Multiprocessing

Ryan Eberhardt and Armin Namavari
April 28, 2020

Hello week 4!

You're killing it!! 🎉🔥

Class logistics

- Week 3 exercises due Wednesday
    - Please let us know if you get stuck / feel confused! We want you to sleep!
    - Also, remember you can substitute any week’s exercises for a blog post if you’d like!
- First project (mini GDB) will be coming out late this week, due two weeks later
    - You’ll be free to work with a partner!
    - We’ll have some way for you to find someone to work with if you’d like (suggestions welcome)
- No exercise this week (just the survey)

This week

- Taking a brief break from Rust-land!
- Today: why you shouldn’t use fork(), pipe(), or signal() 🔥🚒
- Thursday: multiprocessing case study of Google Chrome

Don’t call fork()

Why 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)

Concurrent execution

- How might we mess this up? (live code)

Concurrent execution

- How might we mess this up?
  - Accidentally nesting forks when spawning multiple child processes
  - Runaway children
  - Using data structures when threads are involved
  - Failure to clean up (zombie processes)

Concurrent execution

- I argue: It’s better to take the code you want to run concurrently and put it in a separate executable
  - You won’t inherit data from the parent process’s virtual address space, but that’s the point
  - Use arguments or pipes to provide whatever information is needed for the child process to run

Why 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)

Invoking external functionality

- How do you start a subprocess?
    - fork(), then exec()
- Almost every fork() is followed by an exec()
- Why didn’t they just make a combined syscall?

Child processes in Windows

```
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
);
```

fork() and exec() rationale

- The Unix approach is *simple* and *powerful*
    - You can make *any* desired customizations to your child process before it executes the desired binary
    - Change environment variables, rewire file descriptors, block/unblock signals, take control of the terminal, enable debugging, etc.
- Simple != easy
    - malloc() and free() are simple, too!

Common multiprocessing tactic

- Let fork() and exec() be. The power is there if you need it.
- Define a higher-level abstraction to take care of the common cases
    - You’re implementing one such simple abstraction in CS 110 assign3!
    - Usually, these abstractions allow a “pre-exec function” to be specified, which is called after fork() but before exec()
    - With such an abstraction, really no reason to call fork() or exec()!

Command in Rust

- Build a Command:

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

- Run, and get the output in a buffer:

```
let output = Command::new("ps")
    .args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
    .output()
    .expect("Failed to execute subprocess”)
```

  - Includes exit status, stdout, and stderr

Command in Rust

- Run (without swallowing output), and get the status code:

  ```
  let status = Command::new("ps")
      .args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
      .status()
      .expect("Failed to execute subprocess")
  ```
- Spawn and immediately return:

  ```
  let child = Command::new("ps")
      .args(&["--pid", &pid.to_string(), "-o", "pid= ppid= command="])
      .spawn()
      .expect("Failed to execute subprocess")
  ```
  - This returns a `Child`, which you need to wait on at some point!

    ```
    let status = child.wait()
    ```

Command in Rust

- Pre-exec function:

```
use std::os::unix::process::CommandExt;
...
let cmd = Command::new("ls");
unsafe {
    cmd.pre_exec(function_to_run);
}
let child = cmd.spawn();
```

  - The `unsafe` block acts as a warning to avoid allocating memory or accessing shared data in the presence of threads
  - It's quite rare that you would need to specify a pre_exec function (the `Command` API takes care of most things), but you'll need it for Project 1

Concurrent execution

- How might we mess this up?
    - ~~Accidentally nesting forks when spawning multiple child processes~~
    - ~~Runaway children~~
    - ~~Using data structures when threads are involved~~
    - Failure to clean up (zombie processes)
        - You could implement a struct with a Drop trait that calls wait()

Don’t call pipe()

Problems with pipes

What can you think of?

Problems with pipes

- Leaked file descriptors
- Calling close() on bad values

  Example:

```
if (close(fds[1] == -1)) {
    printf("Error closing!");
}
```

- Use-before-pipe (i.e. use of uninitialized ints)
- Use-after-close

Potential solution

- Add a layer of abstraction!
- Writing to a stdin pipe:

```
let mut child = Command::new("cat")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()?;
child.stdin.as_mut().unwrap().write_all(b"Hello, world!\n")?;
let output = child.wait_with_output()?;
```

- The os_pipe crate allows for creating arbitrary pipes. (The Drop trait closes the pipe.)

Aside

Don’t call signal()

Is it safe?

- Discuss in groups
    - Introduce yourself!
    - See Lecture Notes on course website

(Continued next time)