Digging a rabbit hole with a fork()

CocoaHeads Paris, January 2024

Frank Lefebvre

About me

- Staff Engineer @ 

- Freelance macOS, iOS & back-end developer

- @franklefebvre@mastodon.social

Context

- macOS command-line tool
- Control long-lived processes
- Single executable file

Option #1: Launch Daemon

- Apple-sanctioned way to support a long-lived process
- /Library/LaunchDaemons/config_file.plist
- Run at startup or on demand (launchctl)
- Multiple files to install
- One plist file == one process

There must be an easier way...

Option #2: UNIX fork()

- `pid_t fork(void);`
- Usually followed by `exec*()` in child branch
- `posix_spawn()`, `Process.run()`
- Parent-child relationship

Using fork() without exec()

- Parent can exit
- Child's ppid == 1
- Daemon

POC

- daemonize.c

```
#include <stdlib.h>
#include <unistd.h>
#include "daemonize.h"

void detachCurrentProcess(void) {
    if (fork()) { // not 0, this is the parent process
        exit(0);
    }
}
```

- Bridging-Header.h

```
#import "daemonize.h"
```

POC

CLI tool (1/3)

- swift-argument-parser package

```
import Foundation
import ArgumentParser

@main
struct UDPRecorder: ParsableCommand {
    static var configuration = CommandConfiguration(
        commandName: "udp-recorder",
        subcommands: [Create.self, StartDaemon.self]
    )

    struct Create: ParsableCommand { /* TBD */ }

    struct StartDaemon: ParsableCommand { /* TBD */ }
}
```

POC

CLI tool (2/3)

```
struct Create: ParsableCommand {
    static var configuration = CommandConfiguration(
        commandName: "create"
        )

    func run() {
        guard let path = CommandLine.arguments.first else { fatalError() }
        let url = URL(filePath: path)
        let id = UUID().uuidString
        try? Process.run(url, arguments: ["start-daemon", id]).waitUntilExit()
        print("[CREATE] done \(id)")
    }
}
```

POC

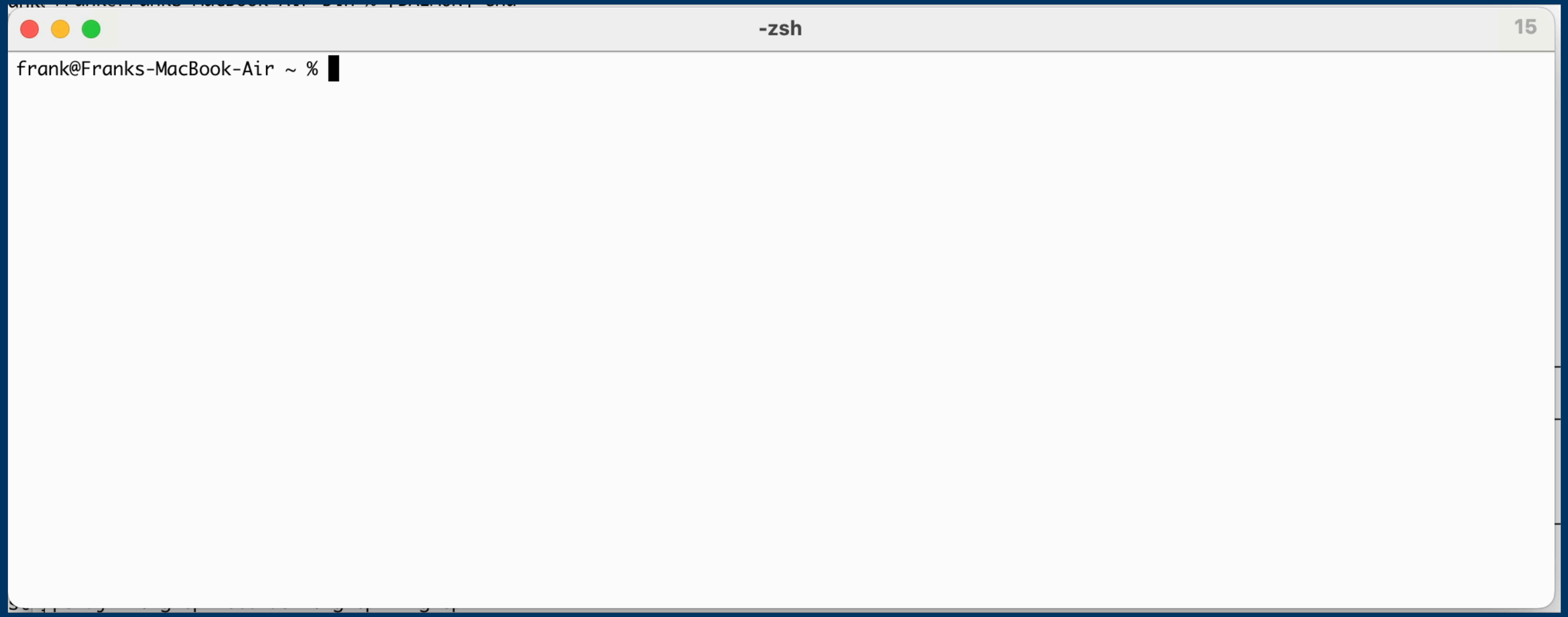
CLI tool (3/3)

```
struct StartDaemon: ParsableCommand {
    static var configuration = CommandConfiguration(
        commandName: "start-daemon",
        shouldDisplay: false
        )
    @Argument var id: String

    func run() {
        print("[DAEMON] enter - pid = \(ProcessInfo.processInfo.processIdentifier) - id = \(id)")
        detachCurrentProcess()
        print("[DAEMON] child - pid = \(ProcessInfo.processInfo.processIdentifier) - id = \(id)")
        Thread.sleep(forTimeInterval: 20)
        print("[DAEMON] end")
    }
}
```

