Linaro

LEADING COLLABORATION IN THE ARM ECOSYSTEM

Kernel Debug Stories

Leo Yan
Linaro Support and Solutions Engineering

Introduction

The Linux kernel provides scores of tools to assist with debugging. Every single one could (and usually has) been the topic for an entire hour of a conference schedule!

We have one hours to describe all of them! We should finish the material in about 60 minutes.

This session is a short introductory course on Linux kernel debugging. The course is will examine a number of different debugging challenges and discuss the techniques and tools that can be employed to overcome them. By focusing on stories rather than the minute details of each tool we can cover a lot of topics in a short space of time, providing a springboard for further independent study by trainees.

Overview

- The Basics
  - Tracing, profiling and stop-the-world
  - Failing early
- The Stories
  - I can’t reproduce but by customer can (and I hate flying)
  - My XYZ missed its deadline
  - My board just stopped dead

Tracing, profiling and stop-the-world

- Tracing
  - Gathering of **events** during system execution
  - Events often have a **timestamp** to assist interpretation
  - `printk()` is a (low performance) form of tracing
- Profiling
  - Gathering of **statistics** during system execution
  - Threads that dominate CPU, L2 cache-miss, etc.
  - Profiles can also be derived from detailed traces
- Stop-the-world/postmortem
  - Halt execution to **collect state information** useful for debugging
  - Traditional debuggers, such as gdb, are interactive stop-the-world debuggers
  - Postmortem analysis is a special case of stop-the-world where it is impossible to continue
  - Oops traces could be considered a form of automated stop-the-world analysis
- Combinations are powerful
  - Trace logs are part of the state that can be recovered by a stop-the-world debugger

Failing early

- Why fail early?
  - Many system trace tools use circular buffers ⇒ *must fail before evidence is evicted*
  - Bugs may “injure” the system ⇒ *best to fail whilst system is still alive enough to be analysed*
  - Symptom may be a second-order effect ⇒ *unearthing underlying cause directly saves effort*
  - Trace data can be huge ⇒ *failing early (or logging) helps us navigate the trace data*
- Many of the Linux debugging tools can **automate failing early**
  - Most tools report failures via `printk`: very defensively coded so it doesn’t fail easily
  - `git grep` makes it easy to find a `printk` with something more aggressive
- Can you recognise the symptoms of your bug in code?
  - If you can automatically recognise your bug you can sprinkle calls to your recogniser function all over the kernel in order to fail early
  - Ideally your recogniser code needs to spot first-order symptoms. You may need to debug for a bit to identify the nature of the damage.
  - Can also help you reason about how recently the system was damaged (which helps identify who)

Failing early - Common techniques

- Stack overflow detection
  - `FRAME_WARN` - *statically warning about large stack frames*
  - `SCHED_STACK_END_CHECK` - *check for stack overrun when a task deschedules*
- Memory debugging
  - `slub_debug=` - *selectively enable automatic bug detection, poisoning and tracing*
    - `SLUB_DEBUG` primarily impacts code size rather than performance (so leave it enabled)
  - `DEBUG_PAGEALLOC` - *use MMU to detect access to free pages (not on arm32)*
  - `PAGE_POISONING` - *fill empty pages with poison patterns (and validate pattern on realloc)*
- Lock debugging
  - `DEBUG_MUTEXES` - *sanity tests… relies on other tools to report deadlock*
  - `DEBUG_ATOMIC_SLEEP` - *shout if we try to sleep from atomic sections*
  - `DEBUG_LOCK_ALLOC` - *detect using locks after free*
  - `LOCKUP_DETECTOR` - *uses hrtimer irq as a watchdog (on non-ARM platforms also an NMI)*
- RCU stall detection

