Multiprocessing

CS110L
Feb 2 & Feb 7

Equivalent material is from 2020 and is available here directly or via Ryan's website.

Quick reflection: where are we in the class?

- Where we’ve been:
  - Memory safety
  - The *ownership* model that’s fundamental to Rust
  - Error handling in C, C++, and Rust
  - Learning enough Rust to build real, useful things
- Now:
  - Into 110 material
  - Concurrency, multiprocessing, threads
  - …and how this relates to safety in systems programming

Today and next week

- A break from Rust-land
  - Don’t call fork
  - Later (probably Monday): don’t call pipe and don’t call signal
- Multiprocessing syntax in Rust
- Intro to project 1, which involves everything we’ve done so far but with multiple processes!

fork()

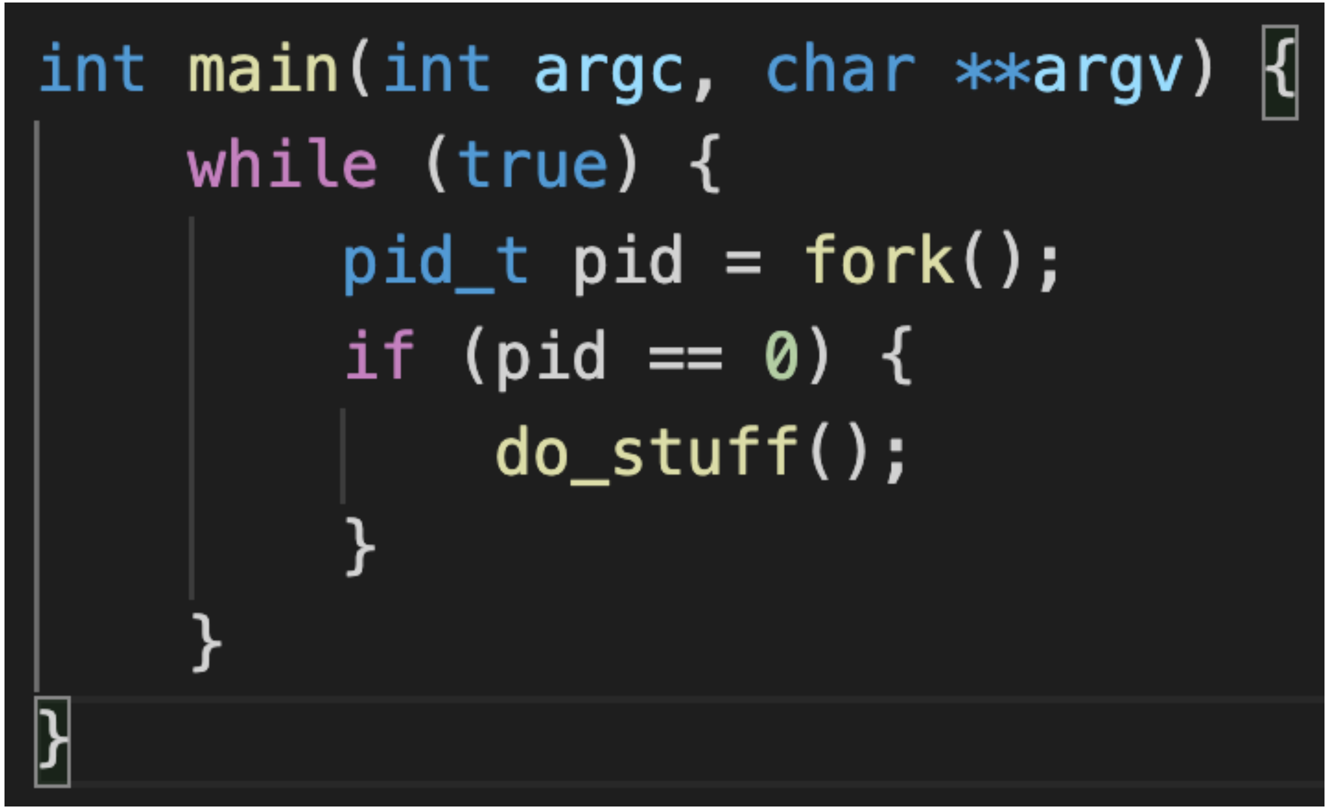
Why might you call fork? 🍴

- Get concurrent execution (i.e. run another piece of your own program at the same time)
- Invoke external functionality on the system (i.e. run a different executable)

Fork: what could go wrong?

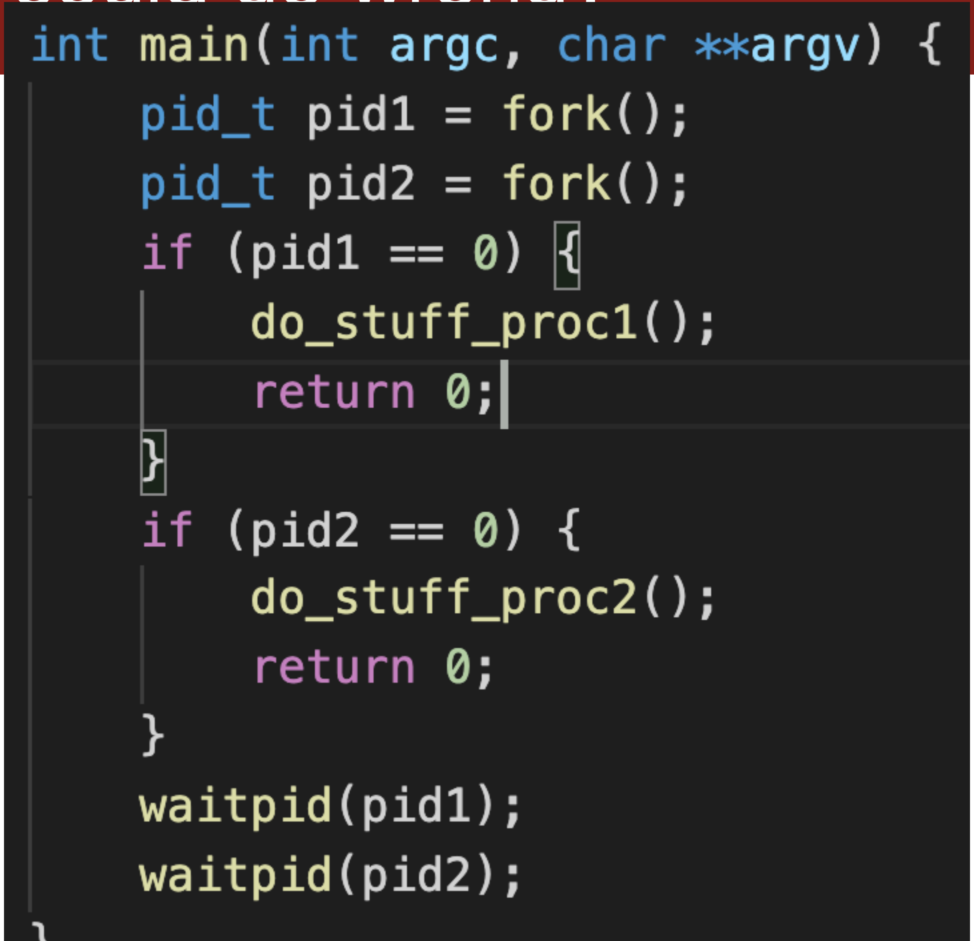
- Going to go through some code examples, which are all either incorrect or potentially dangerous depending on context
- Take ~15 seconds to think about them
- Imagine jeopardy music playing in the background
- **What could go wrong? Why does it matter?**

