Cross debugging on Linux
Cross debugging on Linux
“A history, current state of the art, and coming improvements”
Tech Topics of this presentation

- Debugger mysteries
- Remote debugging
  - Serial
  - Network
  - Jtag
- Essential tools for the developer
- Debug on Zephyr OS
Remote Debugging

• What it is:
  – Debugging an application on a remote embedded target
  – Often of a different CPU architecture
  – Always considering `target ARCH != host ARCH` prevents number of bad assumptions leading to unexpected issues

• What it is not
  – Remote connection on target to run a local debugger (this does not fit on light embedded systems, and does not suit system mode debugging)
Remote Debugging: why people don’t use it

- Badly integrated in popular IDEs (VsCode)
- Has suffered from instabilities for years
- Why use a debugger at all, tell me?

**SCARY GDB 10**
Remote Debugging: System mode

- System mode = Kernel debug
  - Hardware (JTAG)
  - software (kgdb)
  - Software emulation (qemu-system gdb debug port)

- STOP → suspends the whole system
  - In pure software, switches control to a debug context, with IRQ disabled
  - Needs to implement a communication channel/driver (serial or ethernet polling)

- May have OS awareness (display of threads info: registers, backtraces, TLS ...)

Cross debugging
Remote Debugging: User mode

• User mode = user application debug
  – Needs kernel support: PTRACE API syscall
    https://man7.org/linux/man-pages/man2/ptrace.2.html
  – Mind the kernel version!
    • Better signal handling since linux 3.4
    • Seccomp handling since linux 4.8
  – And appropriate security credentials
  – Used by gdb, gdbserver
Software Breakpoints

• A software breakpoint is a special, architecture dependent, instruction that generates a software interrupt

• Example:
  - x86 → INT X
  - ARM: UND or BKPT or BRK

• Put in place of the code line to break to (text memory of process is modified)

• The debugger keeps a list of its breakpoints and of the original instruction
Hardware breakpoints

- Hardware breakpoints rely on special debug registers
- Their number, and usage, depends on the CPU
- 2 types
  - Instruction → exception is raised when the program counter matches the value in the register
  - Address → exception is raised when memory is read/written in a given range
- Tracee memory is **not** modified
SingleStep

• Singlestep consist in:
  – executing a single machine instruction
  – stopping immediately after

• Can be supported by the hardware, or not

• Can be offered by the kernel (PTRACE_SINGLEEP), relying on hardware support, or emulated.

• No support (MIPS) at all → trickiest issue! (see why later)
Emulating Singlestep

- Find out where next program counter will be (very architecture dependent)
- Place temporary instruction breakpoint at that position
- Continue the tracee
- Hit (hopefully) the breakpoint
- Remove the breakpoint by restoring the original instruction
  → tracee memory is thus modified twice !
Multithread debugging (user+system) nightmare

- In both cases, memory is the same address space
  - breakpoints intended to be hit by a specific thread will eventually be hit by another one
  - When this happens, the victim must be resumed
  - But for being resumed, the breakpoint has to be removed
  - And for doing so, all the threads must be suspended
  - Else the breakpoint can be missed!

- Things get even worse when emulating singlestep by inserting temporary breakpoints
Debugging with an IDE on Linux 2.6.x

- Wind River wanted a linux debugger (system+user) with the same features as Tornado for VxWorks
Everything began in ... 2005

- First POC of a unified (dual) software system/user debugger agent
  - Consisted in a set of ugly patches on top of 2.6.x kernel
    - VxWorks’ WDB protocol
    - Serial and Net polling routines (no NETPOLL yet !)
    - Ability to spawn processes from kernel context
Abandoned to switch to:

- Kgdb for system mode
  - Who uses it today?
- PTRACE Api + WDB protocol → WR “Usermode agent”
  - Udp / TipC / Serial backend
  - x86 / x86-64 / ARM32be / ARM32le / PPC32 / PPC64 / Mips32 / Mips64 support
  - Tracee IO in debug console
  - Support of hardware breakpoints (instructions + data)
  - Support ARM thumb mode and interworking mode
  - Support NPTL + LinuxThreads
  - Per thread fine grained debugging (stop/step/cont without stopping the whole process)
Achieved result

- 100 threads with lots of shared code being debugged, individually or not
- A set of +2000 Unit/Stress Tests run from host debugger
- Thanks to a design based on a FiniteStateMachine (more than 80 states)
- With a GUI!
- Internally used for developing … the JTAG probes firmware!
User Debugger State Machine

https://github.com/tbultel/gred
Caveats

- Stepping over ghost breakpoints is inappropriate design for system mode with SMP
- The continued project(s) (Linux+VxWorks) decided to use another trick
  - When stepping over a breakpoint, execute the instruction “somewhere else”, in order not to modify the memory (and miss breakpoints)
new life
chapter one
A remote debug session (x86-64)

• Setup :
  - Localbuilder container as host machine, running gdb
  - Redpesk OS in kvm as guest machine, running gdbserver
Cross debugging
Another Debug Session (x86-64 → ARM64)