Failing early - Levels of intrusion

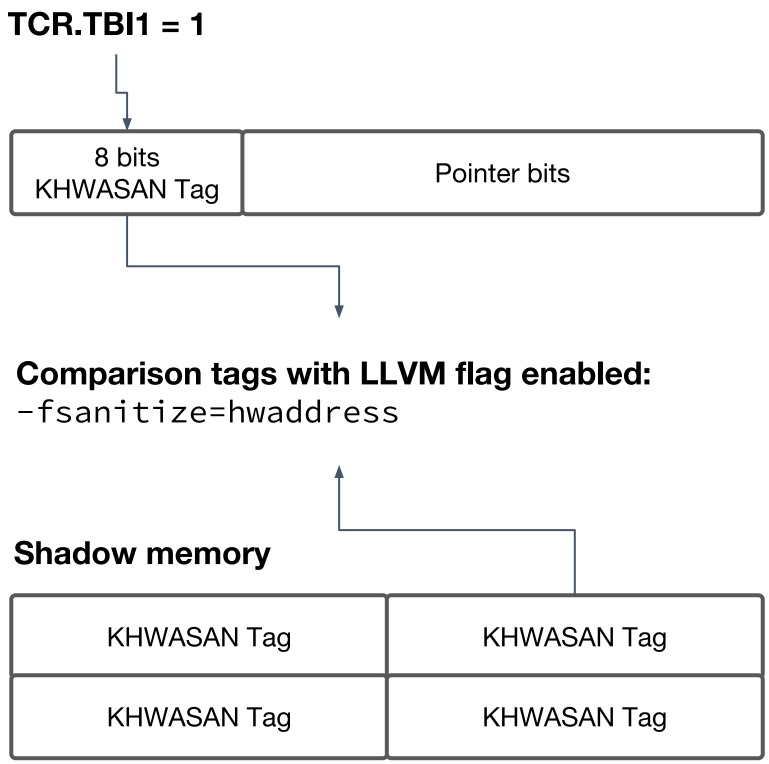
- Some runtime debug techniques may require significant CPU or memory
    - Intrusive debug tools can be very powerful and are capable of detecting errors quickly
    - The more resources the debug tool needs to more likely it is to alter the way a bug reproduces
    - For some (nasty) bugs even tiny instrumentation changes may alter or prevent reproduction. These are often called heisenbugs (despite the lack of quantum uncertainty in ARM arch.)
- Some tools cannot be run on low-resource embedded systems
    - Think about what the tool is designed to detect and whether it is likely the to help
    - Consider running test suites on a partially integrated system (e.g. sub-system or unit tests) to free up resources needed to run the tool
- Examples of useful, but expensive, debug tools
    - Poisoning (various) - *Poisoning costs can harm allocation intensive workloads*
    - `PROVE_LOCKING` - *Detect when two tasks take locks in differing orders (i.e. risk deadlock)*
    - `KASAN` - *Instrument all memory accesses and perform validity checks at runtime (no scribbles, kernel runs ~3x slower than normal)*
    - KHWASAN - *KASAN with hardware assistance*

Kernel hardware assisted address sanitizer (KHWASAN)

HWASAN exists in kernel and user space; by using the Top Byte Ignore arm64 CPU feature, we can store the a tag within pointers and, by using compiler instrumentation, we can verify the tag against shadow memory before a pointer is dereferenced.

KHWASAN requires less shadow memory than KASAN (1:16 rather than KASAN's 1:8). It also has no need for guard memory to detect overflow and can detect **use-after-free** bugs without extra cost.

KHWASAN bug detection is imprecise for two cases: 1. Won't catch some small out-of-bounds accesses, as the last byte of a slab object which in the same shadow cell; 2. Only have 1 byte to store tags, so have a 1/256 probability of a tag match for an incorrect access.



Overview

- The Basics
  - Tracing, profiling and stop-the-world
  - Failing early
- The Stories
  - I can’t reproduce but by customer can (and I hate flying)
  - My XYZ missed its deadline
  - My board just stopped dead

The story - I can’t reproduce but my customer can

“Everything ran great when I ran this on my desk. I’ve delivered the kernel to my customer and they keep seeing this odd behaviour. I don’t even have the equipment needed to reproduce this properly. I’ve already done three on-site visits this year and I could do with spending a few weeks nearer to home.”

Scope:

- “Odd behaviour” could be any of the behaviours we will discuss today (and more)
- Level of skill of the customer may differ from your own
- Customer is probably external (for internal customers tele-presence technologies such as VNC can be used between sites and allow joint investigation)

Notes:

- Debug cycles will be longer than usual
- Deployment of debug tools must be simple enough that customer can run useful experiments
- Need to be able to transfer trace/profile results back to your desk for analysis

Source navigation

I can't reproduce but my customer can

Why?

