Full Stack Debugging: From CI to ISS

Alexey Brodkin, Synopsys
October 26, 2020
Alexey Brodkin
Engineering manager @ Synopsys

Open Hub says:
• Most experienced in C
• First commit about 8 years ago
• Has made ~600 commits
• Most contributions to:
  – U-Boot
  – Linux kernel
  – Buildroot
  – Yocto Project / OpenEmbedded
  – Zephyr
  – uClibc
  – etc
Agenda

• Why Zephyr RTOS?
• The problem
• Going down the stack
• Getting back
• Lessons learned
Why Zephyr RTOS

Open source fully featured RTOS with first class support of ARC processors

• Synopsys ARC processors are supported in Zephyr RTOS from day 1
• Synopsys actively maintains Zephyr for ARC processors
• Powerful upstream CI: https://buildkite.com/zephyr/zephyr
  – Per pull-request build & run tests
  – Runs only on QEMU “boards”
• More targets of interest
  – Proprietary nSIM simulator
  – Real boards with ARC cores
    – in ASIC
    – in FPGA
• Internal CI built on top of Jenkins
Zephyr’s sanitycheck

How it really works

• Python script: https://docs.zephyrproject.org/latest/guides/test/sanitycheck.html
• Builds tests
• Executes tests
  – Starts test
  – Monitors console (stdout or real serial port)
  – Waits for “PROJECT EXECUTION {SUCCESSFUL|FAILED}” or kills execution after 60 seconds
• Collects statistics & generates reports:

  INFO - Total complete: 190/190 100% skipped: 37, failed: 17
  INFO - 136 of 153 tests passed (88.89%), 17 failed, 37 skipped with 0 warnings in 359.47 seconds
The problem

Some tests fail in the CI only

• Executed in Jenkins
INFO - 136 of 153 tests passed (88.89%),
17 failed, 37 skipped with 0 warnings in 359.47 seconds

• Executed on the same machine manually
INFO - 153 of 153 tests passed (100%),
0 failed, 37 skipped with 0 warnings in 362.55 seconds
Going down the stack

Looking for a root cause
Minimizing test-case

Shorter turn-around time allows for more experiments

• Initial test-case – full sanitycheck run on all nSIM virtual boards, ~30 minutes:
  – 1 min: west init
  – 2.5 min: west update

• Minimal test-case – 1 test on 1 platform in existing source tree, ~30 seconds:
  ./scripts/sanitycheck -p nsim_hs -t tests/subsys/logging/log_immediate/logging.log_immediate.clean_output

"Compiling", under CC BY-NC 2.5, originally posted here: https://xkcd.com/303/
Looking at logs

There might be something useful, at least some hints

- In logs we see:
  - ERROR - 126/190 nsim_hs .../logging.log_immediate.clean_output FAILED: Timeout

- With verbose mode ("-v -v -v"):
  - INFO - 126/190 nsim_hs .../logging.log_immediate.clean_output FAILED: Timeout (nsim 60.520s)

- No “PROJECT EXECUTION {SUCCESSFUL | FAILED}” in handler.log

  [00:00:01.460,000] test: test string printed 1 2 0x80000000
  [00:00:01.460,000] test: data:
  00 00 00 00 00 00 00 00 |........
  [00:00:01.460,000] test: test string printed 1 2 0x80000180
  [00:00:01.460,000] test: data:
  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ |........ |........ ......... ❯ Stops here
Check if that’s buffering of stdout

By default stdout of Python script is buffered, but that didn’t help

• **sanitycheck** is a Python script

• **sanitycheck** parses stdout of the simulator

• What if simulator output gets buffered?

