Multiprocessing (part 2)

Ryan Eberhardt and Julio Ballista
May 4, 2021

Project logistics

- Project (mini gdb) came out last Thursday, due next Thursday
- You're also welcome to propose your own project! Run your idea by us before you start working on it
  - Rust tooling (e.g. annotate code showing where values get dropped)
  - Write a raytracer
  - Pick a command-line tool and try to beat its performance (e.g. grep)
  - Implement a simple database

Upcoming roadmap

- Today: Why you shouldn't use fork(), pipe(), or signal() 🔥🚒
- Thursday: Multiprocessing case study of Firefox and Google Chrome
- Next week: Multithreading!!
- No exercises (just weekly survey) coming out this week

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 executing code they weren’t supposed to execute
  - Using data structures when threads are involved
  - Failure to clean up (zombie processes)

Concurrent execution

- Key point: It may be extremely hard to tell if you're doing something dangerous
- This code is completely broken:

```
int main() {
    some_external_helper_function();
    sleep(1);

    std::cout << "Forking child process" << std::endl;
    if (fork() == 0) {
        std::cout << "About to concat" << std::endl;
        std::cout << string("hellooooo ") + string("world!!") << std::endl;
        exit(0);
    }
    waitpid(-1, NULL, 0);
}
```

https://cplayground.com/?p=opossum-goosander-alligator

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

- Define a pipe type instead of using numbers!
- 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.)

Why are we bothering with all of this?

- In CS 110, we're spending so much time learning how to call fork and pipe and signal, and now you're telling us not to do all those things… What's up with that?
- Systems code is extremely detail oriented, and you need to have an understanding of how everything works
  - You can swap two lines and get totally different behavior
- We want you to understand how these low-level primitives work…
  - So that you can debug problems no one else can and design new types of systems
  - So that you can respect them enough to *not* use them if you don't need to

Don’t call signal()

signal() is dead. Long live sigaction()

signal(2) - Linux man page

Name

signal - ANSI C signal handling

Synopsis

#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int *signum*, **sighandler_t** *handler*);

Description

The behavior of **signal**() varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use**: use **sigaction**(2) instead. See *Portability* below.

signal() is dead. Long live sigaction()

> **Portability**
>
> The only portable use of **signal**() is to set a signal's disposition to **SIG_DFL** or **SIG_IGN**. The semantics when using **signal**() to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose.**
> POSIX.1 solved the portability mess by specifying **sigaction**(2), which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of **signal**().

Check out the man page if you have time!

Exit on ctrl+c

```c
void handler(int sig) {
    exit(0);
}

int main() {
    signal(SIGINT, handler);
    while (true) {
        sleep(1);
    }
    return 0;
}
```

Looks good! ✅

Count number of SIGCHLDs received

```c
static volatile int sigchld_count = 0;

void handler(int sig) {
    sigchld_count += 1;
}

int main() {
    signal(SIGCHLD, handler);
    const int num_processes = 10;
    for (int i = 0; i < num_processes; i++) {
        if (fork() == 0) {
            sleep(1);
            exit(0);
        }
    }
    while (waitpid(-1, NULL, 0) != -1) {}
    printf("All %d processes exited, got %d SIGCHLDs.\n",
        num_processes, sigchld_count);
    return 0;
}
```

Okay **if we were to use sigaction** ⚠️

Count number of running processes

```c
static volatile int running_processes = 0;

void handler(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0) {
        running_processes -= 1;
    }
}

int main() {
    signal(SIGCHLD, handler);
    const int num_processes = 10;
    for (int i = 0; i < num_processes; i++) {
        if (fork() == 0) {
            sleep(1);
            exit(0);
        }
        running_processes += 1;
        printf("%d running processes\n", running_processes);
    }
    while(running_processes > 0) {
        pause();
    }
    printf("All processes exited! %d running processes\n", running_processes);
    return 0;
}
```

Not safe (concurrent use of running_processes) 🚫

Print on ctrl+c

```c
void handler(int sig) {
    printf("Hehe, not exiting!\n");
}

int main() {
    signal(SIGINT, handler);
    while (true) {
        printf("Looping...\n");
        sleep(1);
    }
    return 0;
}
```