Effective source navigation is **always** important but for remote diagnosis its importance increases because we can **use the source navigator** to **avoid debug cycles.**

How?

Be sure to have one (or more) indexing tools integrated into your workflow

e.g. `make cscope` + editor integration (or cbrowser) + `git grep 'struct foo {'`

Regex is very effective at mapping `printk()` messages to source

`[    0.001636] xyz: Found 10 widgets` ⇒ `git grep "Found .* widgets"`

printk and dmesg

I can't reproduce but my customer can

- printk is an easy to use, **robust** and (almost) **always-on** trace system making it a critical tool for remote diagnostics
    - Study the existing log buffer and form one or more theories about possible failure
    - Enrich the log messages to prove/disprove each theory
    - Share with customer and cycle again
- Performance sucks on embedded systems with UART-based console handler
    - CPU spins waiting for UART meaning a half line of text at 115200 takes ~3ms to output
    - Heavy logging to console is very intrusive ⇒ beware of heisenbugs
    - Disable console (`quiet`) if it is possible to access the dmesg buffer (and set a large value for `LOG_BUF_SHIFT/log_bug_len=`)
- Understand `pr_debug()`
    - Disabled by default and outputs at `<8>` when enabled (console log level is typically `<7>`)
    - Can be statically enabled by adding #define DEBUG to a suspect compilation unit
    - Better to use `DYNAMIC_DEBUG`. This allows you to add very rich log messages for each debug cycle and propose multiple experiments, each with a different kernel command line.

AArch64 procedure call standard

I can't reproduce but my customer can

SP	**Stack pointer**
LR (r30)	**Link register (look here if the PC looks borked)**
FP (r29)	Frame pointer (optional within the ABI… but if this is a pointer value check memory here carefully)
r19..r28	Callee-saved registers
r18	Platform register (or additional temporary)
IP0/IP1 (r16-r17)	Intra-procedure-call scratch registers
r9..r15	Temporary registers
r8	Indirect result location register (for >16 byte results)
r0..r7	**Parameter/result registers**

How to read an oops

I can't reproduce but my customer can

Kernel reports bug with "**oops**", it's not only for panic but also for warning and assertion. An "oops" includes hardware and software related info dumping to assist analysis to output to console and save into syslog files.

Firstly, we can get clear what's exception type:
Internal or **external** exception;
Synchronous or **asynchronous** error;

Explore info from CPU system registers:
ESR_ELx - Exception Syndrome Register, holds syndrome information for an exception taken
pstate - Process state for condition flags, the asynchronous exception mask bits, Instruction set state, Endianness, Execution state, etc.
4 levels **page table** walk through - pgd, pud, pmd, pte

```
[  469.453498] Unable to handle kernel NULL
pointer dereference at virtual address 00000000
[  469.461969] Mem abort info:
[  469.464841]   ESR = 0x96000046
[  469.467924]   Exception class = DABT (current
EL), IL = 32 bits
[  469.473869]   SET = 0, FnV = 0
[  469.476944]   EA = 0, S1PTW = 0
[  469.480104] Data abort info:
[  469.482982]   ISV = 0, ISS = 0x00000046
[  469.486842]   CM = 0, WnR = 1
[  469.489836] user pgtable: 4k pages, 48-bit
VAs, pgd = 00000000400d56e0
[  469.496393] [0000000000000000]
*pgd=000000003ac45003, *pud=000000003b36e003,
*pmd=0000000000000000
[  469.505392] Internal error: Oops: 96000046
[#1] PREEMPT SMP
```

How to read an oops - cont.

I can't reproduce but my customer can

Check the bug happening context:
CPU id;
Task name and PID;
Is in interrupt context or not;
Which modules are linked;
Tainted;

The string '**Tainted:** ' indicates that the kernel has been tainted by some mechanism, e.g. flag **F** indicates if any module was force loaded by **`insmod -f`**; flag **L** indicates if a soft lockup has previously occurred on the system.

```
[  469.510967] Modules linked in: bnep hci_uart
adv7511 bluetooth crc32_ce crct10dif_ce cec
ecdh_generic dw_drm_dsi kirin_drm
drm_kms_helper drm ip_tables x_tables btrfs xor
zstd_decompress zstd_compress xxhash raid6_pq
[  469.530235] CPU: 1 PID: 2212 Comm: bash Not
tainted 4.15-hikey #1 kernel:
```

How to read an oops - cont.

