Rust Debugging Techniques

Robert O’Callahan
Pernosco
Context

- Fed up with state of debugging tools
- Created “rr” project at Mozilla
- Left Mozilla in 2016 to start Pernosco
- Pernosco: 113K lines of Rust
“Printf” Debugging

#[derive(Debug)]
struct Point { x: i32, y: i32 }

fn stuff(p: &Point) {
    println!("Point is: {{:?}}", p);
    ...
}
let a = 2;
let b = dbg!(a*2) + 1;

[src/main.rs:2] a * 2 = 4
Logging

use log::debug;
fn stuff(p: &Point) {
    debug!("Point is: {:#?}", p);
    ...
}

RUST_LOG=debug target/debug/test_crate
DEBUG 2019-05-06T02:34:20Z: test_crate: hello

Don’t forget env_logger::init()!
Assertions

- Make your code easier to debug by catching mistakes early

```r
assert!(remaining == 0);
assert!(remaining == 0, "{} remaining", remaining);
assert_eq!(remaining, 0);
```

- Good documentation

- Makes tests more powerful
Release Assertions

- `assert!` runs in release builds
- `debug_assert!` runs only in debug builds
- Pernosco mostly uses `assert!`
Backtraces

RUST_BACKTRACE=1 target/debug/test_crate
thread 'main' panicked at 'assertion failed: remaining == 0', src/main.rs:4:3
...
  6: test_crate::stuff
     at src/main.rs:4
  7: test_crate::main
     at src/main.rs:7
...

Levelling Up: gdb

- Lots of functionality, but non-discoverable

break  where  continue  print
run    commands  condition
watch -l  up    down  finish
next  step  nexti  steipi
disass  registers
info threads  thread
print user_function()
Rust gdb Tips

• Use gdb 8.2 or later for improved Rust support (prettyprinting enums)
• `break rust_panic_with_hook`
• `set lang c` when you get desperate
Debuginfo Quality

- LLVM’s DWARF debuginfo is poor in opt builds
Limitations Of Traditional Debuggers

- Debugging follows effects back to causes
- Traditional debuggers let you execute forwards, stop, inspect program state
  - “execute forwards”: wrong direction
  - “stop”: can break your application
- Traditional debuggers require multiple runs
  - Don’t work with hard-to-reproduce bugs
Record And Replay

- **Record** program execution without slowing it down much
- **Replay** recorded execution as many times as you want, with a debugger
- Simulate **reverse** execution
rr Overhead

![Bar chart showing overhead relative to baseline for different workloads: cp, octane, htmltest, sambatest. The chart includes bars for Record, Replay, and Single Core.](chart.png)
Reverse Execution

replay

Watchpoint hit
Watchpoint hit
Debugging With Reverse Execution

(gdb) watch -l mRect.width
(gdb) reverse-continue
nsIFrame::SetRect
(this=0x2aaadd7dbeb0, aRect=...)
718       mRect = aRect;

(gdb) reverse-next
Debug on Linux at all? Stop and go get `rr`
*RIGHT NOW*. Biggest improv. to debugging for me ever. H/T Justin Lebar.
Moves Vs Watchpoints

- Reverse execution with data watchpoints is an rr superpower
- Rust move-heavy code messes it up
  - Watchpoints stop at a move
  - Need to adjust address and continue :-(
“Printf” Vs gdb For Rust

- **Printf debugging:**
  - `#[derive(Debug)]`
  - Works well with optimized builds
  - Slow Rust compile times

- **Debugger:**
  - Rust nonoptimized builds are very slow
  - Debugging async code (futures) is horrible
Pernosco

• Is record-and-replay with reverse execution the ultimate in debugging?

NO!
“Point in time” debugging
→
Omniscient data analysis and visualization!
Omniscient Debugging

- Build an efficiently searchable database of all program states
  - E.g. all memory and register writes
    *ODB, Chronomancer, Chronon…*

- How to achieve acceptable overhead in real-world debugging situations?

- What is the ideal debugger UI when you drop “point in time” implementation constraints?
Demo
Time magic has unlimited potential.