Fork: what could go wrong?



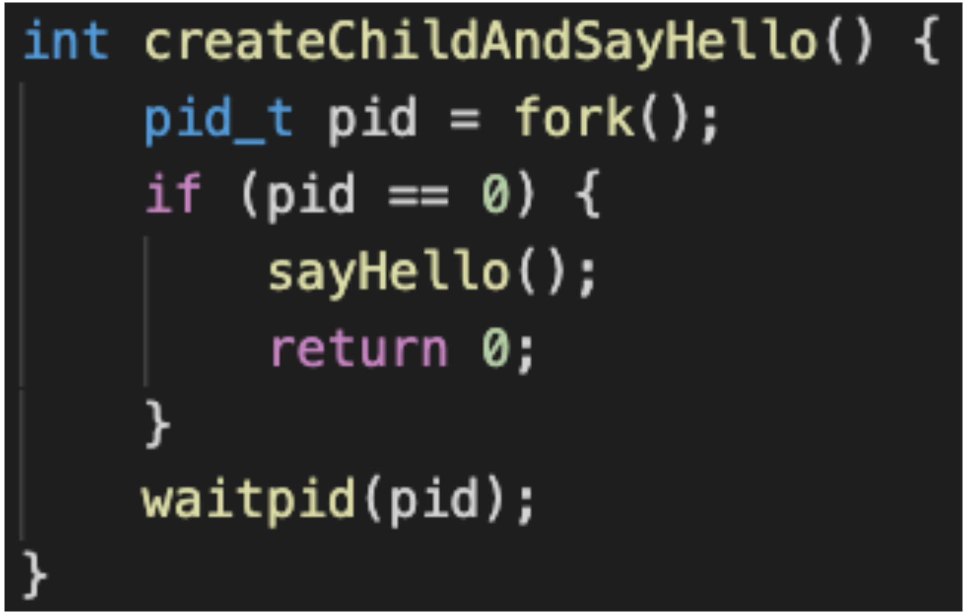
```
int main(int argc, char **argv) {
    while (true) {
        pid_t pid = fork();
        if (pid == 0) {
            do_stuff();
        }
    }
}
```

Fork: what could go wrong?



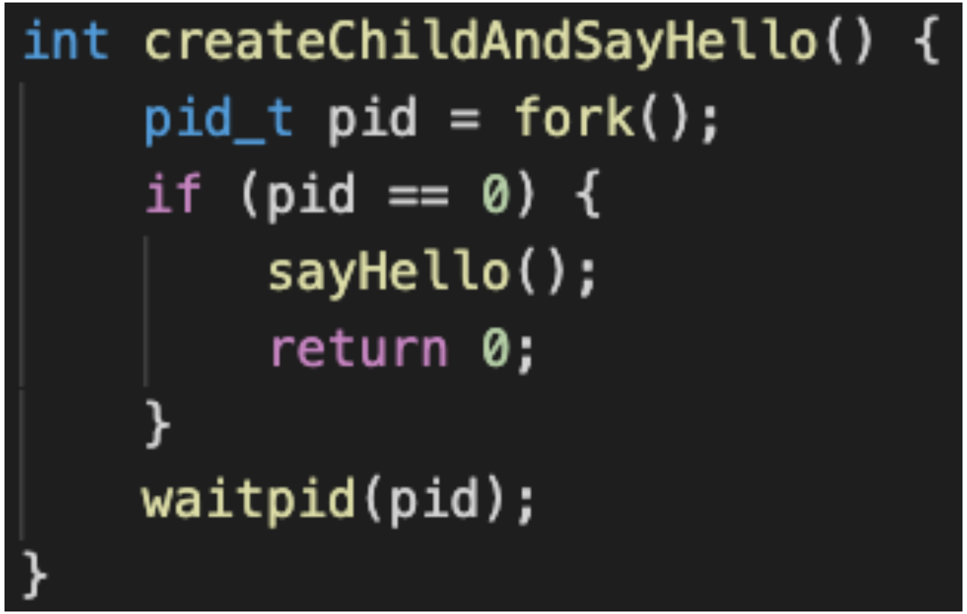
```
int main(int argc, char **argv) {
    pid_t pid1 = fork();
    pid_t pid2 = fork();
    if (pid1 == 0) {
        do_stuff_proc1();
        return 0;
    }
    if (pid2 == 0) {
        do_stuff_proc2();
        return 0;
    }
    waitpid(pid1);
    waitpid(pid2);
```

Fork: what could go wrong?