I can’t reproduce but my customer can

Enable kernel configs `CONFIG_DEBUG_KERNEL` and `CONFIG_DEBUG_INFO` for adding debug info when compile kernel image.

We can check the regular registers `pc`, `lr` and stack backtrace to identify the line inside the Kernel's source code where the bug happened. We can use below two methods, one is based on gdb and another is based on addr2line command:

```
$ aarch64-linux-gnu-gdb vmlinux
(gdb) list *sysrq_handle_crash+0x20
(gdb) disassemble /s sysrq_handle_crash

$ aarch64-linux-gnu-addr2line -e vmlinux \
      0xffff00000d0e3d30
```

```
[  469.553704] pstate: 60000005 (nZCv daif -PAN -UAO)
[  469.558504] pc : sysrq_handle_crash+0x20/0x30
[  469.562861] lr : sysrq_handle_crash+0xc/0x30
[  469.567129] sp : ffff00000d0e3d30
[  469.570440] x29: ffff00000d0e3d30 x28: ffff80003b99e580
[  469.575755] x27: ffff000008ab1000 x26: 0000000000000040
[  469.581068] x25: 0000000000000124 x24: 0000000000000015
[  469.586382] x23: 0000000000000000 x22: 0000000000000007
[  469.591696] x21: ffff00000918f0e0 x20: 0000000000000063
[  469.597010] x19: ffff0000090e4000 x18: 0000000000000010
[  469.602324] x17: 0000ffffaf2e8110 x16: ffff00000821b190
[  469.607638] x15: 0000000000000006 x14: ffff00008927558f
[  469.612952] x13: ffff00000927559d x12: ffff0000090e3f88
[  469.618266] x11: ffff0000090e3000 x10: 0000000005f5e0ff
[  469.623581] x9 : ffff00000d0e3a50 x8 : 6172632061207265
[  469.628895] x7 : ffff000008587cf0 x6 : 00000000000001a9
[  469.634208] x5 : 0000000000000000 x4 : 0000000000000000
[  469.639523] x3 : 0000000000000000 x2 : ffff80003b99e580
[  469.644837] x1 : 0000000000000000 x0 : 0000000000000001
[  469.650152] Process bash (pid: 2212, stack limit =
0x000000008c610bdb)
[  469.656681] Call trace:
[  469.659126]  sysrq_handle_crash+0x20/0x30
[  469.663136]  __handle_sysrq+0x124/0x198
[  469.666972]  write_sysrq_trigger+0x58/0x68
[  469.671072]  proc_reg_write+0x60/0x90
[  469.674735]  __vfs_write+0x1c/0x118
[  469.678222]  vfs_write+0x9c/0x1a8
[  469.681536]  SyS_write+0x44/0xa0
[  469.684765]  el0_svc_naked+0x20/0x24
```

debugfs

I can't reproduce but my customer can

debugfs together with other virtual filesystems such as sysfs and procfs can be used to (automatically) collect information about the system.

Many sub-systems have special debugfs support, sometimes with their own config options to enable/disable it.

As an example, regmap provides direct access to register state for drivers that exploit it.

Try:

```
git grep REGMAP_ALLOW_WRITE_DEBUGFS
```

```
config DEBUG_FS
	bool "Debug Filesystem"
	select SRCU
	help
	  debugfs is a virtual file
	  system that kernel developers
	  use to put debugging files
	  into.  Enable this option to be
	  able to read and write to
	  these files.

	  If unsure, say N.
```

ftrace - Function tracing

I can't reproduce but
my customer can

Compile all code with -pg

```
<release_thread>:
stp     x29, x30, [sp,#-16]!
mov     x29, sp
mov     x0, x30
bl      ffff000008092ea0 <_mcount>
ldp     x29, x30, [sp],#16
ret
```

Kernel writes a nop over all the calls to mcount()

```
<release_thread>:
stp     x29, x30, [sp,#-16]!
mov     x29, sp
mov     x0, x30
nop
ldp     x29, x30, [sp],#16
ret
```