• Setup:
  - Ubuntu host machine running gdb
  - Redpesk OS on Raspberry PI4 running gdbserver
Debugging a big process on RPI4

- Target process = OpenCPN
  - Uses X11 + OpenGL
  - Lots of shared libraries
  - C++

```
thierry@rpi:~$ ldd /usr/bin/opencpn | wc -l
176
thierry@rpi:~$ ```
Cross debugging
Alternatives to GDB/gdbserver

● LLDB
  - PROS: automatic upload of binaries
  - CONS:
    ● Bad experience on RPI
      - Was able to debug the ‘simple’ example
      - Crash of lldb-server when debugging ‘openCPN’
      - No connection lost detected
      - Unable to list/delete breakpoints
      - Documentation (eg ‘process launch’) is outdated or irrelevant for remote connection
      - Symbols completion is not user-friendly
Alternatives to GDB/gdbserver

- TCF (Target Communication Framework)
  - Project initiated by Wind River has the successor of usermode-agent
  - Integrated in Eclipse IDE (CDT)
  - Autodiscovery (UDP broadcast)
Debug on Zephyr

- Some words about Zephyr
- Not Eth, no software debug agent
  → JTAG is mandatory
- Setup:
  - Olimex probe on Renesas h3ulcb Cortex R7
  - OpenOCD
OpenOCD brings up a gdbserver on host, giving an abstraction of the JTAG-specific communication with the probe

- 1 core = 1 TCP port
- 1 additional TCP port for monitoring / management
- Zephyr tools make it even easier with “west debug”
Valgrind
You thought you knew valgrind, did you?

- Mostly known because of memcheck tool
- Its profiling tools are still not as popular as memory tools
- Usually impacts performances substantially
Valgrind profiling

valgrind --tool=callgrind [callgrind options] your-program [program options]

Generates callgrind.out.<PID>

That can be processed with kcachegrind
KCacheGrind

Cross debugging
When should I use Valgrind?

- All the time → Must be in CI!
- Attention must be paid for the memcheck exclusion rules
- See [https://valgrind.org/info/about.html](https://valgrind.org/info/about.html) for more reasons to do so
LTTng
LTTng

- “Instrument, trace, investigate”
- Kernel or User Space instrumentation
- Can save your life
- Examples:
  - I have a periodic task waking at a frequency of 50Hz (20ms), from time to time, it wakes up after 30ms, why? (any clues?)
  - Can diagnose priority-inversion issues
  - Chase IRQ storms
LTTng alternatives

- ftrace, strace, perf
- LTTng combines UserLand + kernel traces
LTTng

- Composed of 3 things
  - Lttng-tools
  - Lttng-modules (+ compliant kernel)
  - Lttng-UST
- The keys of success are the high performances buffers, and the standardized and performing output trace format (CTF, see https://diamon.org/ctf) that lead to ‘almost zero’ instrumentation side effects
  - RCU (https://en.wikipedia.org/wiki/Read-copy-update) technology (that brings the famous liburcu !)
  - SMP support
LTTng

• Putting the things together:
  – Follow the mole! (https://lttng.org/docs/v2.12/#doc-getting-started)

• Needed kernel options
  – CONFIG_MODULES
  – CONFIG_KALLSYMS
  – CONFIG_HIGH_RES_TIMERS
  – CONFIG_TRACEPOINTS (! not set on renesas yocto kernel!)

• A little parenthesis about CONFIG_TRACEPOINTS
  – Not displayed in menuconfig but activated by CONFIG_FTRACE
  – CONFIG_FTRACE adds 5 byte NOPs at each function entry (with no measurable performance impact)
LTTng in a nutshell

```bash
lttng create my-kernel-session -output=/tmp/my-kernel-trace
lttng enable-event --kernel --all
lttng start
<do some stuff, or wait a while>
lttng stop
```

```
/tmp/my-kernel-trace/
├── kernel
│   ├── channel0_0
│   ├── channel0_1
│   ├── channel0_2
│   ├── channel0_3
│   ├── channel0_4
│   ├── channel0_5
│   ├── channel0_6
│   └── channel0_7
│       └── index
│           ├── channel0_0.idx
│           ├── channel0_1.idx
│           ├── channel0_2.idx
│           ├── channel0_3.idx
│           ├── channel0_4.idx
│           ├── channel0_5.idx
│           └── channel0_6.idx
└── metadata
```
Cross debugging

LTTng Viewing

• Commandline : Babeltrace2
• Graphical: TraceCompass
  – Was formerly an eclipse plugin
  – Still at *eclipse.org* but is now a standalone application
LTTng: Trace Compass
Q&A

Lorient Harbour, South Brittany, France

Cross debugging
Links

• IoT.bzh:
  - Website: https://iot.bzh/
  - Publications: https://iot.bzh/en/publications
  - Github: https://github.com/iotbzh

• AGL:
  - Website: https://www.automotivelinux.org/
  - Documentation: http://docs.automotivelinux.org/
  - Sources: https://git.automotivelinux.org/