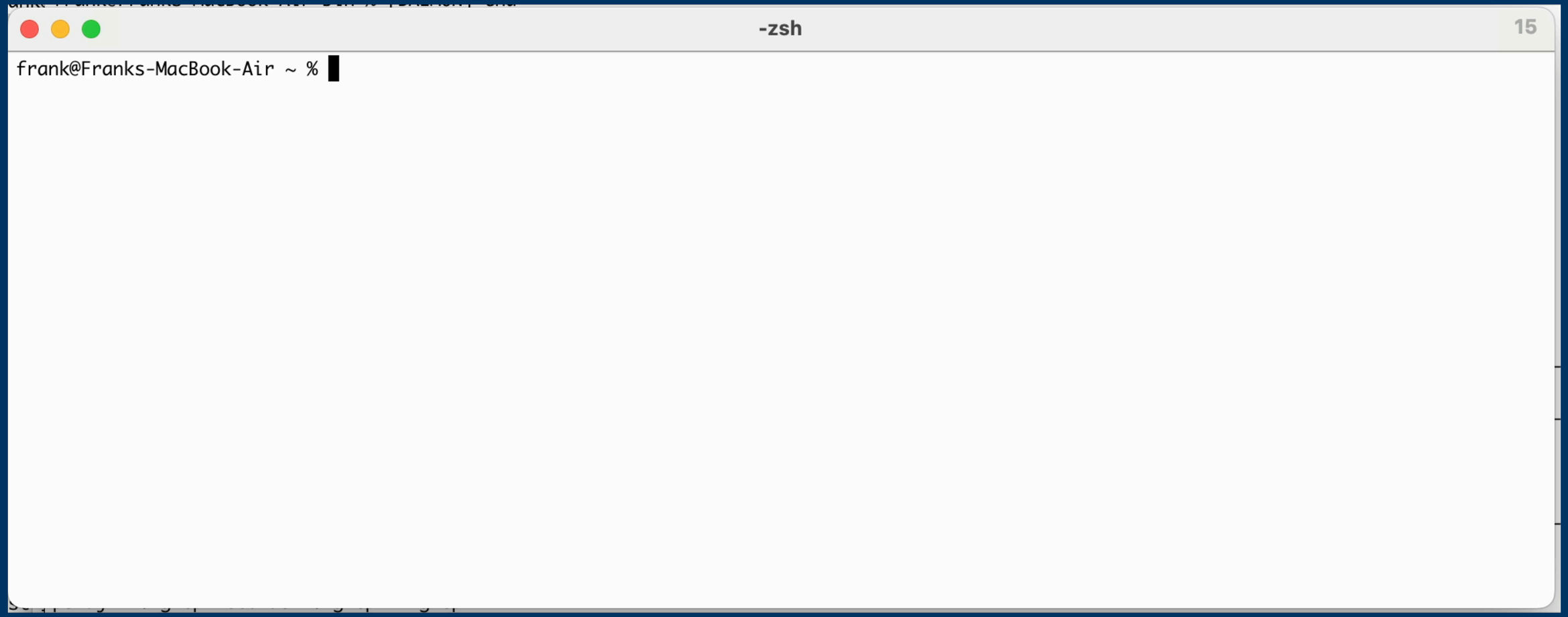
POC

Testing…



POC

Testing…



It works! 

