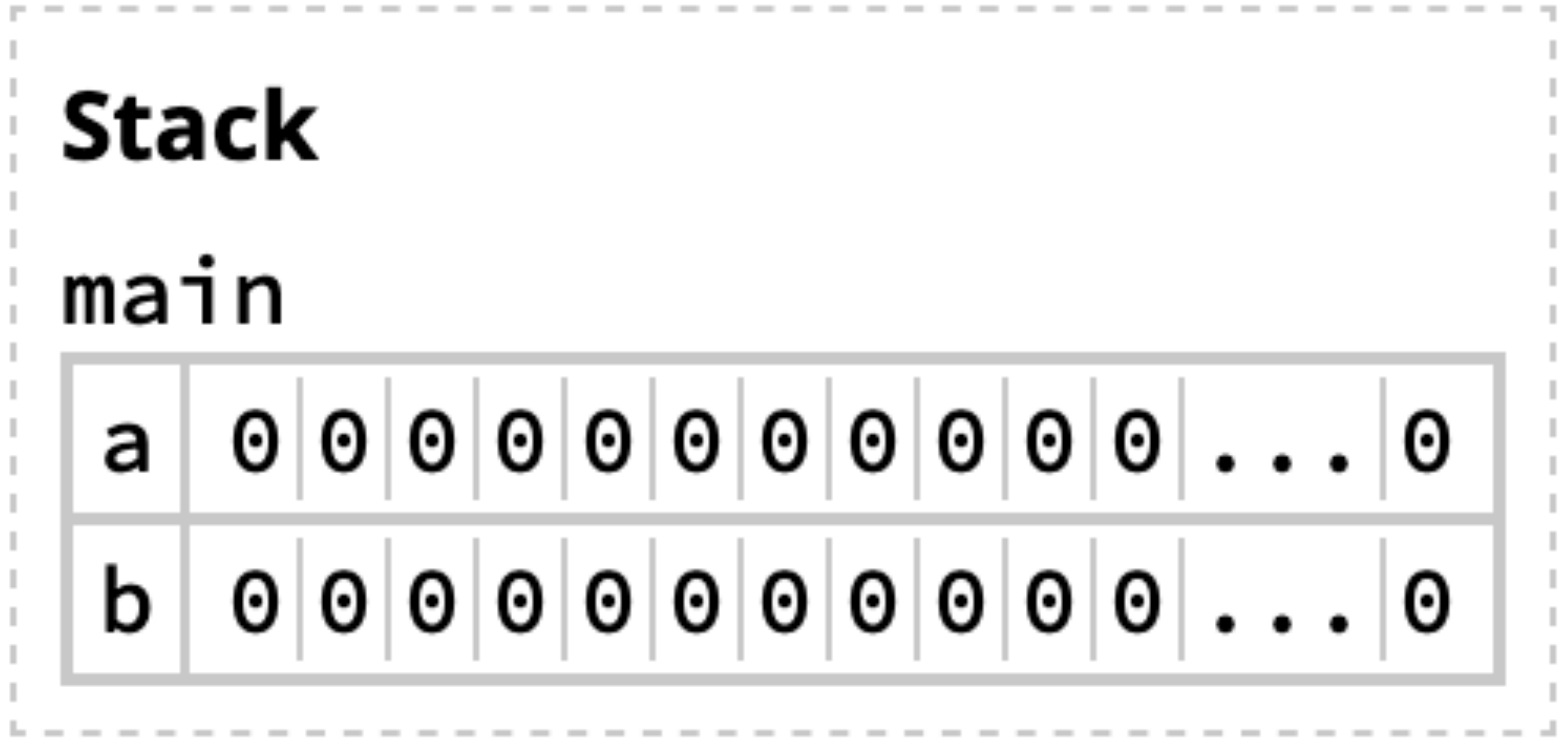
Data on the stack vs. heap

```
let a = [0; 1_000_000]; L1
let b = a; L2
```

L1

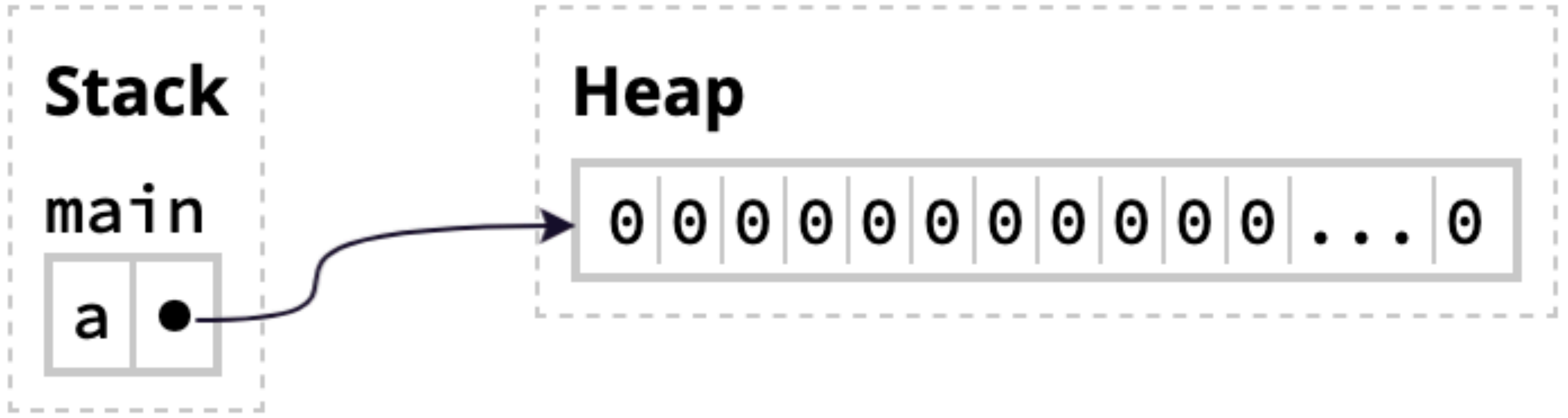


L2

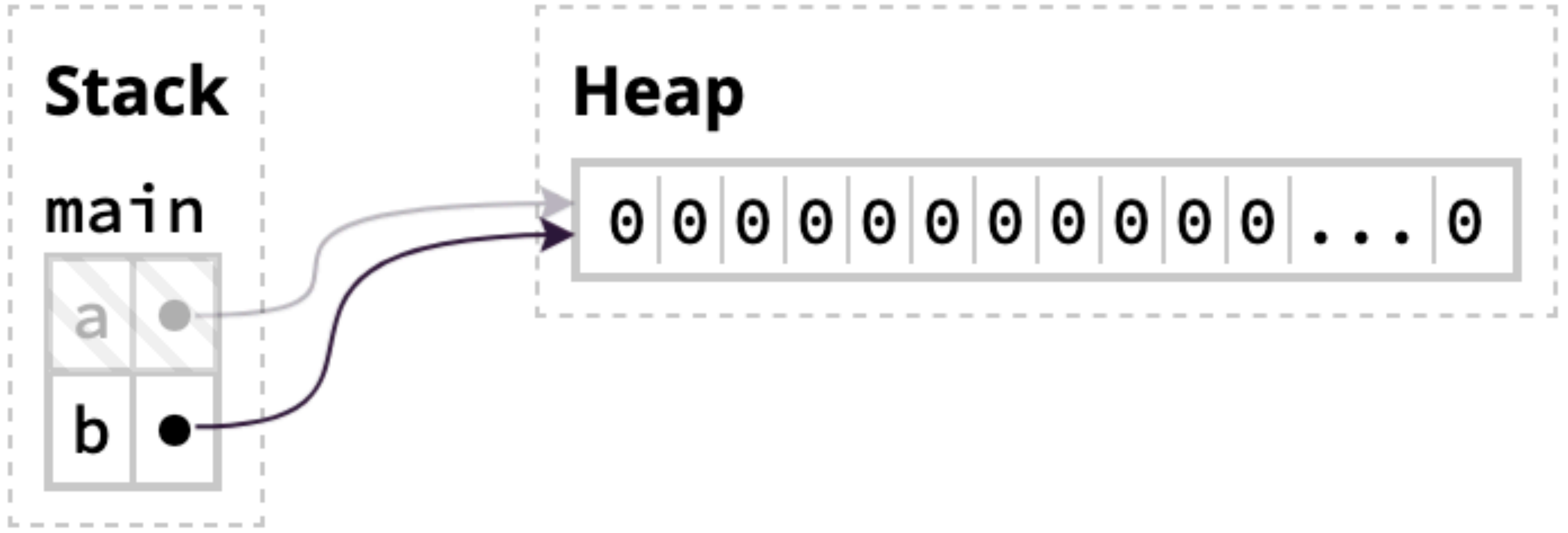


```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```

L1



L2



No use-after-free

A common vulnerability in C and C++ code is

- Allocate some heap memory
- Free the allocated memory
- Use the freed memory; this is **undefined behavior!**

In Rust, that might look something like

