porch(1)

It's not what you expect(1)

By: Kyle Evans

Note on Pronunciation

- Originally (orch)estrator - because it orchestrated a program via a tty
- Rust lib of the same name announced in the interim
- Renamed to porch to disambiguate - (p)rogram (orch)estrator

whoami

- FreeBSD Engineering Manager @ Klara, Inc
- FreeBSD src committer since 2017, some relevant work:
  - lualoader (learned lua here - 2018)
  - Pushed for flua [2019]
  - Rewrote makesyscalls.sh (sed|awk beauty) [2019, fixed in 2024]
  - Rewrote makeman.sh [2025]
  - Stuff
- Not good at pronouncing things
- Not good at producing diagrams

1. **Motivation**
    a. Relevant TTY concepts
    b. What I wanted
    c. What is expect(1)?
    d. Testing challenges
2. **Design**
    a. Overview
    b. Scripted mode
    c. "Direct" execution mode
    d. Scripted vs. Direct
3. **Other Features**
    a. porchgen(1)
    b. rporch(1)
    c. porchfuzz(1)?
4. **Examples**

Motivation

Relevant TTY concepts

- pts(4), pty(3)
  - Commonly available functionality amongst POSIX-y systems, pseudo-terminal driver
  - Software-driven control-side to mimic a user
- Canonicalization
  - Input pre-processed by the TTY layer
  - Line splitting applied at **read(2)** time
  - CEOF (^D by default) terminates a line or signals EOF with no line
    - PR 276220: Premature EOF when **read(2)** one byte short of VEOF marker
- FIONREAD ioctl
  - Peek how many bytes to read
  - With canonicalization applied, # bytes in next line only ("immediately available")
    - Multiple lines may be present at currently canonicalized marker

What I wanted

- Tests for the tty layer
  - Botched EOF handling with perfect buffer size (PR 276220)
  - Canonicalization corner cases (kind of a philosophical problem – canonicalize at read time, or write time?)
  - FIONREAD consistency
- Basic program orchestration
- Potential for other interactive program testing

What is expect(1)?

- "Programmed dialog with interactive programs"
- Tcl
- Why not use it?
  - Complex (needs to handle more use cases)
  - Not a good fit for what I needed
  - Public Domain