Integration

Communication layer: UNIX local socket

```
import Foundation
import KituraNet

final class UnixSocketListener {
    private var server: HTTPServer? = nil

    init(id: String) {
        do {
            try FileManager.default.createDirectory(at: UnixSocket.url,
withIntermediateDirectories: true)
            self.server = try HTTPServer.listen(unixDomainSocketPath: "\(UnixSocket.path)/\(id)",
delegate: self)
        } catch {
            print("UnixSocketListener error \(error.localizedDescription)")
        }
    }
}

extension UnixSocketListener: ServerDelegate { /* TBD */ }
```

Integration

Using ArgumentParser with async/await

```
@main
struct UDPRecorder: ParsableCommand {
    static var configuration = CommandConfiguration(
        commandName: "udp-recorder",
        subcommands: [Create.self, StartDaemon.self]
    )

    struct Create: ParsableCommand {
        static var configuration = CommandConfiguration(
            commandName: "create")

        func run() {
            /* implementation */
        }
    }

    /* struct StartDaemon */
}
```

Integration

Using ArgumentParser with async/await

```
@main
struct UDPRecorder: ParsableCommand {
    static var configuration = CommandConfiguration(
        commandName: "udp-recorder",
        subcommands: [Create.self, StartDaemon.self]
    )

    struct Create: ParsableCommand {
        static var configuration = CommandConfiguration(
            commandName: "create")

        func run() async {
            /* implementation */
        }
    }

    /* struct StartDaemon */
}
```

Integration

Using ArgumentParser with async/await

```
@main
struct UDPRecorder: AsyncParsableCommand {
    static var configuration = CommandConfiguration(
        commandName: "udp-recorder",
        subcommands: [Create.self, StartDaemon.self]
    )

    struct Create: AsyncParsableCommand {
        static var configuration = CommandConfiguration(
            commandName: "create")

        func run() async {
            /* implementation */
        }
    }

    /* struct StartDaemon */
}
```

Testing...

Testing...

```
/tmp/udp-recorder create
```

Testing...

```
/tmp/udp-recorder create

+[Swift.__SharedStringStorage initialize] may have been in progress in another thread
when fork() was called. We cannot safely call it or ignore it in the fork() child
process. Crashing instead. Set a breakpoint on objc_initializeAfterForkError to debug.
```

Testing...

```
/tmp/udp-recorder create

+[Swift.__SharedStringStorage initialize] may have been in progress in another thread
when fork() was called. We cannot safely call it or ignore it in the fork() child
process. Crashing instead. Set a breakpoint on objc_initializeAfterForkError to debug.

OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES /tmp/udp-recorder create
```

Testing...

```
/tmp/udp-recorder create

+[Swift.__SharedStringStorage initialize] may have been in progress in another thread
when fork() was called. We cannot safely call it or ignore it in the fork() child
process. Crashing instead. Set a breakpoint on objc_initializeAfterForkError to debug.

OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES /tmp/udp-recorder create
```

Can’t communicate with the child process…

Testing...

```
/tmp/udp-recorder create

+[Swift.__SharedStringStorage initialize] may have been in progress in another thread
when fork() was called. We cannot safely call it or ignore it in the fork() child
process. Crashing instead. Set a breakpoint on objc_initializeAfterForkError to debug.

OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES /tmp/udp-recorder create
```

Can’t communicate with the child process…

… because the child has crashed

Testing...

```
/tmp/udp-recorder create

+[Swift.__SharedStringStorage initialize] may have been in progress in another thread
when fork() was called. We cannot safely call it or ignore it in the fork() child
process. Crashing instead. Set a breakpoint on objc_initializeAfterForkError to debug.

OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES /tmp/udp-recorder create
```

Can’t communicate with the child process…

… because the child has crashed

… in libDispatch when starting the server

Testing...

```
/tmp/udp-recorder create

+[Swift.__SharedStringStorage initialize] may have been in progress in another thread
when fork() was called. We cannot safely call it or ignore it in the fork() child
process. Crashing instead. Set a breakpoint on objc_initializeAfterForkError to debug.

OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES /tmp/udp-recorder create
```

Can’t communicate with the child process…

… because the child has crashed

… in libDispatch when starting the server

```
The process has forked and you cannot use this CoreFoundation functionality safely.
You MUST exec(). Break on
__THE_PROCESS_HAS_FORKED_AND_YOU_CANNOT_USE_THIS_COREFOUNDATION_FUNCTIONALITY___YOU_MU
ST_EXEC__() to debug.
```

Why?

https://forums.developer.apple.com/forums/thread/737464

Combining `fork` with Apple's frameworks is a tricky business [1]. If you only use Posix APIs, `fork` should behave reliably. Once you start using our higher-level frameworks, well... we try to keep things working but we can't make any guarantees.