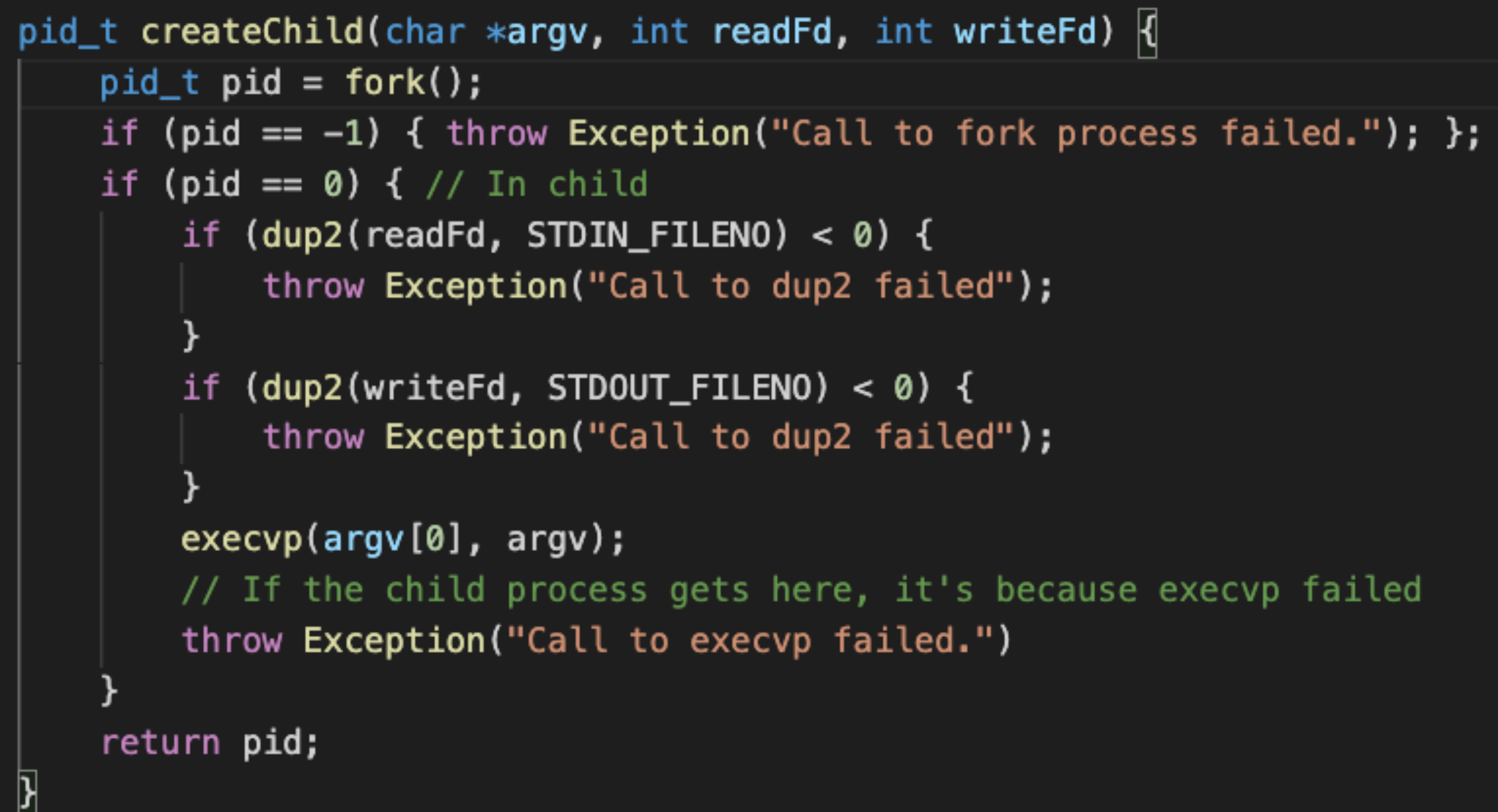
```
int createChildAndSayHello() {
    pid_t pid = fork();
    if (pid == 0) {
        sayHello();
        return 0;
    }
    waitpid(pid);
}
```

Fork: what could go wrong?

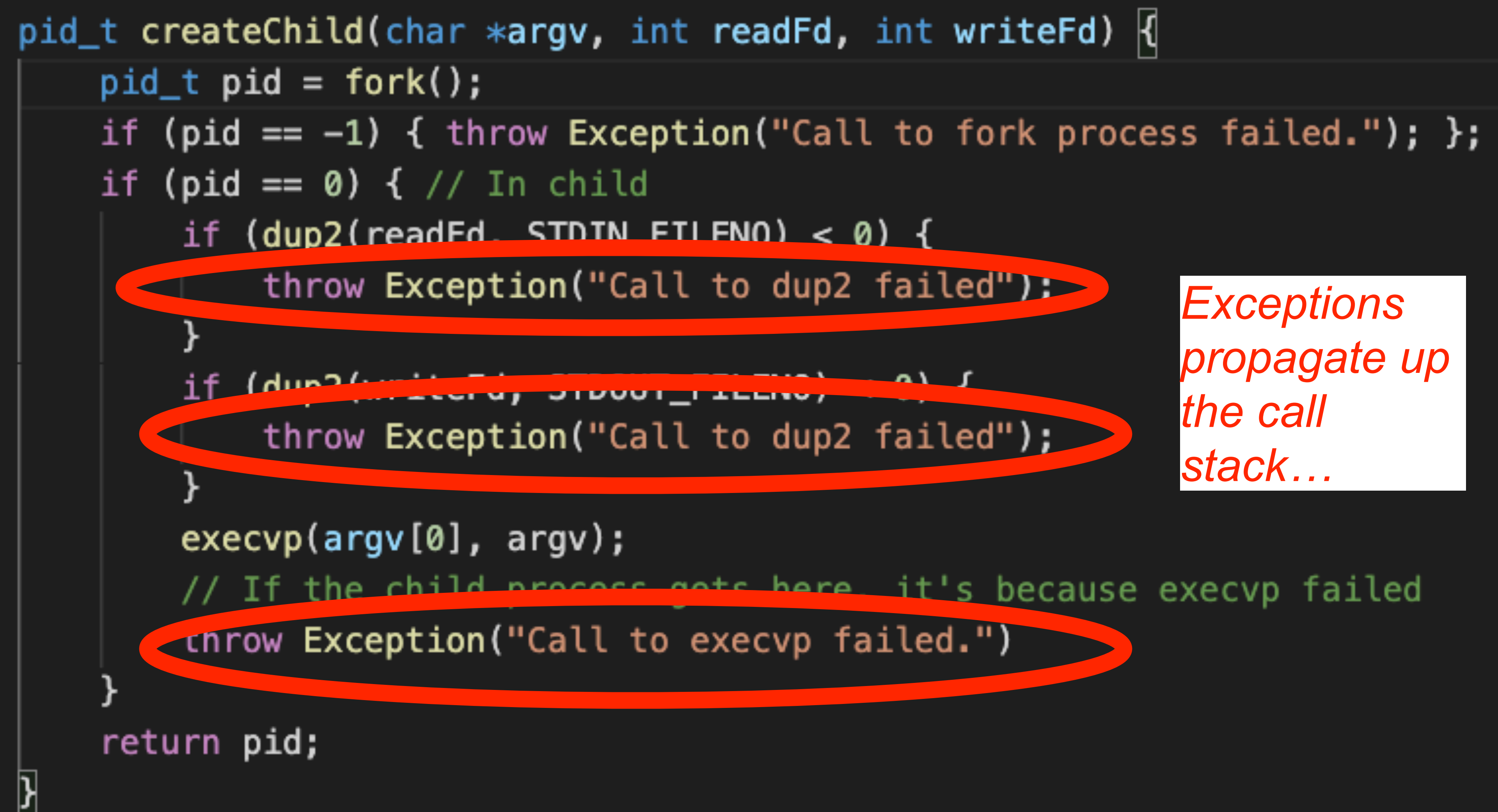


```
int createChildAndSayHello() {
    pid_t pid = fork();
    if (pid == 0) {
        sayHello();
        return 0;
    }
    waitpid(pid);
}
```