Kernel can dynamically enable/disable tracing

```
<release_thread>:
stp     x29, x30, [sp,#-16]!
mov     x29, sp
mov     x0, x30
bl      ftrace_caller
ldp     x29, x30, [sp],#16
ret
```

ftrace - Function tracing

I can't reproduce but my customer can

- ftrace is fairly lightweight… just a few nops when it is inactive
- Configuration can be supplied using kernel command line
  - `ftrace=function ftrace_filter=mydrv_* tp_printk`
  - `ftrace=function_graph ftrace_graph_notrace=rcu*,*lock,*spin*`
- Configuration can also be modified using debugfs

```
cd /sys/kernel/debug/tracing
echo function > current_tracer
echo 1 > tracing_on
```

- Trace information can be extracted in many different ways
  - Accessible from userspace - can be configured and gathered with a shell script
  - Automatically routed to printk - see tp_printk above
  - Trace is dumped automatically by kernel failure handlers - `ftrace_dump_on_oops`
  - Can be examined using kdb - useful if you have a non-serial console and need a pager
- Trace navigation: `trace_printk()`, KernelShark, LISA (by ARM)

kdump/crash

I can't reproduce but my customer can

kdump uses **kexec** to load a **dump-capture kernel** and the system kernel's **memory image** is **preserved** across the software reboot. It is exposed as `/proc/vmcore` and the (new) userspace can can copy this to a storage device or share it via the network.

```
console=ttyS0,1115200 …
                … crashkernel=128M
```

kexec tool called with `-p` will load dump-capture kernel into reserved memory ready to be jumped to during a kernel panic.

```
./kexec -p vmlinux --dtb=xxx.dtb
--append="root=/dev/mmcblk0p9 rw 1
maxcpus=1 reset_devices"
```

kdump images can be loaded by gdb but the **crash tool** provides more **powerful analysis tools**, including thread awareness, the capability to extract the ftrace buffer and more.

Other bespoke tools can be constructed to capture core images. This includes both scripts running in JTAG debuggers and "magic" bootloaders that recover RAM contents.

Note that **kdump for arm64** has been upstreamed in mainline kernel **since v4.12**. Full arm64 support in kexec-tools is also landed.

The story - My widget missed its deadline

"It's important that my widget is handled fast enough. At the moment when the system gets busy and my code misses deadline then we end up dropping frames. My QA team are beating me up because we promised a really smooth user interface"

Scope:

- "Missed deadline" could be an interrupt handler, tasklet, RT thread or regular task
- "When the system gets busy" could be system testing or a synthetic workload
- "My QA team are beating me up" suggests it is a system test that is revealing problems
- Kernel remains functional throughout… no problem accessing trace/profile buffers

Notes:

- Let's assume we can add code to the widget driver to detect when the deadline is missed
- Logging a message at point-of-failure will help us navigate the trace information
- Apart from the message at point-of-failure `printk()` is of little or no use for debugging this type of problem because it is too hard to decide where to add the extra log messages

ftrace - Alternative tracers

My widget missed its deadline

- Even a lightly loaded system will miss deadlines if there are long periods of interrupt lock or task priorities are poorly configured
- ftrace can show what the system was doing instead of meeting the deadline
- Function tracing could be used but this may be too intrusive because we'd likely have to instrument a lot of functions
- Let's look at some other tracers
  - irqsoff, preemptoff and preemptirqoff - Detect long periods of lock
  - wakeup and wakeup_rt - Detect long periods between task being made runnable and task executing
  - Exploit static tracepoints (this is advice from experts about what is "interesting")

    ```
    echo 'sched:*' > /sys/kernel/debug/tracing/set_event
    ```

perf

My widget missed its deadline

perf is a powerful profiling tool. Primarily it exploits the CPU performance counters but can also gather information from other sources (including hrtimers, static tracepoints and dynamic probes).

Performance counters can be free-run to count cycles, cache misses and branch misprediction, or they can interrupt after N samples to allow statistical profiling.