```rust
let b = Box::new(10);
drop(b); // Frees the allocated memory
println!("{b}");
```

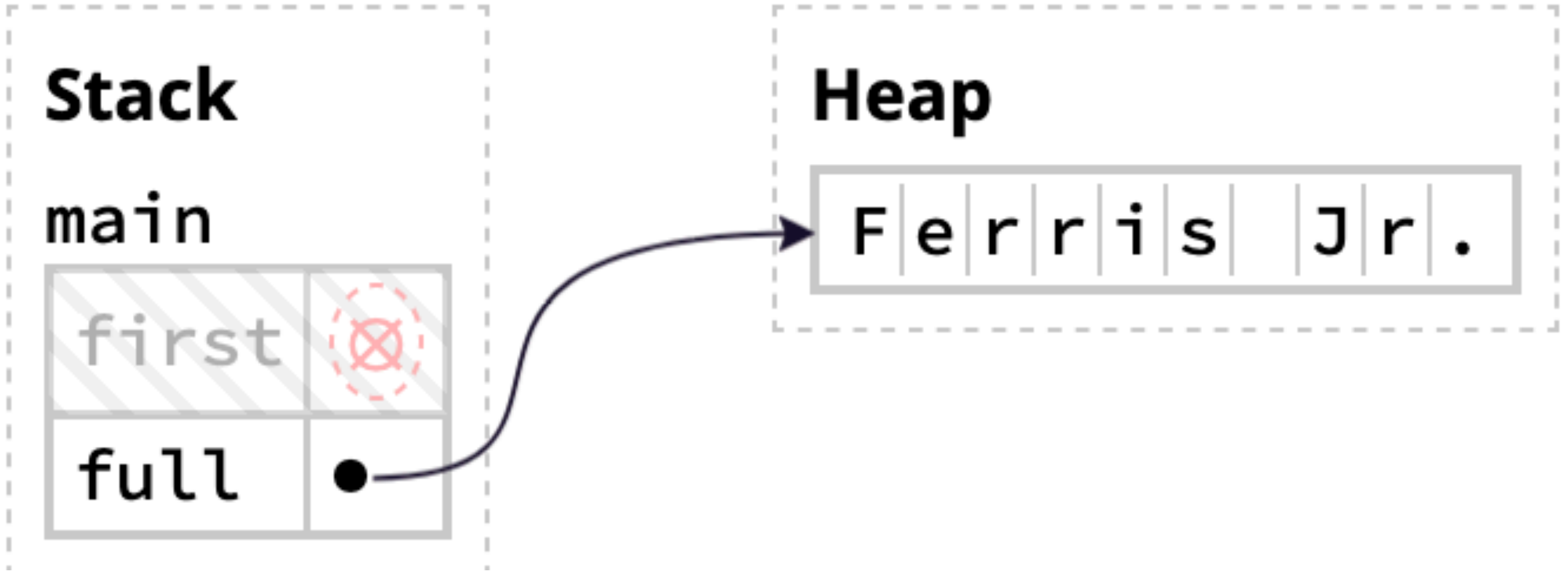
Rust gives a compile time error

Cannot use a variable after moving it

```
fn main() {
    let first = String::from("Ferris");
    let full = add_suffix(first);
    println!("{full}, originally {first}"); L1  // first is now used here
}

fn add_suffix(mut name: String) -> String {
    name.push_str(" Jr.");
    name
}
```

L1 undefined behavior: pointer used after its pointee is freed



Appending the string " Jr." causes the string to be reallocated

If we could continue to access first, it would point to freed memory! **Undefined behavior!**

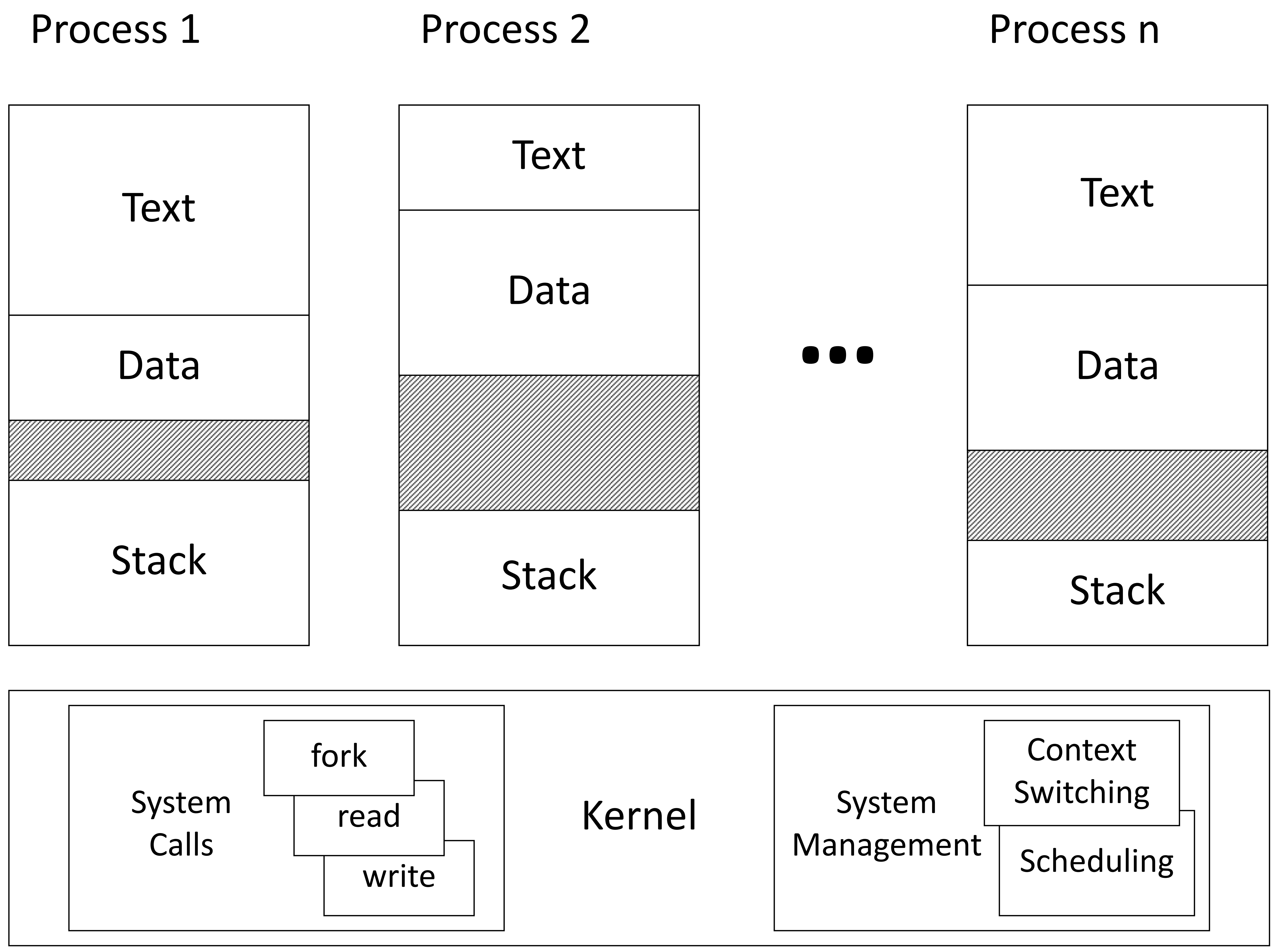
```rust
fn foo(s: String) { /* . . .*/ }