- Returning from `main` = exiting the process
- Returning from a *function* (not main) = returning to its caller
- Child process got copy of its parents call stack
- Returning here, child process isn't exiting — it's going off and executing code that was probably intended for the parent



```
pid_t createChild(char *argv, int readFd, int writeFd) {
    pid_t pid = fork();
    if (pid == -1) { throw Exception("Call to fork process failed."); };
    if (pid == 0) { // In child
        if (dup2(readFd, STDIN_FILENO) < 0) {
            throw Exception("Call to dup2 failed");
        }
        if (dup2(writeFd, STDOUT_FILENO) < 0) {
            throw Exception("Call to dup2 failed");
        }
        execvp(argv[0], argv);
        // If the child process gets here, it's because execvp failed
        throw Exception("Call to execvp failed.")
    }
    return pid;
}
```



```
pid_t createChild(char *argv, int readFd, int writeFd) {
    pid_t pid = fork();
    if (pid == -1) { throw Exception("Call to fork process failed."); };
    if (pid == 0) { // In child
        if (dup2(readFd, STDIN_FILENO) < 0) {
            throw Exception("Call to dup2 failed");
        }
        if (dup2(writeFd, STDOUT_FILENO) < 0) {
            throw Exception("Call to dup2 failed");
        }
        execvp(argv[0], argv);
        // If the child process gets here, it's because execvp failed
        throw Exception("Call to execvp failed.")
    }
    return pid;
}
```