• Let’s try to unbuffer it:
  – `#!/usr/bin/env python -u`
  or
  – export PYTHONUNBUFFERED=true

• Doesn’t help :(  

```bash
#!/usr/bin/env python
import sys;

sys.stderr.write("Print 1\n")
sys.stdout.write("Print 2")
sys.stderr.write("Print 3\n")
```
Check if the simulator is alive

strace -f -o trace.log nsimdrv ...
Check if the model of our CPU is alive

CPU is busy with something, instruction trace keep growing

• Check if the model of our CPU is alive
  – Dump target instruction trace
  – Check if we’re spinning in a tight loop

[x00000000] 0x8a74  [RB:0]  ADLK Z C  ldb_s  r3,[r2,0x14] : lb [x00000046c] => 0x00 : (w1) r3 <= 0x00000000 *
[x00000040fe] 0x4154  [RB:0]  ADLK Z C  ld_s  r1,[r2,0x8] : lw [0x00000460] => 0x80000020 : (w1) r1 <= 0x80000020 *
[x00000040fe] 0x80000001  [RB:0]  ADLK Z C  or_s  r1,r1,r3 : (w0) r1 <= 0x80000200 *
[x00000040fe] 0x80000001  [RB:0]  ADLK Z C  brne  r2,[r0,0x0] : lw [0x8000002c4] => 0x80000020 : (w1) r2 <= 0x80000020 *
[x00000040fe] 0x00000000  [RB:0]  ADLK Z C  st  0,[r0,0x0] : sw [0x0000002c4] <= 0x00000000 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  mov_s  r0,1,0x00000001 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  j_s  r0,0,0x38 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  seti  r14 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  mov_s  r1,0000c350 : (w0) r1 <= 0x0000c350 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  mov_s  r0,13 : (w0) r0 <= 0x000005b5 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  b  0xfffffe7d4 : (w0) r31 <= 0x000018a8 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  lsr_s  r2,r2,0x54 : (w0) r2 <= 0x000002da *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  brcc.d  r1,r2,0x54 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  norm  r2,r2 : (w0) r2 <= 0x00000000 *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  mov_s  r4,r1,r4 : (w0) r4 <= 0xffffffff *
[x0000000000] 0x710c  [RB:0]  ADLK Z C  sdc  r0,r0,0x0 : (w0) r0 <= 0xfffffffff *
Instruction trace analysis

Different returns from IRQ handler

- nSIM is instruction accurate (as opposed to QEMU)
  - Same program flow, including IRQs (as opposed to QEMU)
- Compare logs, 2 GiB each: vimdiff to rescue
- Divergence happens on just 387461 instruction

```
[0x00000242a] mov_s ilink1, r0 : (w0) r29 <= 0x00002b70
[0x00000242c] pop_s r0 : lw [0x80001820] => 0x00084602 : (w0) SP <= 0x80001824 : (w1) r0 <= 0x00084602
[0x00000242e] sr r0, 0xb : aux[STATUS32_P0] <= 0x84602
[0x000002432] rtie : PC <= 0x00002b70, STATUS32 <= 0x00000002, BTA <= 0x00000000

[0x000002b70] seti r14
[0x000002b74] ld_s r0, [r13, 0x8] : lw [0x80000460] => 0x80000200 : (w1) r0 <= 0x80000200
[0x000002b76] ld_s r0, [r0, 0x68] : lw [0x80000268] => 0xffffffff : (w1) r0 <= 0xffffffff

instruction_trace_local.log

387461
```

```
[0x00000242a] mov_s ilink1, r0 : (w0) r29 <= 0x00002b70
[0x00000242c] pop_s r0 : lw [0x80001820] => 0x00084602 : (w0) SP <= 0x80001824 : (w1) r0 <= 0x00084602
[0x00000242e] sr r0, 0xb : aux[STATUS32_P0] <= 0x84602
[0x000002432] rtie : PC <= 0x00002b70, STATUS32 <= 0x00000002, BTA <= 0x00000000

0x0001ec8 <_isr_wrapper>

[0x000001ec8] lr r0, [AUX_IRQ_ACT] : (w0) r0 <= 0x00000001: aux[AUX_IRQ_ACT] => 0x01
[0x000001ecc] ffs r0, r0 : (w0) r0 <= 0x00000000
[0x000001ed0] cmp r0, 0x0

instruction_trace_jenkins.log

387461
```
Going up the stack