fn main() {
    let clickers = String::from("Clickers!");
    foo(XXX); // <-- Here
    println!("{clickers}");
}
```

What should we replace XXX with to pass the clickers string to `foo()`?

A. clickers
B. &clickers
C. clickers.clone()
D. More than one of the above

Review: Processes and the Kernel



Why do we have the kernel control switching which process is on the CPU, instead of the processes themselves?

A. It would be too slow

B. They could refuse to give up the CPU

C. They don’t have enough information about other processes

D. More than one of the above

Operating system tasks

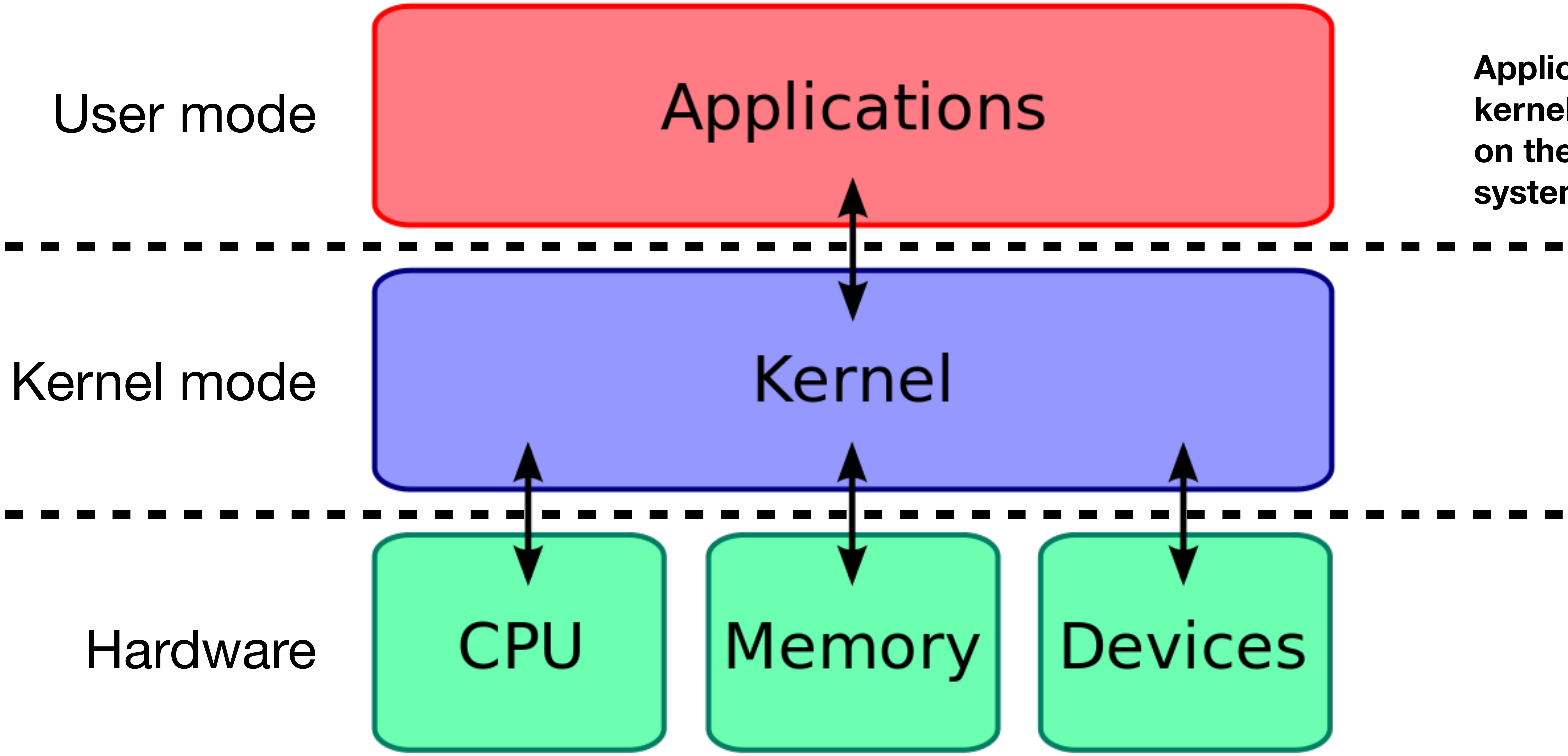
Managing the resources of a computer

- CPU, memory, network, etc.

Coordinate the running of all other programs

OS can be considered as a set of programs

- kernel – name given to the core OS “program”



Applications request the kernel perform an action on their behalf using system calls

https://en.wikipedia.org

Review: System calls

Programs talk to the OS via system calls

- Set of functions to request access to resources of the machine
- System calls vary by operating system and computer architecture

Types of system calls

- Input/output (may be terminal, network, or file I/O)
- File system manipulation (e.g., creating/deleting files/directories)
- Process control (e.g., process creation/termination)
- Resource allocation (e.g., memory)
- Device management (e.g., talking to USB devices)
- Inter-process communication (e.g., pipes and sockets)
- …

File Operation System Calls

`Open` – tells the kernel a process would like to interact with an I/O device/ file.

`Seek` – changes where in the file you are reading/writing

`Read` - copies bytes from a file into process memory

`Write` – copies bytes from process memory into a file

`Close` – tells the kernel we are done using a file

The read and write system calls return the number of bytes read/written. In what scenario would we return less than the number of bytes we asked to read/write?

A. Reading until the end of the file

B. Writing to the end of the file

C. Reading from a network

D. Writing to a network

E. More than one of the above

```
ssize_t write(int fildes, void const *buf, size_t nbyte);
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Creating a new process

Two schools of thought

- Windows way: single system call
  - `CreateProcess("calc.exe", /* other params */)`
- Unix way: two (or more) system calls
  - Create a copy of the currently running process: `fork()`
  - Transform the copy into a new process:
    `execve("/usr/bin/bc", args, env)`

What will print after running this code if the child’s PID is 5? Include output from all processes

```
int child_pid = fork();

if (child_pid == 0) {
    printf(“Child is %d\n”, getpid());
} else {
    printf(“My child is %d\n”, child_pid);
}
```