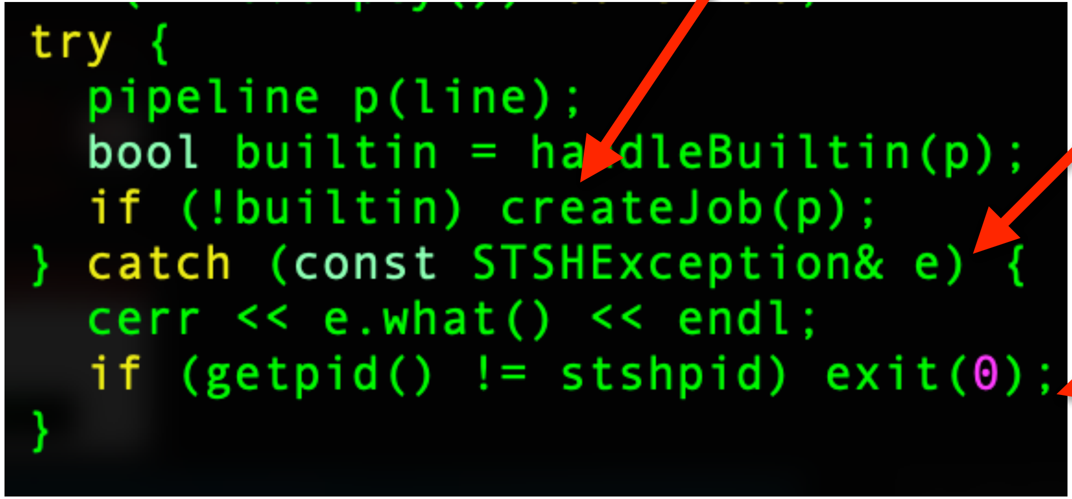
Exceptions propagate up the call stack…

Aside: from 2021 assignment 4 starter code

In parent process, in `main`

might create child process

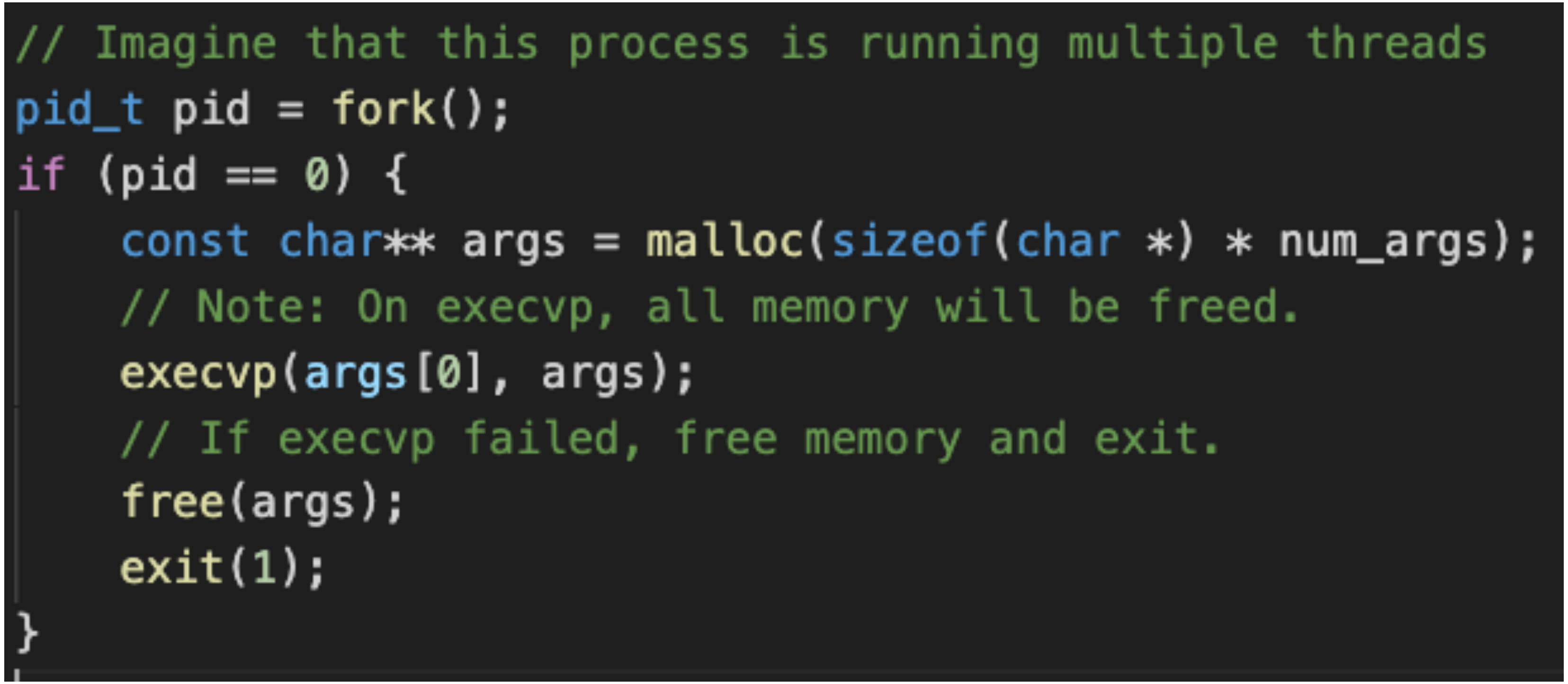
Exception may be thrown in these functions (custom exception type)



```
try {
  pipeline p(line);
  bool builtin = handleBuiltin(p);
  if (!builtin) createJob(p);
} catch (const STSHException& e) {
  cerr << e.what() << endl;
  if (getpid() != stshpid) exit(0);
}
```

- GetPid of the calling process (the one that threw the exception)
- Compare against saved PID of original parent process
- If these don't match, exit the calling process
- *Why is this line of code critical?*

Mixing threads & processes



```
// Imagine that this process is running multiple threads
pid_t pid = fork();
if (pid == 0) {
    const char** args = malloc(sizeof(char *) * num_args);
    // Note: On execvp, all memory will be freed.
    execvp(args[0], args);
    // If execvp failed, free memory and exit.
    free(args);
    exit(1);
}
```

[threads were just introduced today in CS110]

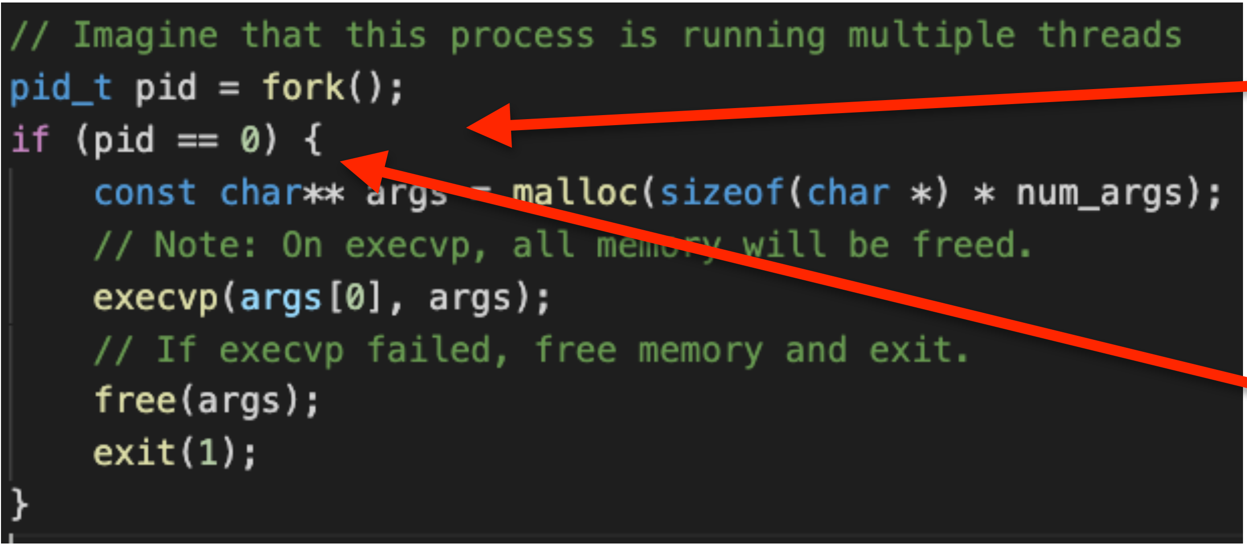
Mixing threads & processes

- Unlike processes, threads share the same virtual address space
- Malloc is thread-safe
    - Internally, uses a *lock* to make sure that two threads can't be in a \`malloc\` call concurrently — to make sure that one thread can't corrupt another thread's heap data.
    - (Think about heap allocator in CS107 — lots of internal memory management and data structures that could get corrupted if concurrent access were allowed!)

Mixing threads & processes

- If the parent process had multiple active threads when it called `fork`, it doesn’t matter — only one thread will exist in the child process (the copy of the one that just called fork).
  - After a fork, only one thread is running in the child.
  - More here.
- But, remember, when you call `fork`, the child gets a duplicate of all parent process memory
  - This includes stack, heap state, etc.
  - It also includes the state of any *locks*