We know who’s guilty, but don’t know why
Suspicious strace log

Not immediately wrong, but strange

• Why 2 PIDs for the simulator?
• Where do those poll() calls come from?
• Is that OK to poll() that often?
• Why reading nothing?
Look in the simulator sources

Let’s see where `poll(..., events=POLLIN | POLLPRI, ...)` comes from

- “git grep POLLIN” points to some `read_stdin()` function
- `read_stdin()` function is executed in a separate thread
- Additional PID!

```c
void read_stdin()
{
    struct pollfd fd;

    fd.fd = 0;
    fd.events = POLLIN | POLLPRI;

    while()
    {
        /* Wait for input */
        if (poll(fd, 1, 100500) <= 0)
            continue;

        ...
    }
}
```
Something’s wrong with the setup
Simulator infinitely polls stdin for nothing

- **poll()** expected to block until
  - There’s anything on input
  - Timeout (100500 seconds) expires
  - But...

- **poll()** returns 1 immediately
  - Nothing is in the input
  - This only happens in Jenkins
    - Not in olde good Free Style job but
    - In new fancy Pipeline Job

```c
void read_stdin()
{
    struct pollfd fd;

    fd.fd = 0;
    fd.events = POLLIN | POLLPRI;

    while()
    {
        /* Wait for input */
        if (poll(fd, 1, 100500) <= 0)
            continue;

        read(0, buf, sizeof(buf));

        /* Process received data */
        /* Update CPU IRQ */
    }
}
```
Meet “Durable Task” plugin in Jenkins

Payload run under nohup

• Automation people mention “Durable Tasks” being used in Jenkins Pipeline jobs
• More info about “Durable Tasks”:
  – https://plugins.jenkins.io/durable-task/
  – https://github.com/jenkinsci/durable-task-plugin

• Another hint: “Durable Task” uses nohup!
• nohup nicely connects /dev/null to the stdin, see:
  https://git.savannah.gnu.org/gitweb/?p=coreutils.git;a= blob;f=src/nohup.c;h=b6cb c8db920e9b8bbcf6f5ca5506243784a4b6b6;hb=HEAD#l116

/* If standard input is a tty, replace it with /dev/null if possible. */
What’s special about /dev/null on stdin

Reading from /dev/null returns E0F immediately

• That’s in the specification: "Single Unix Specification Section 10.1: Directory Structure and Files":

```
/dev/null
An infinite data source and data sink.
Data written to /dev/null shall be discarded.
Reads from /dev/null shall always return end-of-file (EOF).
```

• That explains our funny strace log
  – Immediately returning `poll()` in the simulator
  – Following `read()` returning 0 [bytes read]

• And here’s our minimal test-case:
  `nsimdrv -propsfile nsim_hs.props zephyr.elf < /dev/null`
Answers

We may now explain [almost] everything

- Why nobody faced that problem before?
  - Nobody ever tried to attach /dev/null to the stdin of the simulator

- How Jenkins affects simulator behavior?
  - Due to use of nohup, /dev/null gets attached to the simulator stdin

- Why reliably fail in IRQ handler?
  - Race in the simulator due to IRQ storm

- How to prepare minimalistic test-case for simulator engineers?
  - nsimdrv -propsfile nsim_hs.props zephyr.elf < /dev/null
Next steps

Improve the simulator

• Fix internal race on IRQ handling

• Accommodate /dev/null on input
  – Add check for real data availability
  – Add delay between polls

```c
void read_stdin()
{
    ...
    while()
    {
        /* Wait for input */
        if (poll(fd, 1, 100500) <= 0)
            continue;

        count = read(0, buf, sizeof(buf));
        if (count <= 0) {
            usleep(100500);
            continue;
        }

        /* Process received data */
        /* Update CPU IRQ */
    }
```
Lessons learned

- Be curious & persistent
- Get lucky
- Sources, documentation & knowledge base availability helps
- Talk to people
- Know your tools
Thank You