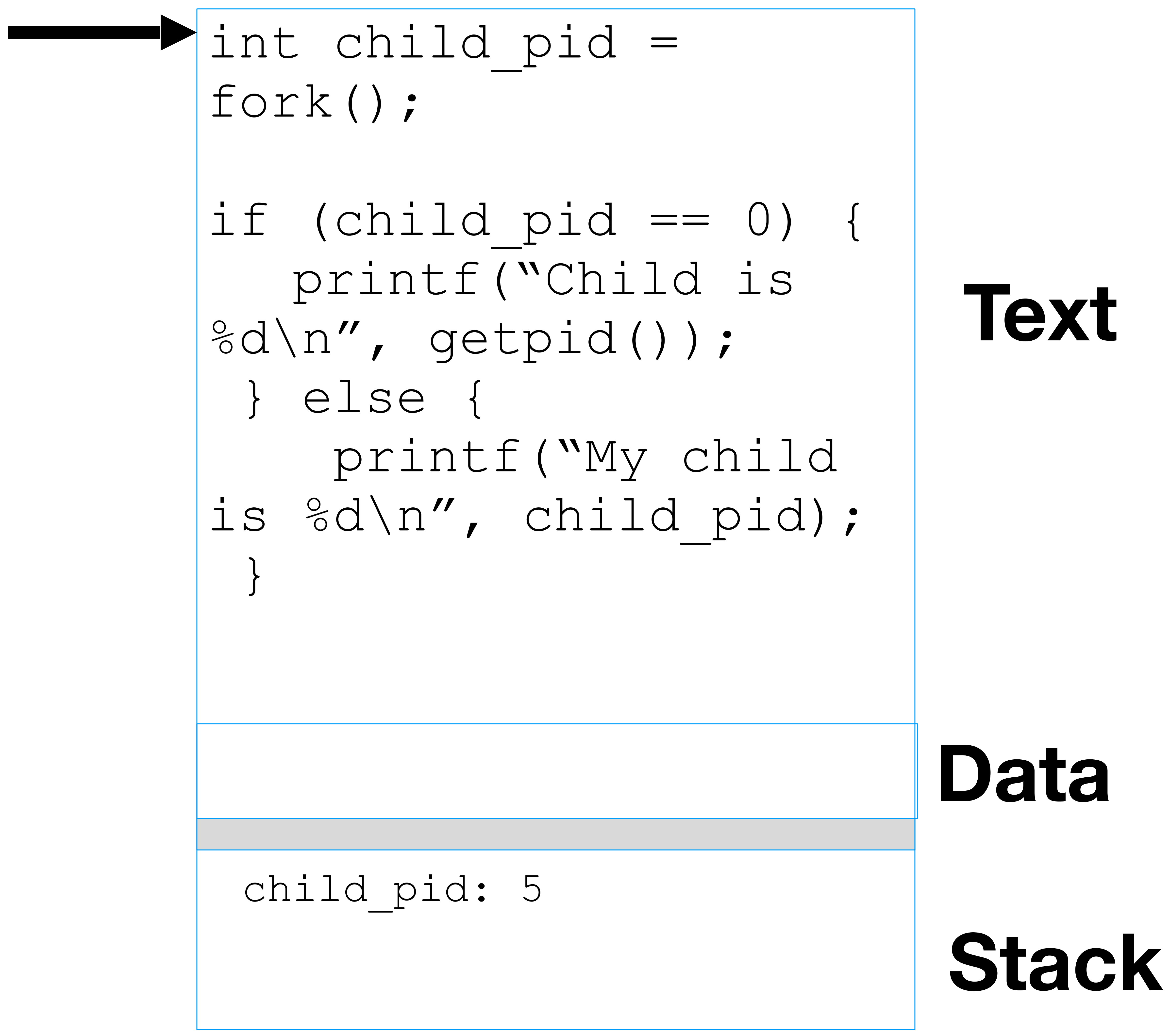
A. “Child is 5”

B. “My child is 5”

C. “My child is 0”

D. More than 1 of the above

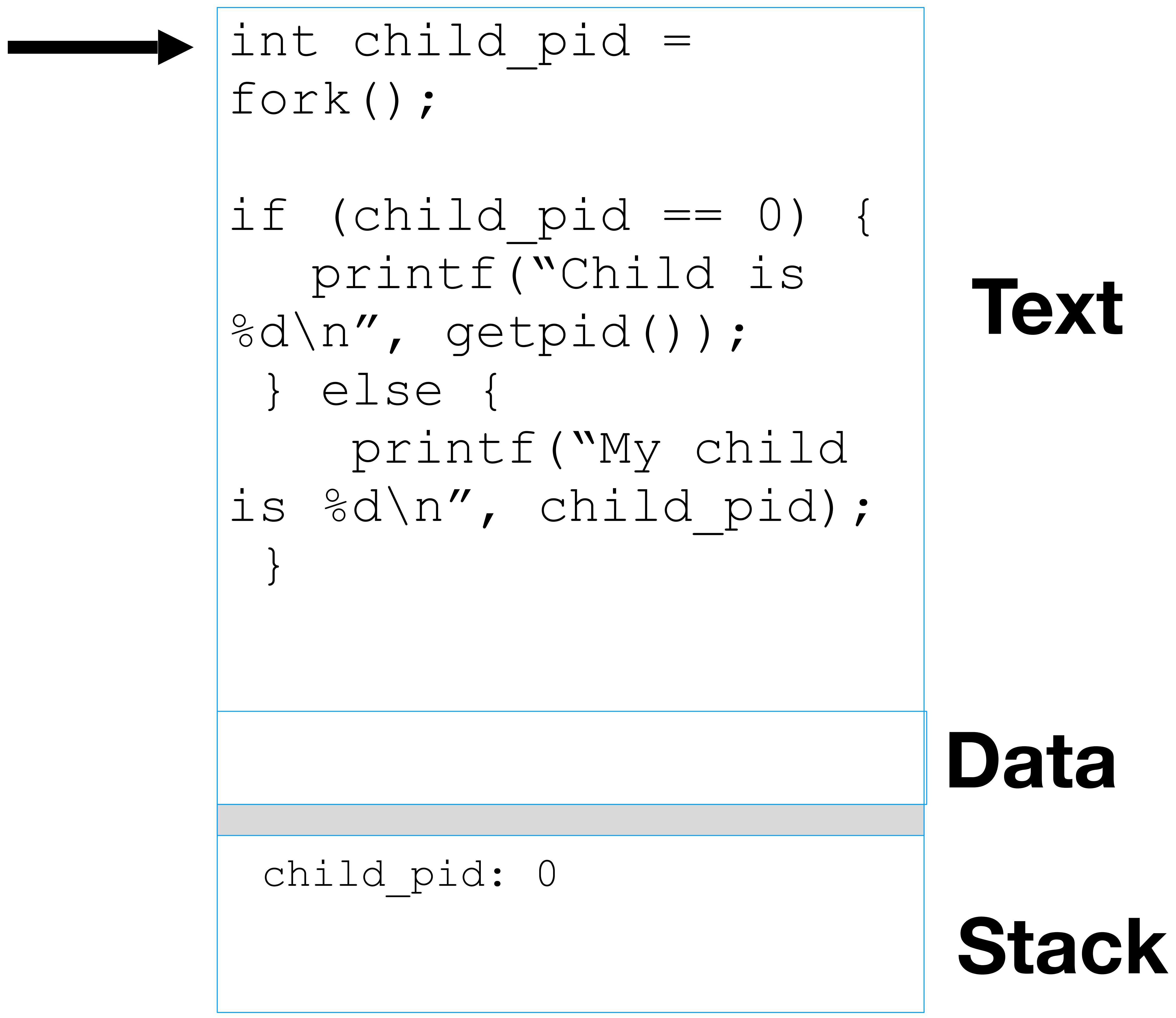
Fork



```
int child_pid =
fork();

if (child_pid == 0) {
   printf("Child is
%d\n", getpid());
 } else {
    printf("My child
is %d\n", child_pid);
 }
```

```
child_pid: 5
```