Mixing threads & processes



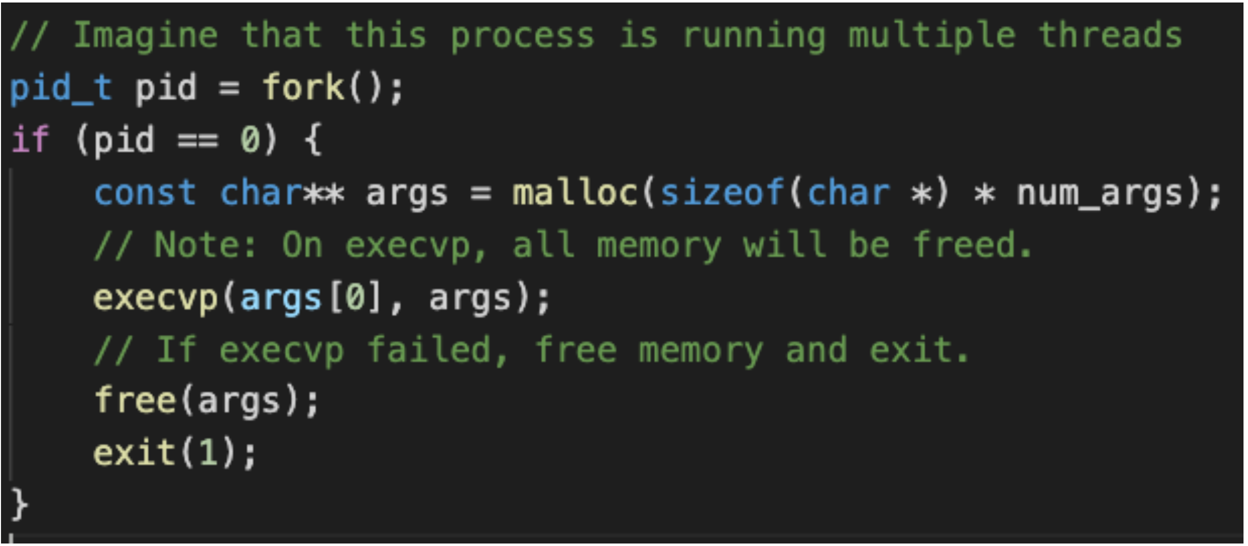
```
// Imagine that this process is running multiple threads
pid_t pid = fork();
if (pid == 0) {
    const char** args = malloc(sizeof(char *) * num_args);
    // Note: On execvp, all memory will be freed.
    execvp(args[0], args);
    // If execvp failed, free memory and exit.
    free(args);
    exit(1);
}
```

imagine this is called by thread A, but imagine that — at the moment this is called — thread B holds the “malloc lock”

the “malloc lock” is just a value in memory, so its value at the moment of `fork` is copied over into the child’s address space

…but thread B is not running in the child process, so it can’t actually release the lock

Mixing threads & processes



```
// Imagine that this process is running multiple threads
pid_t pid = fork();
if (pid == 0) {
    const char** args = malloc(sizeof(char *) * num_args);
    // Note: On execvp, all memory will be freed.
    execvp(args[0], args);
    // If execvp failed, free memory and exit.
    free(args);
    exit(1);
}
```

…so the child process gets stuck on this `malloc` call forever

Mixing threads & processes

- You might be sure that the piece of code that you’re writing doesn’t have any threads
    - But are you sure that the libraries you call don’t use any threads?
    - Are you sure that there isn’t (for example) a background thread running that was spawned in an entirely different part of the codebase?

Mixing threads & processes

- In practice, **don’t mix multiprocessing with threads** if child processes are going to do **any meaningful work in the cloned memory space** (i.e., anything other than immediately calling `exec`).
  - (And, remember that, in practice, it can be **hard to guarantee that there *aren’t* threads** somewhere in your calling process)
- This category of issues (e.g., deadlock from failing to free resources) **is probably the biggest danger with `fork()`.**
- If you want to use multiprocessing for concurrent execution, take the code you want to run concurrently and **put it in a separate executable**
  - Invoking it with `exec` will “reset” virtual memory space

What can go wrong with fork()? (recap)

- How did we (maybe) mess things up when calling fork()?
    - Accidentally nesting forks when spawning multiple child processes
    - Runaway children
    - Failing to free locks when threads are involved
    - Failure to clean up (zombie processes)

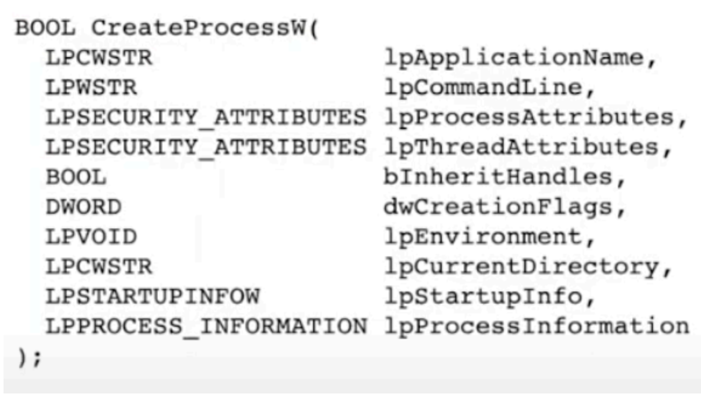
Why might you call fork? (recap/edited) 🍴

- ~~Get concurrent execution (i.e. run another piece of your own program at the same time)~~
- Invoke external functionality on the system (i.e. run a different executable) **but remember to properly exit if exec fails!**

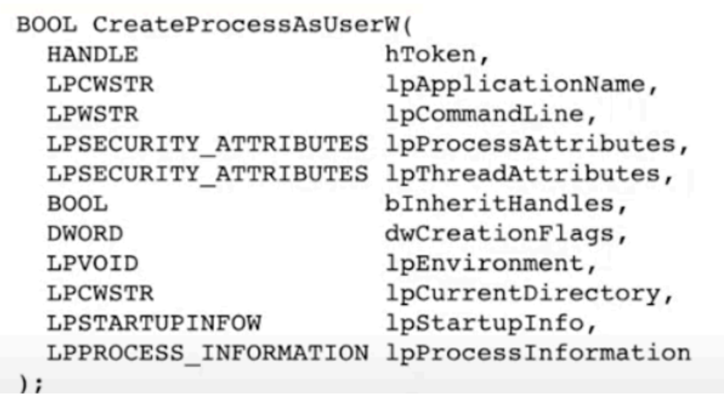
Why separate fork and exec?