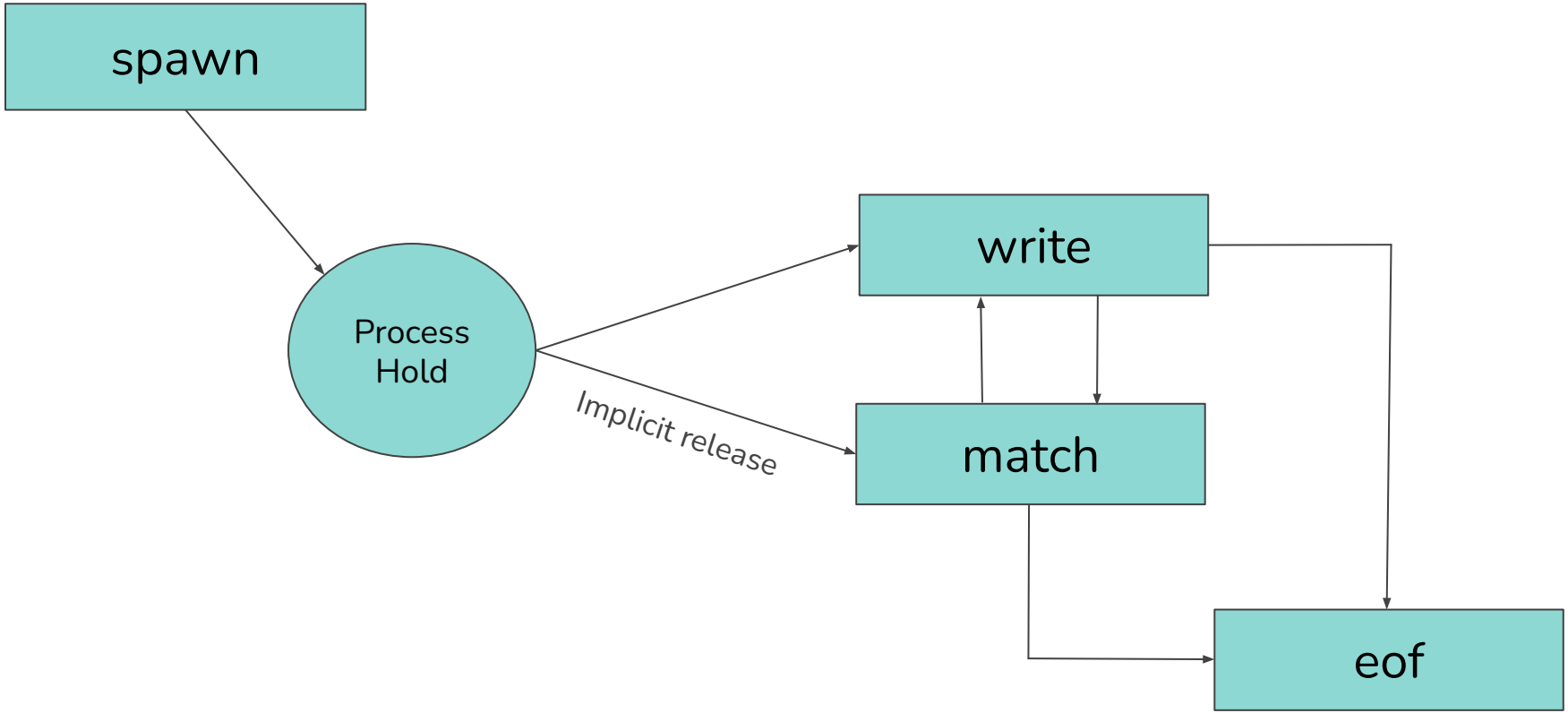
Testing challenges

- Avoiding racing a read(2) without requiring ptrace(2)
- Some test cases tedious to generate (a lot of input text)
- Want an easy way to send ctrl sequences without thinking about it
  - ^C, no magic numbers please (readability)
- Want stty manipulation (e.g., disable canonicalization)