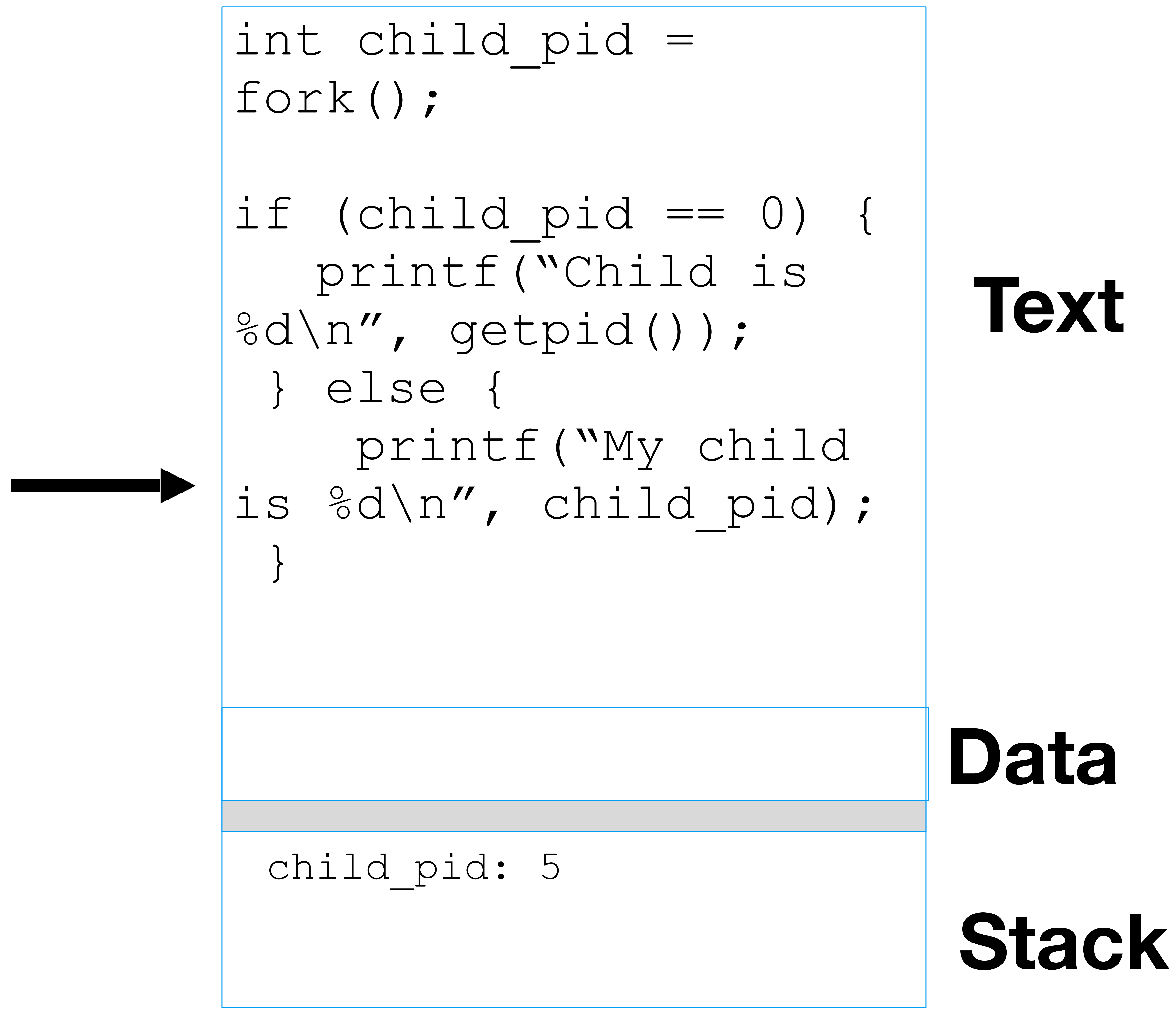


```
int child_pid =
fork();

if (child_pid == 0) {
   printf("Child is
%d\n", getpid());
 } else {
    printf("My child
is %d\n", child_pid);
 }
```

```
child_pid: 0
```

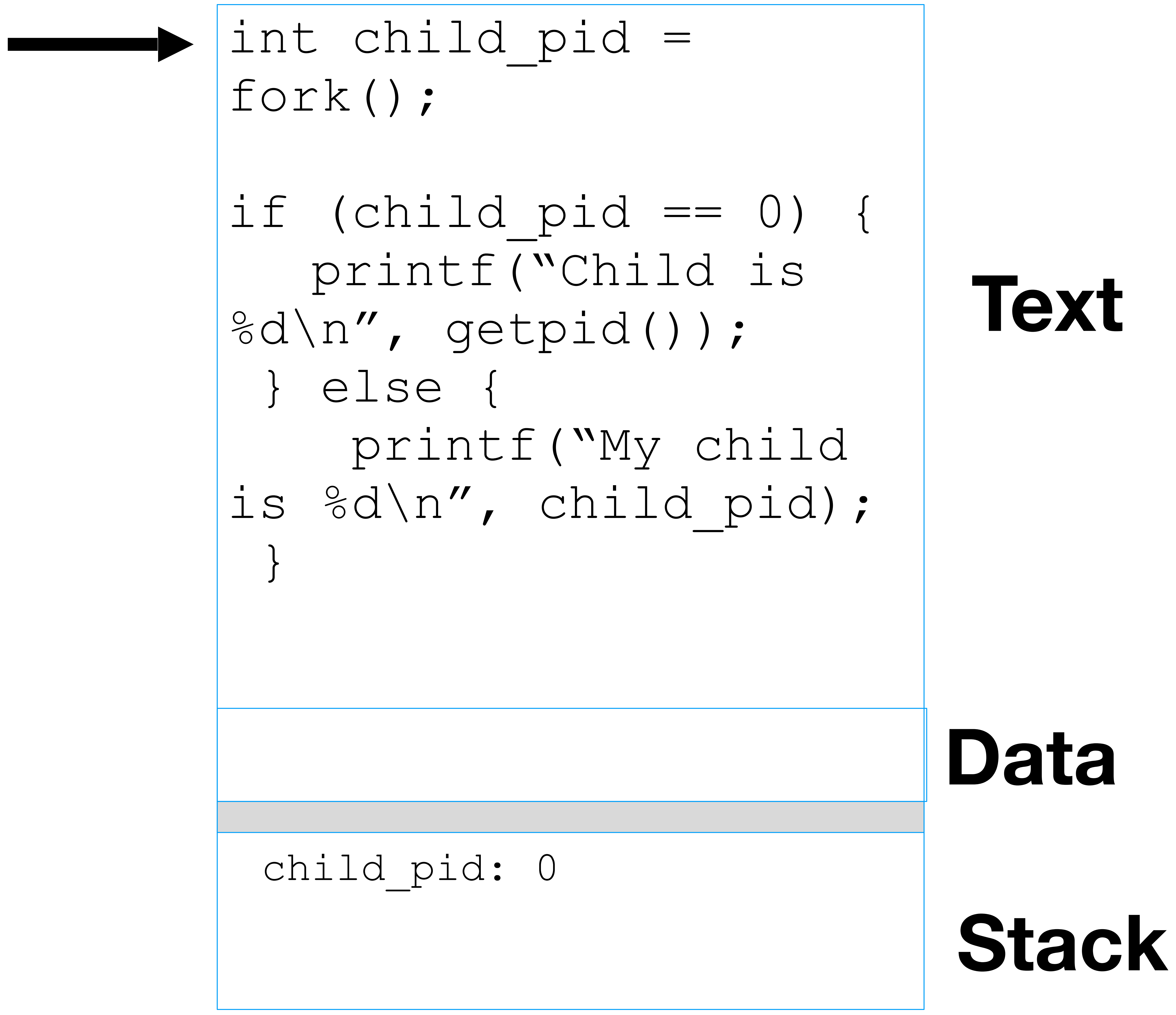
Fork



```
int child_pid =
fork();

if (child_pid == 0) {
   printf("Child is
%d\n", getpid());
 } else {
    printf("My child
is %d\n", child_pid);
 }
```

```
 child_pid: 5
```

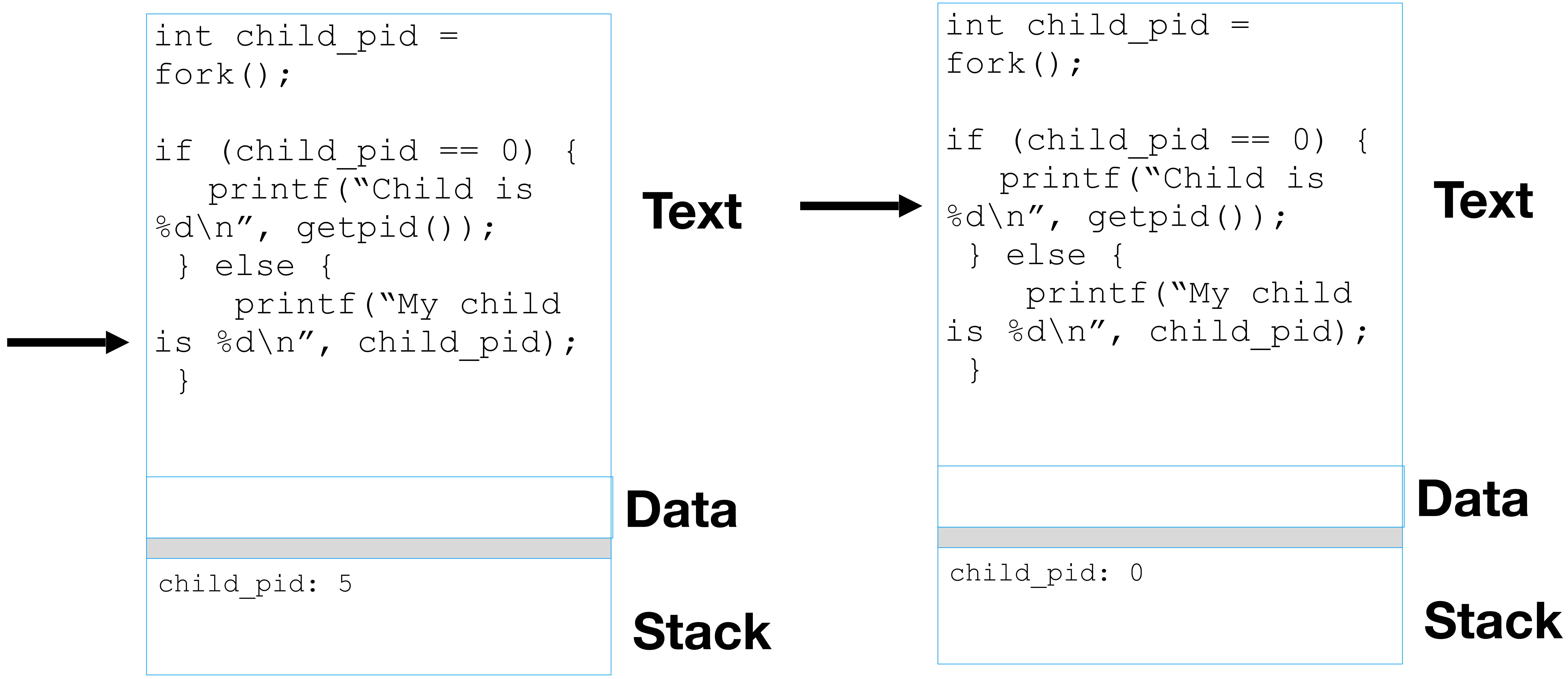


```
int child_pid =
fork();

if (child_pid == 0) {
   printf("Child is
%d\n", getpid());
 } else {
    printf("My child
is %d\n", child_pid);
 }
```

```
 child_pid: 0
```