Not safe!! 🚫

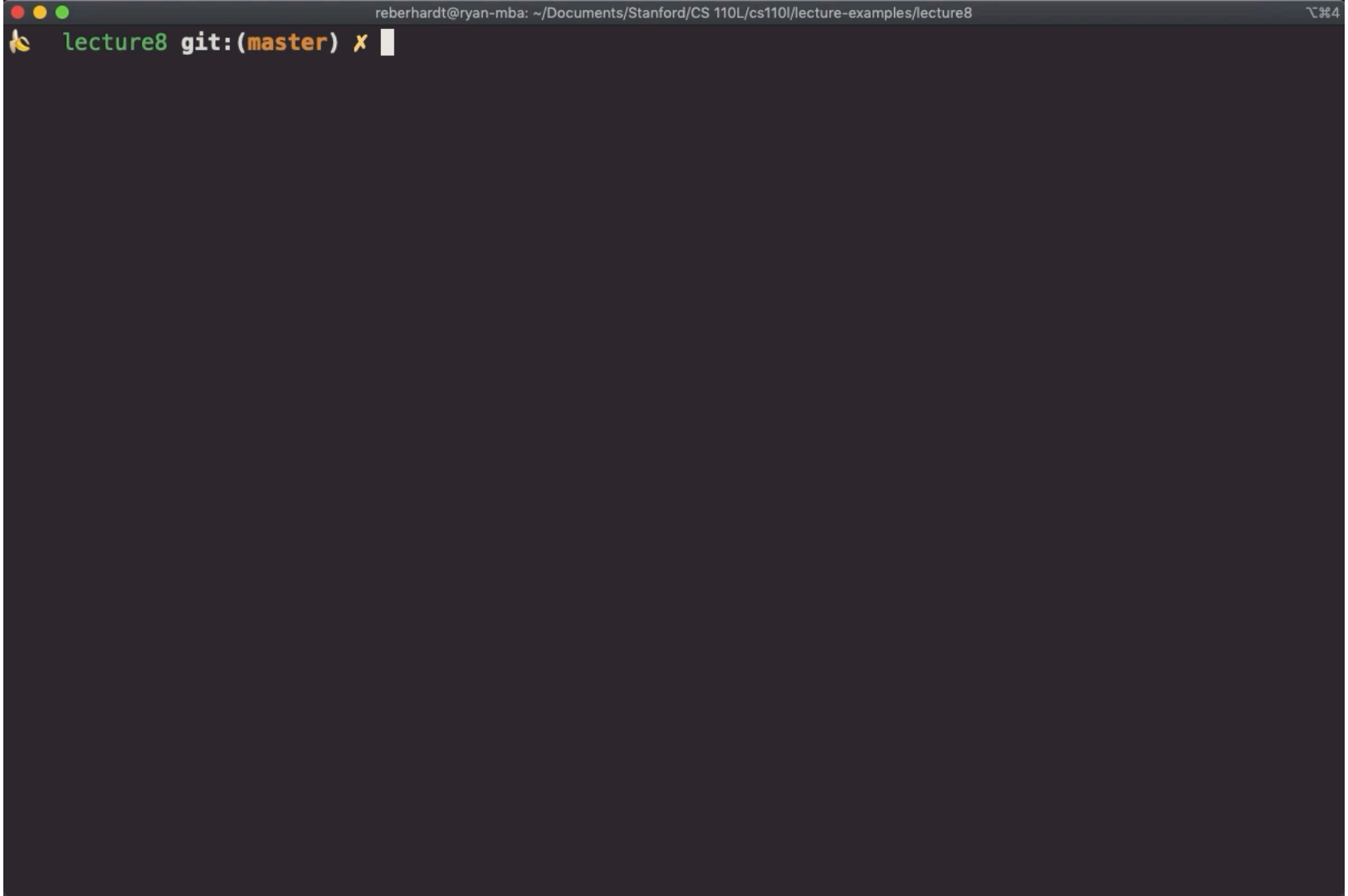
Print on ctrl+c

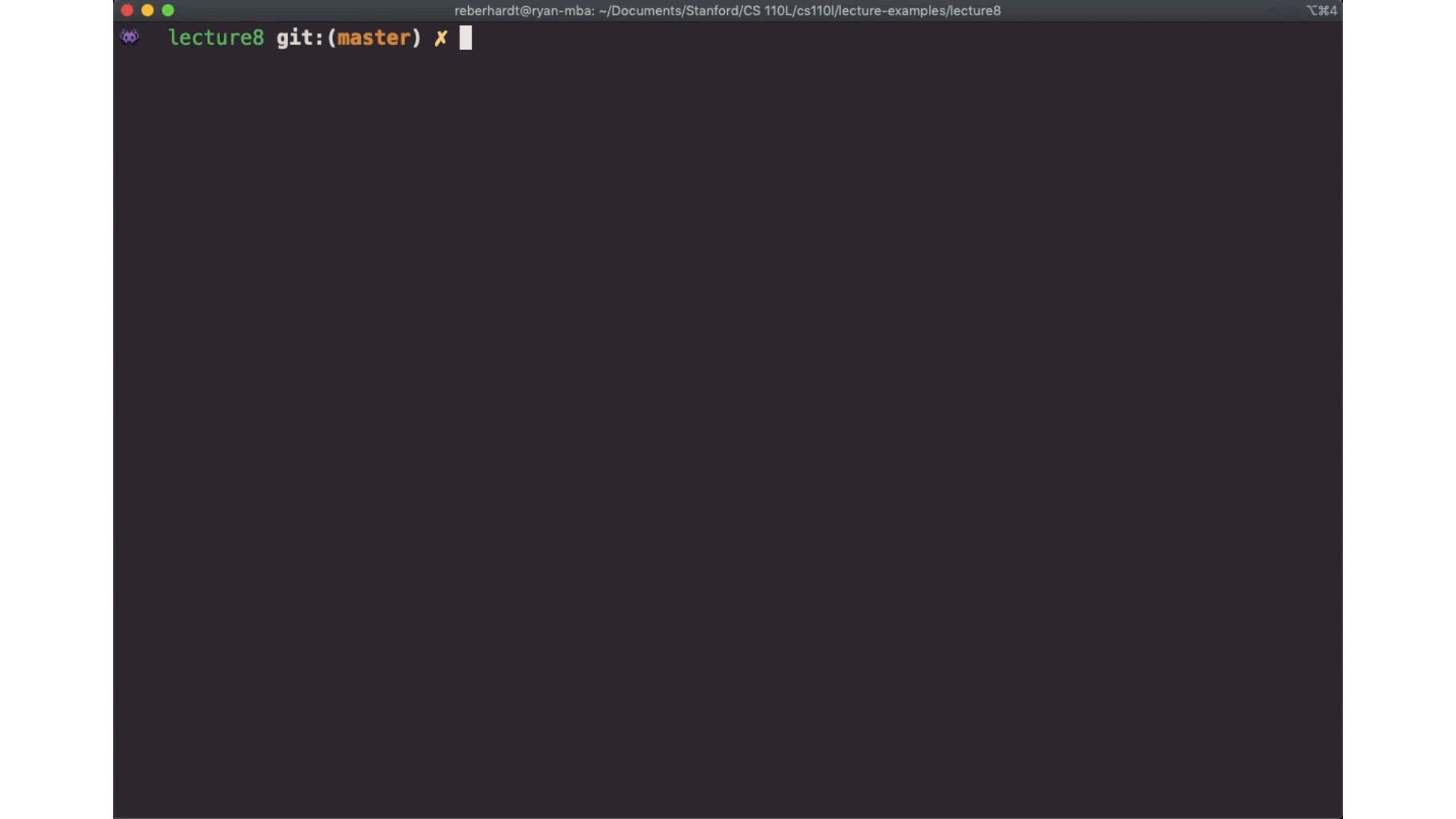
```c
void handler(int sig) {
    printf("Hehe, not exiting!\n");
}

int main() {
    signal(SIGINT, handler);
    while (true) {
        printf("Looping...\n");
        sleep(1);
    }
    return 0;
}
```