There are two cases here:

- Calling `fork` without `exec*` to create a long-running 'clone' of the current process
- Combining `fork` and `exec*` to run a new program in a child process

The first case will (usually :-) work if you limit yourself to Posix APIs. It's fundamentally incompatible with our higher-level frameworks.

For the second case, the best way to avoid problems is to use an API that combines `fork` and `exec*`. At the BSD level, that's `posix_spawn`. Higher up, you have `NSTask` (aka `Process` in Swift).

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

[1] There's a fundamental disconnect between BSD and Mach on this topic, and Apple's frameworks rely on Mach a *lot*.

Posted 2 months ago by eskimo

Why?

https://forums.developer.apple.com/forums/thread/737464

Combining `fork` with Apple's frameworks is a tricky business [1]. If you only use Posix APIs, `fork` should behave reliably. Once you start using our higher-level frameworks, well... we try to keep things working but we can't make any guarantees.

There are two cases here:

- Calling `fork` without `exec*` to create a long-running 'clone' of the current process
- Combining `fork` and `exec*` to run a new program in a child process

The first case will (usually :-) work if you limit yourself to Posix APIs. It's fundamentally incompatible with our higher-level frameworks.

For the second case, the best way to avoid problems is to use an API that combines `fork` and `exec*`. At the BSD level, that's `posix_spawn`. Higher up, you have `NSTask` (aka `Process` in Swift).

Share and Enjoy

—

Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

[1] There's a fundamental disconnect between BSD and Mach on this topic, and Apple's frameworks rely on Mach a *lot*.

Posted 2 months ago by eskimo

Why?

https://forums.developer.apple.com/forums/thread/737464

Combining `fork` with Apple's frameworks is a tricky business [1]. If you only use Posix APIs, `fork` should behave reliably. Once you start using our higher-level frameworks, well... we try to keep things working but we can't make any guarantees.

There are two cases here:

- Calling `fork` without `exec*` to create a long-running 'clone' of the current process
- Combining `fork` and `exec*` to run a new program in a child process

The first case will (usually :-) work if you limit yourself to Posix APIs. It's fundamentally incompatible with our higher-level frameworks.

For the second case, the best way to avoid problems is to use an API that combines `fork` and `exec*`. At the BSD level, that's `posix_spawn`. Higher up, you have `NSTask` (aka `Process` in Swift).

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

[1] There's a fundamental disconnect between BSD and Mach on this topic, and Apple's frameworks rely on Mach a *lot*.

Posted 2 months ago by eskimo

Why?

https://forums.developer.apple.com/forums/thread/737464

Combining `fork` with Apple's frameworks is a tricky business [1]. If you only use Posix APIs, `fork` should behave reliably. Once you start using our higher-level frameworks, well... we try to keep things working but we can't make any guarantees.

There are two cases here:

- Calling `fork` without `exec*` to create a long-running 'clone' of the current process
- Combining `fork` and `exec*` to run a new program in a child process

The first case will (usually :-) work if you limit yourself to Posix APIs. It's fundamentally incompatible with our higher-level frameworks.

For the second case, the best way to avoid problems is to use an API that combines `fork` and `exec*`. At the BSD level, that's `posix_spawn`. Higher up, you have `NSTask` (aka `Process` in Swift).

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

[1] There's a fundamental disconnect between BSD and Mach on this topic, and Apple's frameworks rely on Mach a *lot*.

Posted 2 months ago by eskimo

Option #3

Trampoline CLI tool

- POSIX-only
- daemonize + `posix_spawn` multithreaded CLI tool

Option #3

Trampoline CLI tool

- POSIX-only
- daemonize + `posix_spawn` multithreaded CLI tool
- This tool exists: `/usr/bin/nohup`

Final Code

```
struct Create: AsyncParsableCommand {
    static var configuration = CommandConfiguration(commandName: "create")

    func run() async {
        let id = UUID().uuidString
        guard let path = CommandLine.arguments.first else { fatalError() }
        let command = URL(filePath: "/usr/bin/nohup")
        let arguments = [path, “start-recorder”, id]
        do {
            let process = Process()
            process.executableURL = command
            process.arguments = arguments
            process.standardOutput = nil
            process.standardError = nil
            try process.run()
        } catch {
            print("Process failed: \(error)")
        }
        // ... send request to local socket and wait for response
    }
}
```

Questions?