Fork



```
int child_pid =
fork();

if (child_pid == 0) {
   printf("Child is
%d\n", getpid());
 } else {
    printf("My child
is %d\n", child_pid);
 }
```

```
int child_pid =
fork();

if (child_pid == 0) {
   printf("Child is
%d\n", getpid());
 } else {
    printf("My child
is %d\n", child_pid);
 }
```

What order will the two statements print in?

```
int child_pid = fork();

if (child_pid == 0) {
    printf("Child is %d\n", getpid());
} else {
    printf("My child is %d\n", child_pid);
}
```

A. First parent, then child

B. First child, then parent

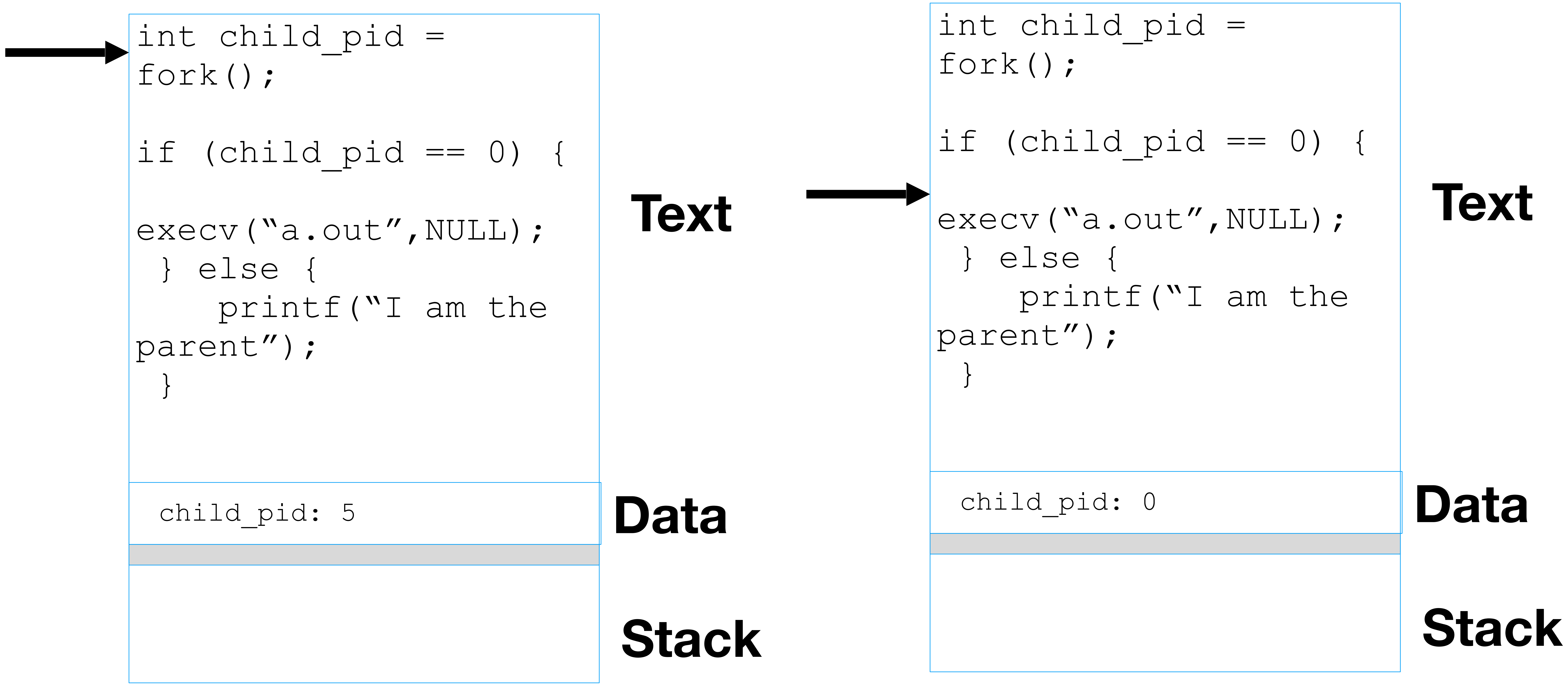
C. It depends

Running another program

```
int execve(char const *path, char *const argv[],
           char *const envp[]);
```

- The PID of the process doesn't change
- The open file descriptors remain open (unless marked close on exec)
- Returns `-1` and sets **`errno`** on error

Exec



```
int child_pid =
fork();

if (child_pid == 0) {

execv("a.out",NULL);
 } else {
    printf("I am the
parent");
 }
```

```
child_pid: 5
```