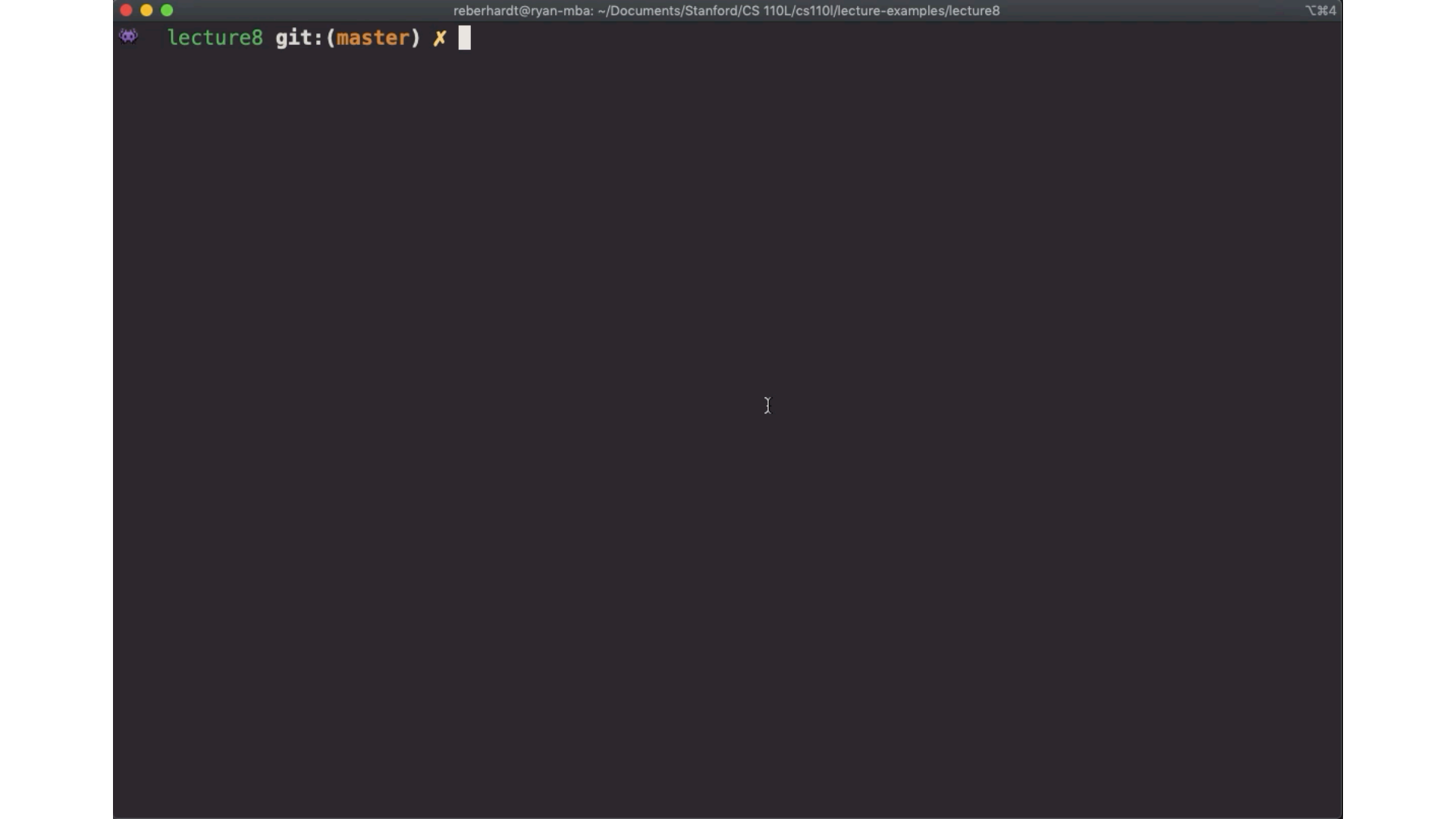
Not safe!! 🚫

```
void print_hello(int sig) {
    printf("Hello world!\n");
}
```

```
int main() {
    const char* message = "Hello world ";
    const size_t repeat = 1000;

    char *repeated_msg = malloc(repeat * strlen(message) + 2);
    for (int i = 0; i < repeat; i++) {
        strcpy(repeated_msg + (i * strlen(message)), message);
    }
    repeated_msg[repeat * strlen(message)] = '\n';
    repeated_msg[repeat * strlen(message) + 1] = '\0';

    signal(SIGUSR1, print_hello);
    if (fork() == 0) {
        pid_t parent_pid = getppid();
        while (true) {
            kill(parent_pid, SIGUSR1);
        }
        return 0;
    }

    while (true) {
        printf(repeated_msg);
    }

    free(repeated_msg);
    return 0;
}
```


reberhardt@ryan-mba: ~/Documents/Stanford/CS 110L/cs110l/lecture-examples/lecture8
⌥⌘4
lecture8 git:(master) ✗


reberhardt@ryan-mba: ~/Documents/Stanford/CS 110L/cs110l/lecture-examples/lecture8
⌥⌘4
lecture8 git:(master) ✗


reberhardt@ryan-mba: ~/Documents/Stanford/CS 110L/cs110l/lecture-examples/lecture8
⌥⌘4
lecture8 git:(master) ✗