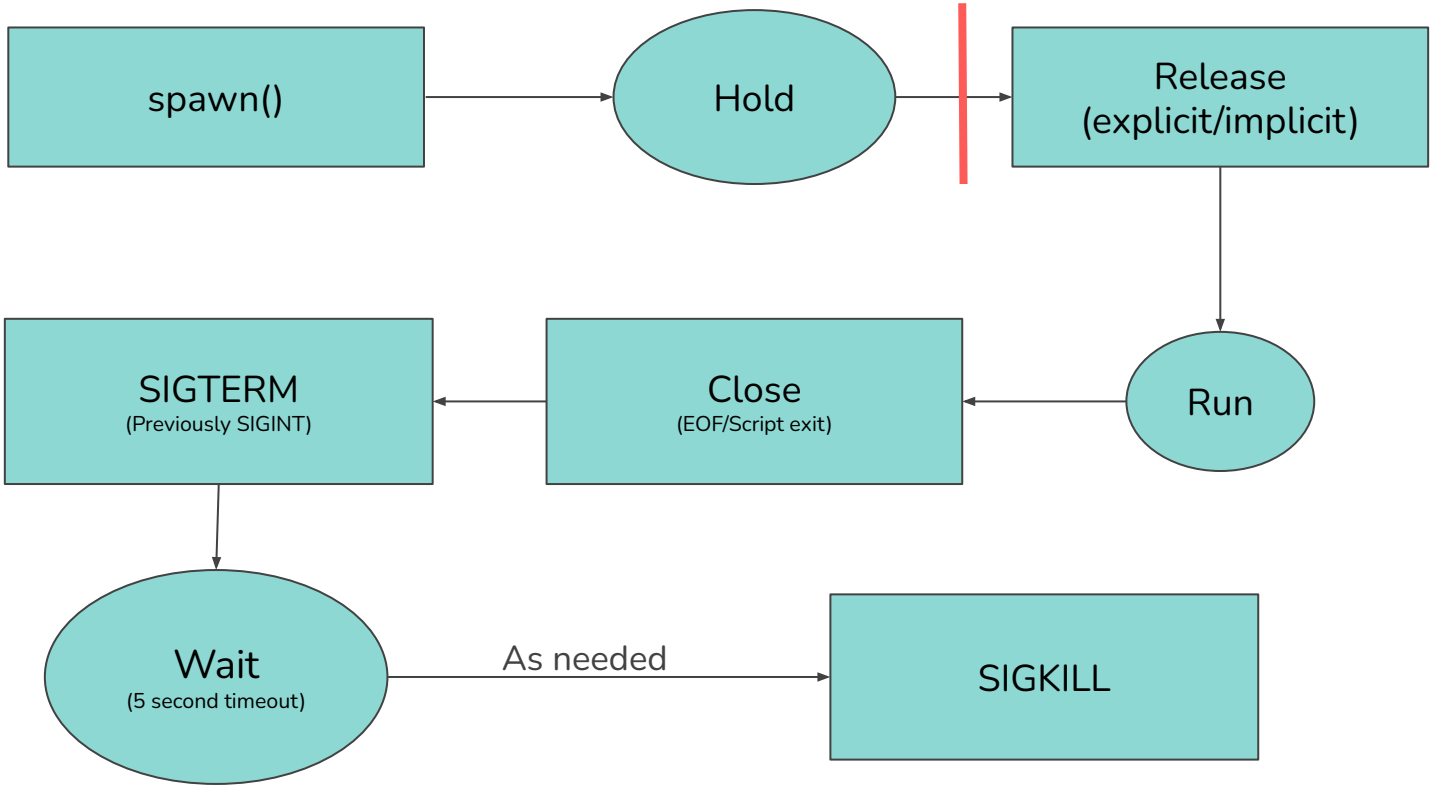
Design

Overview

- Scripted with Lua (5.3+ compatible)
- Features a scripted mode, as well as a 'direct execution' mode (exposed via lua lib)
- Portable
  - FreeBSD/macOS/Linux tested regularly via Cirrus and GitHub Actions
  - NetBSD/OpenBSD solutions welcome, currently verified manually pre-release
  - NetBSD 10.0+ for realpath(1) use in test suite
  - OpenBSD and NetBSD both fail one minor test due to missing `env -S` (running porch as script interpreter, filename goes to -f arg)



Execution Flow - User Model



Execution Flow - Internals

Overview - Process Configuration

Process

- match() timeout
- signals masked and ignored (**until release**)
- write speed (rate)

Terminal

- termios settings (cflag, iflag, lflag, oflag, cc)
- size

Scripted mode

- "orch" scripts
- Very limited environment
- Series of **actions** that get queued
- One process at a time
- spawn() implicitly closes any open process, kills it
- End of script closes an open process, kills it

Scripted mode - actions

- cfg (write delay)
- enqueue (scheduled callback)
- eof
- exec
- exit
- fail (error handler callback)
- flush
- getenv/setenv
- log (i/o transcript)
- matcher (lua, plain, posix (EREs))
- pipe
- stty
- raw
- release
- sigblock
- sigcatch
- sigclear
- sigignore
- signal
- sigreset
- sigunblock
- sleep (seconds)
- spawn
- write / match / one (match multiplexer)

Scripted mode - match / one

- match: basic pattern match
  - Can match multiple patterns
  - Earliest, longest match wins
- one: match multiplexer
  - Takes a callback
  - Callback should consist of a series of match() actions in order of precedence
  - First one to match wins
- one came first, match grew multiple patterns later; one might get deprecated

Scripted mode - exec

- Execute arbitrary command
- Allows user to collect output from said command as necessary
- Termination callback that we supply a wait status to
- Potential use-cases:
  - Testing for file existence
  - Kicking off non-interactive scripts that are necessary

Scripted mode - pipe

- Pipe input in from elsewhere
- io.popen() the specified command
- Read line-by-line, optionally applying a filter
- Write line to process
- Potential use-cases:
  - Extracting externally stored secrets
  - Fetching data from disk (io not available in the sandbox)

Scripted mode - signals

- Inspired by UNIX conformance requirements
- Testing applications that will send output in response to a signal, rather than exiting
- Collect a WaitStatus with eof() in case we need to check if the signal terminated the application properly
- Borderline out-of-scope of porch's original purpose, but useful functionality to have
- Signal mask configuration
  - sigblock
  - sigclear
  - sigunblock
- Configuration of signals caught/ignored
  - sigcatch
  - sigignore
  - sigreset (also clears the signal mask)

Scripted mode - debuggability

- fail() callback takes the remainder of the unmatched buffer as a param
- Without a fail() handler, prints out some diagnostics about the action we were trying to run
- debug() / hexdump() for outputting