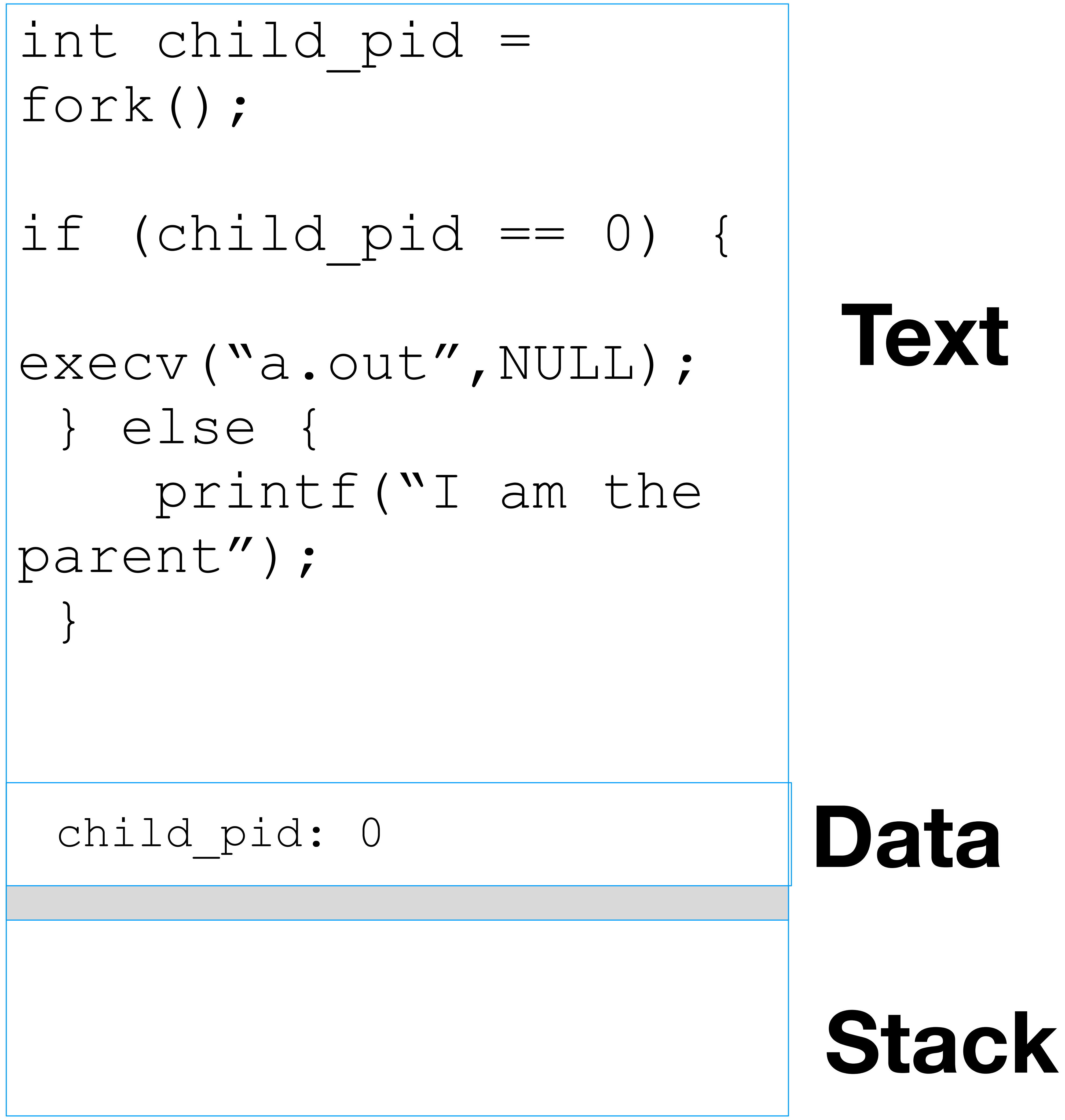
```
int child_pid =
fork();

if (child_pid == 0) {

execv("a.out",NULL);
 } else {
    printf("I am the
parent");
 }
```

```
child_pid: 0
```

If there are no errors, exec will return



```
int child_pid =
fork();

if (child_pid == 0) {

execv("a.out",NULL);
 } else {
    printf("I am the
parent");
 }
```

```
child_pid: 0
```

A. Zero times

B. Once

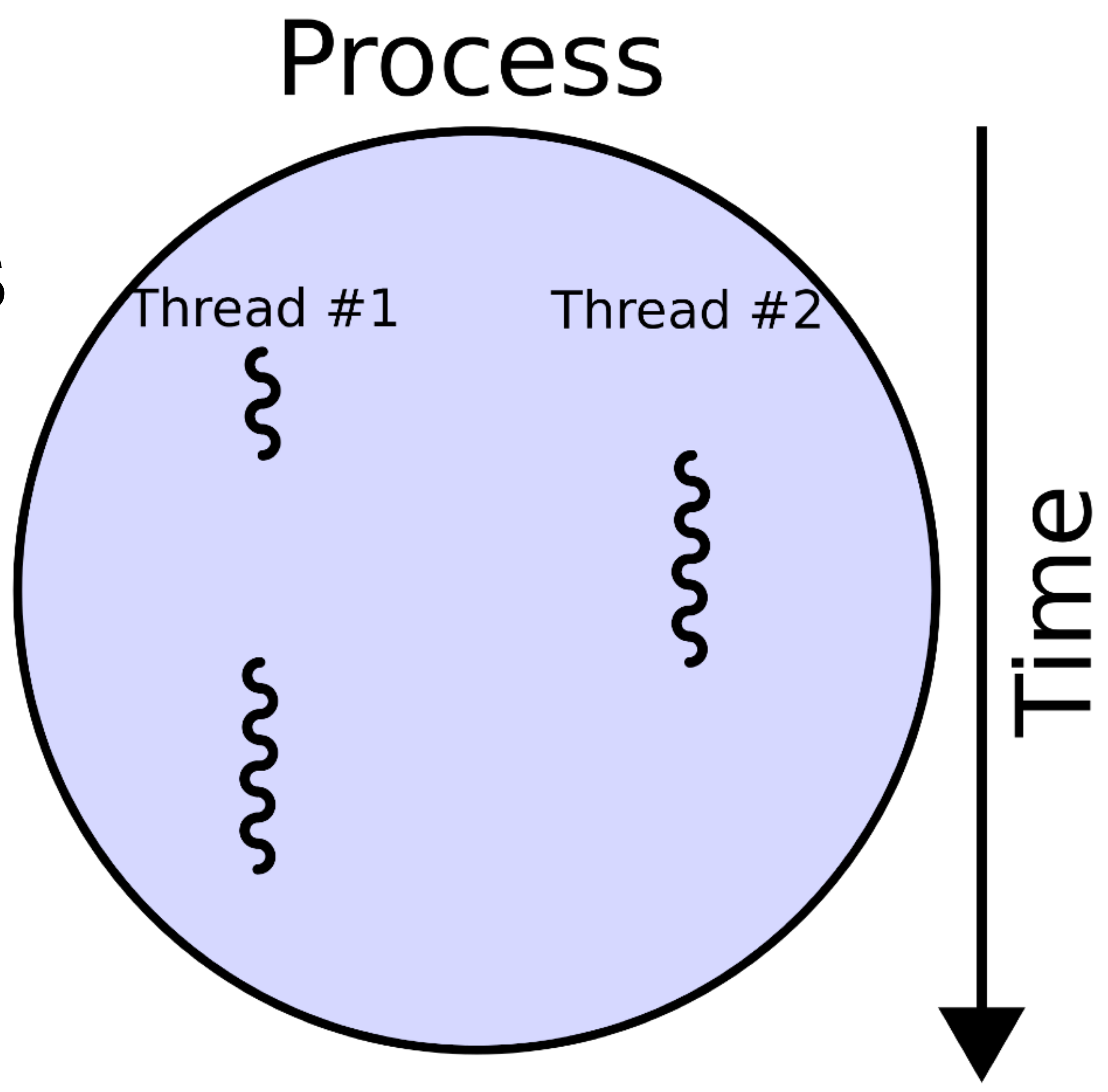
C. Twice

Review: Threads

A process may be composed of multiple **threads of execution**

Each thread runs concurrently with but independent of other threads in the process

Threads are a bit like cooperating processes inside a process



https://en.wikipedia.org/wiki/Thread_(computing)#/media/File:Multithreaded_process.svg