Async-safe functions

- vfprintf is a 1787-line function!
- Deadlock:

```
1309   /* Lock stream.  */
1310   _IO_cleanup_region_start ((void (*) (void *)) &_IO_funlockfile, s);
1311   _IO_flockfile (s);
```

- I'm guessing other bizarre behavior comes from allocating memory
- You should avoid functions that use global state
  - Many functions do this, even if you may not realize it
  - malloc and free are not async-signal-safe!
- List of safe functions: http://man7.org/linux/man-pages/man7/signal-safety.7.html

What should we do?

Avoiding signal handling

- Anything substantial should not be done in a signal handler
- How can we handle signals, then?
- The “self-pipe” trick was invented in the early 90s:
  - Create a pipe
  - When you’re awaiting a signal, read from the pipe (this will block until something is written to it)
  - In the signal handler, write a single byte to the pipe

Avoiding signal handling

- signalfd added official support for this hack

```c
int main(int argc, char *argv[]) {
    sigset_t mask;
    int sfd;
    struct signalfd_siginfo fdsi;
    ssize_t s;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);

    /* Block signals so that they aren't handled
       according to their default dispositions */

    if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
        handle_error("sigprocmask");

    sfd = signalfd(-1, &mask, 0);
    if (sfd == -1) handle_error("signalfd");
```

```c
    for (;;) {
        s = read(sfd, &fdsi,
            sizeof(struct signalfd_siginfo));
        if (s != sizeof(struct signalfd_siginfo))
            handle_error("read");

        if (fdsi.ssi_signo == SIGINT) {
            printf("Got SIGINT\n");
        } else if (fdsi.ssi_signo == SIGQUIT) {
            printf("Got SIGQUIT\n");
            exit(EXIT_SUCCESS);
        } else {
            printf("Read unexpected signal\n");
        }
    }
}
```

What about asynchronous signal handling?

- I thought part of the benefit of signal handlers was you can handle events asynchronously! (You can be doing work in your program, and quickly take a break to do something to handle a signal)
- Reading from a pipe or signalfd precludes concurrency: I'm either doing work, or reading to wait for a signal, but not both at the same time
- How can we address this?
  - Use threads
    - Can still have concurrency problems!
    - But we have more tools to reason about and control those problems
  - Use non-blocking I/O (week 8)

Ctrlc crate

- Rust has a ctrlc crate: register a function to be executed on ctrl+c (SIGINT)
- How does it work?
  - Creates a self-pipe
  - Installs a signal handler that writes to the pipe when SIGINT is received
  - Spawns a thread: loop { read from pipe; call handler function; }
- The Rust borrow checker prevents data races caused by concurrent access/modification from threads. If your handler function touches data in a racey way, the compiler will complain

Why is this different?

- printf from signal handler can deadlock:
  - printf from main body of code calls flock()
  - signal handler interrupts execution. printf from signal handler calls flock()
  - signal handler can't continue until main code releases lock, but main code can't continue until the signal handler exits
- printf from threads are safe:
  - printf from main thread calls flock()
  - printf from signal handling thread calls flock() and is blocked
  - printf from main thread finishes
  - printf from signal handling thread finishes
- malloc() calls (including the ones printf makes) work similarly.

Why is this different?

- Threads and signal handlers have the same concurrency problems
- But the scheduling of code is completely different
- Threads:
  - Multiple (usually) equal-priority threads of execution that constantly swap on the processor
  - Can use locks to protect data
- Signal handlers:
  - Handler will completely preempt all other code and hog the CPU until it finishes
  - Can't use locks or any other synchronization primitives
    - In fact, signal handlers should avoid all kinds of blocking! (Why?)
  - Consequently, signal handlers play very poorly with library code. Libraries don't know what signal handlers you have installed or what those signal handlers do, so they can't disable signal handling to protect themselves from concurrency problems