- **Linux: customization and simplicity**
  - Rewire file descriptors? Change some environment variables? Block signals? Pin a process to a particular CPU core (cache optimization)?
  - Maybe too much flexibility? —> more mistakes you can make

- **Windows:** One sys call: well-defined API, but *complicated*



```
BOOL CreateProcessW(
   LPCWSTR                lpApplicationName,
   LPWSTR                 lpCommandLine,
   LPSECURITY_ATTRIBUTES  lpProcessAttributes,
   LPSECURITY_ATTRIBUTES  lpThreadAttributes,
   BOOL                   bInheritHandles,
   DWORD                  dwCreationFlags,
   LPVOID                 lpEnvironment,
   LPCWSTR                lpCurrentDirectory,
   LPSTARTUPINFOW         lpStartupInfo,
   LPPROCESS_INFORMATION  lpProcessInformation
);
```



```
BOOL CreateProcessAsUserW(
   HANDLE                 hToken,
   LPCWSTR                lpApplicationName,
   LPWSTR                 lpCommandLine,
   LPSECURITY_ATTRIBUTES  lpProcessAttributes,
   LPSECURITY_ATTRIBUTES  lpThreadAttributes,
   BOOL                   bInheritHandles,
   DWORD                  dwCreationFlags,
   LPVOID                 lpEnvironment,
   LPCWSTR                lpCurrentDirectory,
   LPSTARTUPINFOW         lpStartupInfo,
   LPPROCESS_INFORMATION  lpProcessInformation
);
```

Common multiprocessing tactic

- The flexibility of `fork` and `exec` is there if you need it.
- Define/use a **higher level abstraction** to take care of common cases
  - Ex: “subprocess” (from CS110 assign3/lab3)
  - Like the Windows approach, but no need for the OS to cover all possible valid use cases
- Most of these abstractions allow you to **redirect standard input/output** and **provide a *function*** that you want to be executed **after fork** and **before exec.**

Recap (starting here on 2/7)

- Common mistakes when calling fork():
  - Accidentally running code in child that’s meant for the parent (via runaway children, or mistakenly putting code before `if pid == 0`)
    - Common: failing to properly error handle `exec`, or throwing an exception that is caught somewhere else
  - Inheriting virtual memory state —> risk of deadlock with threads
  - Failing to clean up (can run out of space in OS process table!)

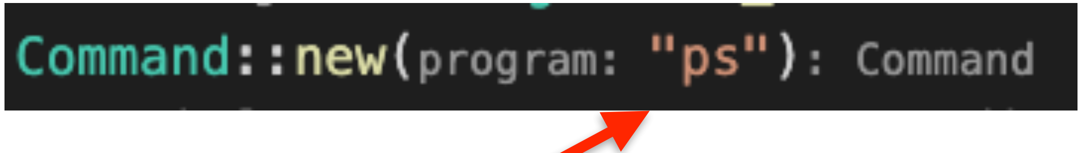
Recap (starting here on 2/7)

- We argue: don’t call **fork** unless you’re about to call **exec**
- And: define/use a **higher level abstraction** to combine `fork` and `exec` for the common cases. Generally, this will include some interface to, e.g., create pipes.
    - *This abstraction is built into higher-level languages — like Python and Rust — and can be created in C/C++ (e.g., `subprocess` class)*

`Command` in Rust

https://doc.rust-lang.org/std/process/struct.Command.html

- Step 1: set up the command.
    - What do you want to run?



```
Command::new(program: "ps"): Command
```

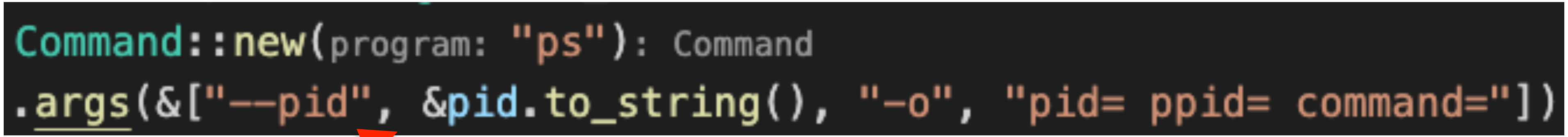
Name of executable you want the process to run
- *Here, it's `ps`, a* Linux utility *for displaying info about running processes*

From week 3 starter code
With rust-analyzer type annotations turned on

`Command` in Rust

https://doc.rust-lang.org/std/process/struct.Command.html

- Step 2: set up the command (part 2)
  - What arguments do you want to run it with?



```
Command::new(program: "ps"): Command
.args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
```

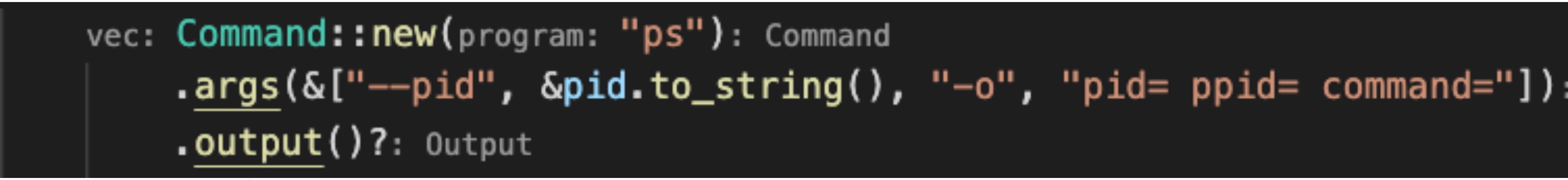
Arguments you want to run `ps` with
- --pid [pid] => show info about process with a specific `pid`
- -o and on: specify how to format the output

From week 3 starter code With rust-analyzer type annotations turned on

`Command` in Rust

https://doc.rust-lang.org/std/process/struct.Command.html

- Step 3: run the command
  - There are a few different ways to do this; here's one



```
vec: Command::new(program: "ps"): Command
    .args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
    .output()?: Output
```

.output() will:

- Run this subprocess and block (wait for it to finish)
- On success, return a Result with (if Ok) stdout, stderr, and exit status.