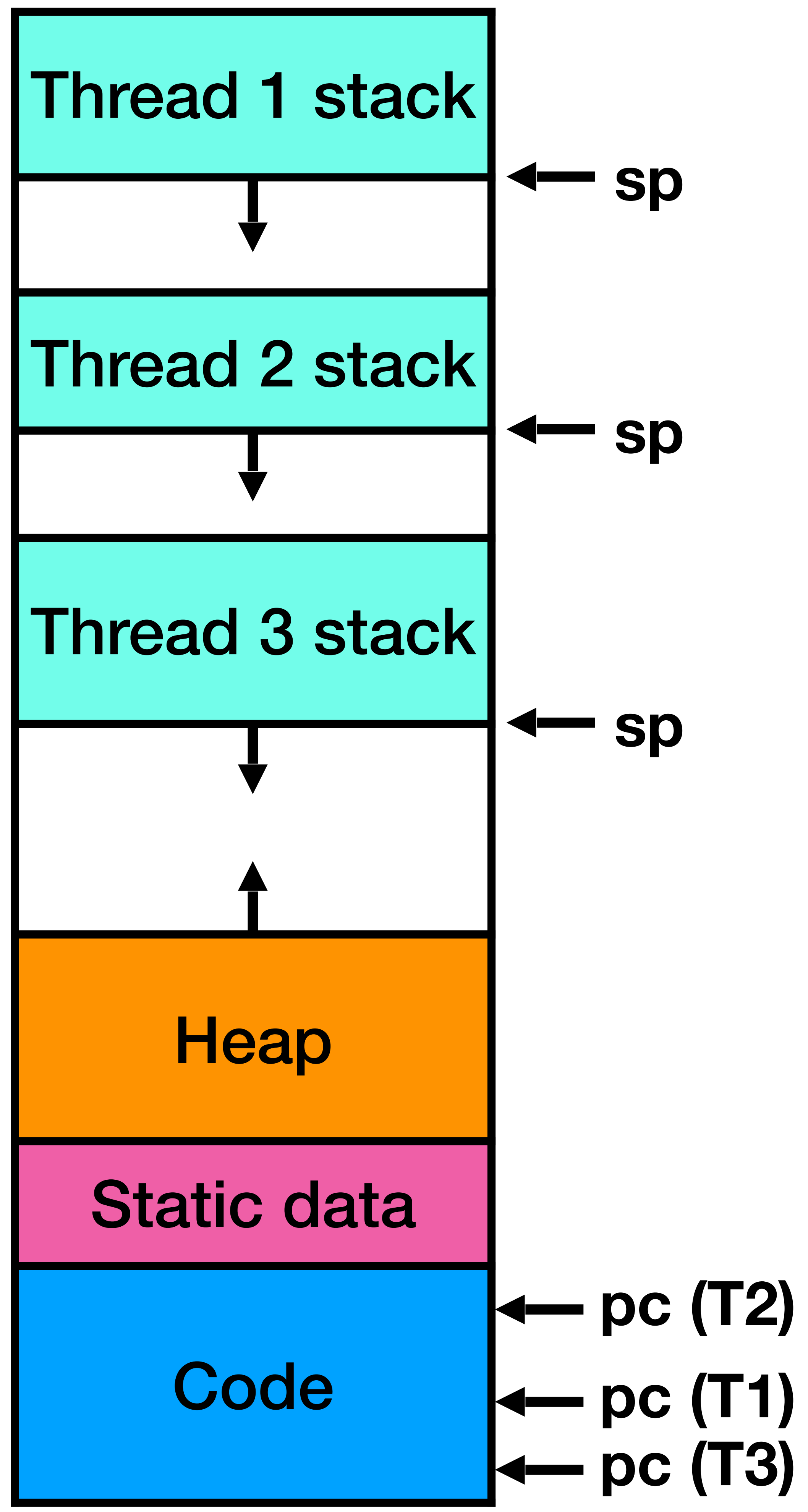
Relationship between threads

Each thread in a process shares all of the process's

- memory (data and code)
- open files
- permissions (e.g., to access the file system)
- user ID, group ID, process ID

Each thread in a process has its own

- function call stack with a stack pointer (sp)
- program counter (pc) indicating the next instruction to execute



Which of these is possible as the first five output lines from this code?

```rust
for thread_num in 0..10 {
    let t = thread::spawn(move || {
        for _ in 0..5 {
            println!("Hello from thread {thread_num}");
        }
    });
}
```

A:
```
Hello from thread 5
Hello from thread 6
Hello from thread 3
Hello from thread 4
Hello from thread 4
```

B:
```
Hello from thread 0
Hello from thread 0
Hello from thread 0
Hello from thread 0
Hello from thread 0
```

C:
```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 4
```

D. All of the above (and more!)

Review: Networks

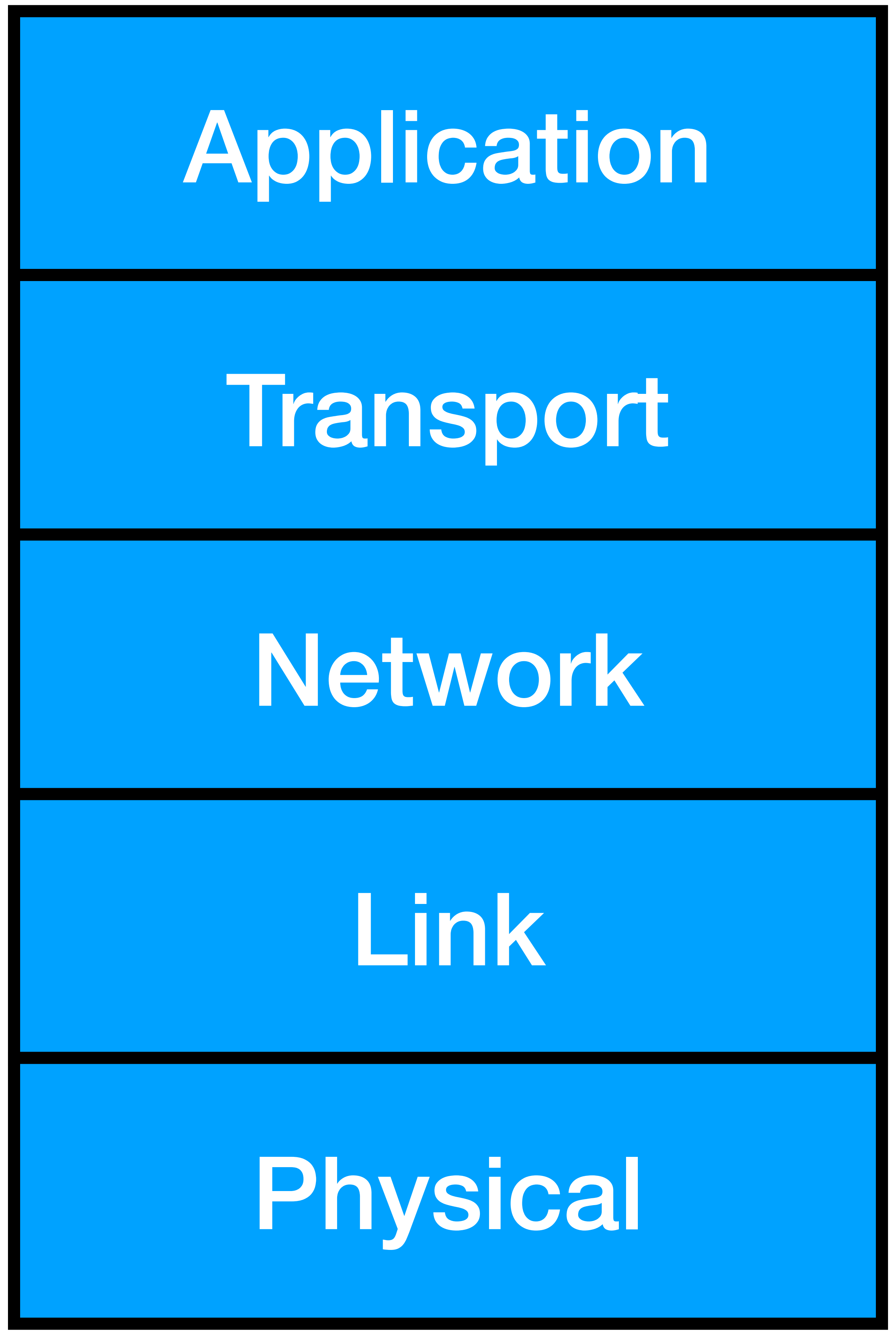
Application: supporting network applications
- e.g., HTTP

Transport: data transfer between processes on hosts
- e.g., TCP, UDP

Network: routing packets from source to destination
- e.g., IP

Link: data transfer between neighboring elements
- e.g., Ethernet, WiFi

Physical: transmit data over wires (or wireless signals)



TCP vs UDP

TCP: Transmission Control Protocol

TCP guarantees reliability

- All messages will get sent to the application, in order
- If a message gets lost, TCP will retransmit the message until it's received

TCP makes sure it doesn't overwhelm receiver by sending too much, too quickly

UDP: User Datagram Protocol

UDP does NOT guarantee reliability

- Messages may be lost or arrive out-of-order

Because UDP doesn't have to worry about reliability, it is much faster

For each of the following applications, choose whether you would use TCP or UDP, and justify why you would choose it. [Select any letter on your clicker]

A. Online gaming

B. SSH remote access

C. Email

D. Video conferencing

E. Whatsapp

We covered a lot this semester

Thank you for your work, and I look forward to seeing your final presentations!