- perf stat - free-running event counts
- perf record - record events for later reporting
- perf report - decode a recorded trace
- perf annotate - annotate assembly or source
- perf top - real time analysis
- perf ftrace record - wrapper for ftrace

```
drt@birch:/home/drt/Development/Kernel/linux/tools/perf
Samples: 4K of event 'cycles:ppp', Event count (approx.): 130918717213723
Overhead  Shared Object                  Symbol
  91.13%  libpulsecore-10.0.so           [.] pa_asyncq_read_before_poll
   3.36%  libc-2.24.so                   [.] __libc_disable_asynccancel
   2.94%  i965_dri.so                    [.] 0x0000000000404865
   2.57%  liblzma.so.5.2.2               [.] 0x0000000000015195
   0.00%  [kernel]                       [k] module_get_kallsym
   0.00%  [kernel]                       [k] __bpf_prog_run
   0.00%  chrome                         [.] 0x00000000011d747e
   0.00%  perf                           [.] map__process_kallsym_symbol
   0.00%  [kernel]                       [k] kallsyms_expand_symbol.constprop.1
   0.00%  [kernel]                       [k] format_decode
   0.00%  perf                           [.] __symbols__insert
   0.00%  perf                           [.] rb_next
   0.00%  [kernel]                       [k] number
   0.00%  libglib-2.0.so.0.5000.3        [.] g_slice_alloc
   0.00%  perf                           [.] internal_cplus_demangle
   0.00%  [kernel]                       [k] vsnprintf
   0.00%  libc-2.24.so                   [.] _int_malloc
   0.00%  perf                           [.] rb_insert_color
   0.00%  chrome                         [.] 0x00000000011daa0c
   0.00%  [kernel]                       [k] __seccomp_filter
   0.00%  [kernel]                       [k] string
   0.00%  chrome                         [.] operator new[]
no symbols found in /usr/lib64/gstreamer-1.0/libgstcoreelements.so, maybe instal
```

Coresight and OpenCSD

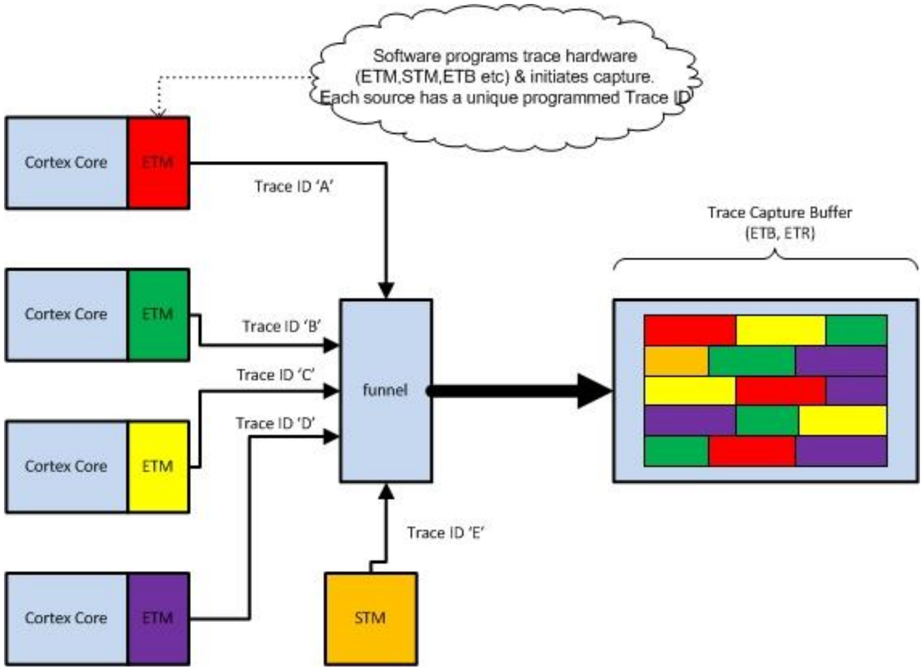
My widget missed its deadline

Traces **waypoints**:

- Some branch instruction
- Exceptions
- Returns
- Memory barriers

Similar power to ftrace but trace events are generated by the hardware.

OpenCSD library can parse Coresight trace data and is integrated into the perf tool.