From week 3 starter code With rust-analyzer type annotations turned on

`output() -> Output`

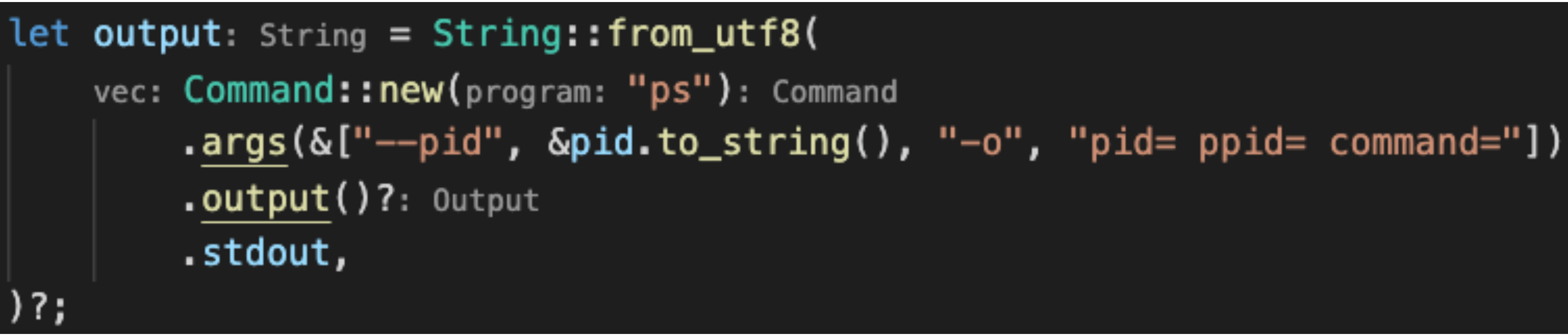
- More specifically, `Command.**output()**` will:
  - Start the subprocess
  - PAUSE the parent process
  - Return a Result:
    - Could be an error, e.g., if `exec` failed
    - If it's Ok, it'll contain an **Output** struct that contains **ExitStatus**, **stdout**, and **stderr** fields
- *Can think of this as a combination of fork, exec, waitpid, and rewiring stdin/stdout to pipes!*

https://doc.rust-lang.org/std/process/struct.Output.html

`Command` in Rust

https://doc.rust-lang.org/std/process/struct.Command.html

- Full code from week 3 starter code:
  - Sets up command to invoke `ps` with arguments
  - Calls output() to pause execution and get back Result<Output>
  - Applies `?` to the result to extract the Output (or return Error)
  - Gets the `stdout` field from the Output, and converts it to a string.

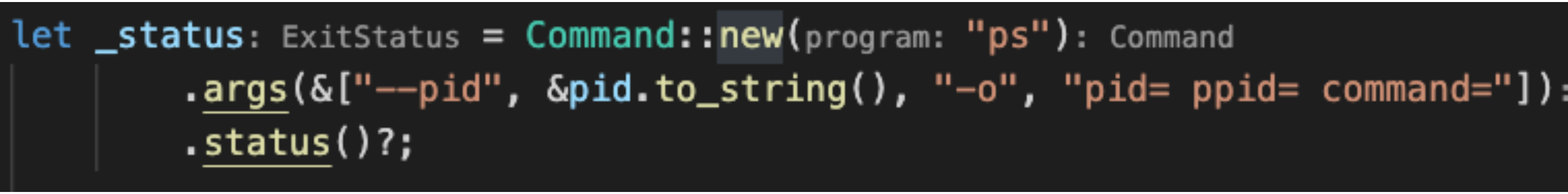


```
let output: String = String::from_utf8(
    vec: Command::new(program: "ps"): Command
        .args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
        .output()?: Output
        .stdout,
)?;
```

`Command` in Rust

https://doc.rust-lang.org/std/process/struct.Command.html

- Alternative step 3: **.status()**
  - Run child process
  - Pause parent’s execution until finished
  - Don’t get back stdout/stderr, but do get Result(exit status)

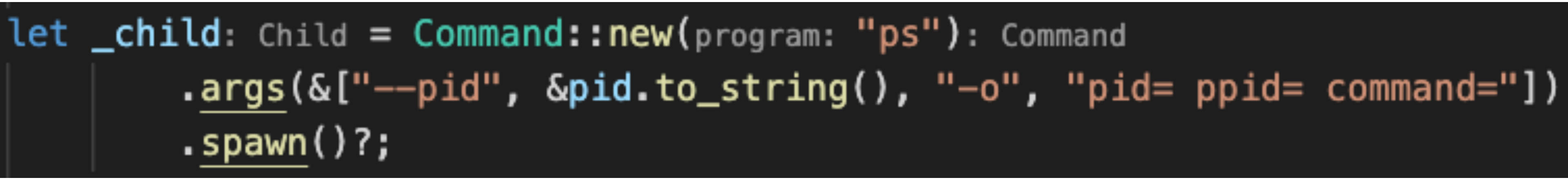


```
let _status: ExitStatus = Command::new(program: "ps"): Command
        .args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
        .status()?;
```

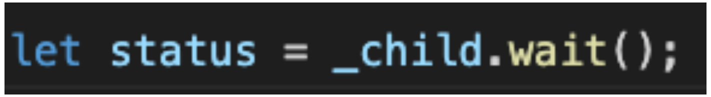
`Command` in Rust

https://doc.rust-lang.org/std/process/struct.Command.html

- Alternative step 3: **.spawn()**
  - Start up child process
  - DON'T pause parent's execution while child is running
  - Get a Result(Child struct) back



```
let _child: Child = Command::new(program: "ps"): Command
        .args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
        .spawn()?;
```



```
let status = _child.wait();
```

- *Must call "wait" on child later!*

Pre-exec function (needed for project 1)

```
use std::os::unix::process::CommandExt;
...
// Initialize Command
let cmd = Command::new("ls");
// Add pre-exec function
unsafe {
  cmd.pre_exec(function_to_run);
}
// Spawn child process
let child = cmd.spawn()
```

We haven't talked about "unsafe" Rust yet. Think about the **unsafe** block here as a **warning** — telling you to limit what you do in this function. (E.g., avoid allocating memory or accessing shared data in the presence of threads.)

It's rare that you would need to specify a pre-exec function, but you'll need it to make a system call to set up debugging in project 1

Concurrent execution

- How did we (maybe) mess things up when calling fork()?
    - ~~Accidentally nesting forks when spawning multiple child processes~~
    - ~~Runaway children~~
    - ~~Failing to free locks when threads are involved~~
    - Failure to clean up (zombie processes)
        - Still a thing!
        - You could implement a struct with a Drop trait that calls wait()?
- *You can also do all of these things in C++*

Project 1 :)

- Is a thing! Lots of multiprocessing! Lots of Rust!
- Walkthrough slides are linked from the project 1 handout!