"Direct" execution mode

- lib interface to some other lua script
- Exposes run_script() to run an orch script
- Exposes porch.spawn() to spawn a process
- Retains signal and terminal configuration functionality (adds sigis{blocked,ignored,caught,unblocked} functions to check the current status)

"Direct" execution mode - expected usage

- porch.spawn() returns a Process object
- Returned Process has most of the scripted actions defined on it
  - No "one" action, though! Only multi-match
- write/match to drive the Process to completion
- Multiple processes could be spawned

Scripted vs. Direct

- Scripted intended primarily for testing things
- Direct intended for arbitrary use
- Prefer to to keep built-in functionality to a minimum, with other interfaces wrapping the lib as needed
  - e.g., providing user interaction

Other Features

porchgen(1)

- Launches program
- Proxies output to stdout
- On user input:
  - Generates match/write statements
  - Passes input through to the application
- Leaves some previous match context just in case

rporch(1)

- Just another porch(1) executor
- Script is executed locally, as usual
- spawn() commands are executed via the rsh program, specified either via $PORCH_RSH in the environment or as an argument to **-e**, defaults to ssh
- Does word-splitting to allow arguments to rsh without requiring a wrapper script

porchfuzz(1)?

- **Not currently implemented**
- Fuzz-testing application input handling
- Unsure of best approach
  - Fuzz every input prompt
  - Fuzz specific input prompt
  - Combination of the two?
- Unsure of how to implement an effective fuzzer like this
  - e.g., no instrumentation/metrics to gauge whether a mutation was useful or not

Examples

Example: nc(1)

```
#!/usr/bin/env -S porch -f
--
-- Copyright (c) 2024 Kyle Evans <kevans@FreeBSD.org>
--
-- SPDX-License-Identifier: BSD-2-Clause
--

-- On the same machine, open up `nc -l 9999` and play with it.
spawn("nc", "localhost", "9999")

write "Hello from the other side\r"

-- Write any response on the listening side (don't forget to hit return), and...
match "." {
        callback = function()
                -- ...we'll shoot a debug message when it comes over.
                debug("We received a response!")
        end
}
```

Example: multi-match (cat)

```
12
13 write "Send One Two\r"
14 match {
15         One = function()
16                 write "LOL\r"
17
18                 debug "Matched one"
19                 match "LOL" {
20                         callback = function()
21                                 debug "lol"
22                                 write "Foo\r"
23                         end
24                 }
25                 -- Also valid:
26                 -- write "Foo"
27                 match "Foo" {
28                         callback = function()
29                                 debug "foo matched too"
30                         end
31                 }
32         end,
33         Two = function()
34                 debug "Called two"
35         end,
36 }
```

Example: parameterized tests

```
72
73 local function readsz_test(str, arg, expected)
74         spawn("readsz", table.unpack(arg))
75
76         if type(str) == "table" then
77                 assert(#str == 2)
78                 write(str[1])
79                 release()
80
81                 -- Give readsz a chance to consume the partial input before we send more
82                 -- along.
83                 sleep(1)
84                 write(str[2])
85         else
86                 write(str)
87         end
88         match(expected)
89 end
90
91 readsz_test("partial", {"-b", 3}, "^$")
92 readsz_test("partial^D", {"-b", 3}, "^par$")
93 readsz_test("partial^D", {"-c", 1}, "^partial$")
94 for s = 1, #"partial" do
95                 readsz_test("partial^D", {"-s", s}, "^partial$")
96 end
97 -- Send part of the line, release and pause, then finish it.
98 readsz_test({"par", "tial^D"}, {"-c", 1}, "^partial$")
99 -- line is incomplete, so we'll just see the "partial" even if we want two
100 readsz_test("partial^Dline", {"-c", 2}, "^partial$")
101 readsz_test("partial^Dline^D", {"-c", 1}, "^partial$")
102 readsz_test("partial^Dline^D", {"-c", 2}, "^partialline$")
```

Future Work

- Writing more tests for interactive stuff in base
  - tty behavior
  - tee(1) (SIGINT handling, more for simplicity)
- libporch (+ python interface)
- porchfuzz(1)

Questions?

kevans@FreeBSD.org

https://git.kevans.dev/kevans/porch

https://github.com/kevans91/porch
(public-facing mirror)