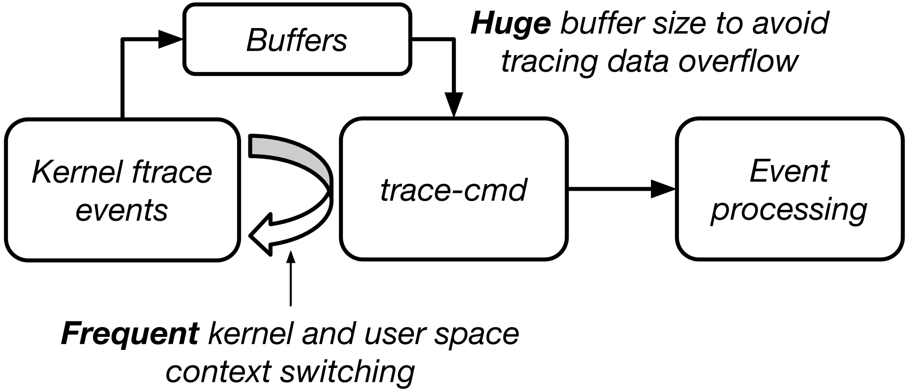
eBPF

My widget missed its deadline

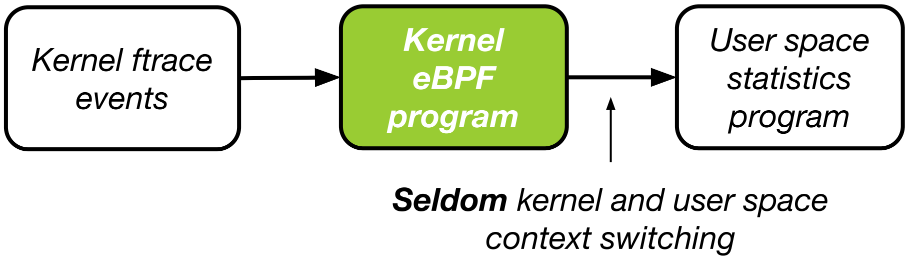
During the Linux 4.x series, eBPF's role has expanded from network packet filtering to be much more widely used across the kernel. For debugging purpose, it allows user code to be attached to kernel events (e.g. kprobe, ftrace) for dynamic filtering or statistics gathering.

To efficiently process the kernel event stream, we write C based program and pass resulting bytecode into kernel; in the kernel a simple JIT will convert the eBPF bytecode to CPU native instructions.

Without eBPF



With eBPF



The story - My board just stopped dead

"It was running fine and then it just stopped dead. There are no more console messages, I can't even figure out how grab the extra debug messages in my dmesg buffer."

Scope:

- The scope of this story is very wide
- We've got no clues about why the machine is not responding
- However, for now we will assume that the hardware itself it not faulty (e.g. reliable memory, reliable bus, etc)

Notes:

- The details of this story are extremely device specific
- The board state (up to and including bus health and power state) is unknown
- We have to understand how certain debug tools work to reason about how deeply injured the board is

Initial analysis

My board just stopped dead

- Fail early…
  - Theory: *the console still works but nothing is doing any logging…* how can we test that?
  - Set `initcall_debug` if the problem occurs during boot
  - If `initcall_debug` identifies `clk_disable_unused()` then set `clk_ignore_unused`
  - Enable (at least) `DEBUG_SPINLOCK`, `DEBUG_MUTEX` and `LOCKUP_DETECTOR`
  - Try instrumenting the GIC driver with a rate limited print (to show if we have locked up servicing an interrupt)
- JTAG debugger
  - A **JTAG debugger** is the **ideal tool** to investigate the system when it has failed like this
  - AArch64 support was recently merged into OpenOCD master branch and GDB_SCRIPTS has been a kconfig option since v4.0
  - Debugger will be delicate after serious failures such as bus hang and **be extra careful** if the problem is difficult to reproduce… it could take days before you get another shot at this!
  - Study **CPU register state** first before studying memory mapped registers or RAM
  - Learn how to **extract dmesg and ftrace buffers** using your debugger. It can also be useful to learn to connect without resetting the target (hot-connect).

An aside - SoC level debug

My board just stopped dead

- Almost all Linux debug tools are software based
  - **Bus hangs kill software** so kernel engineers should also study SoC level debug techniques
- Non-CPU bus initiators
  - Can provide clues about what hardware is still functional (e.g. an image on the LCD panel implies DDR controller remains functional)
  - SoC built-in coprocessors and controllers can often gather system information from anything memory mapped (perhaps even provide peek/poke via a serial port owned by co-processor)
  - Linux can return the favour… make sure your co-processor driver can gather a core dump
- Post-reset memory recovery
  - With a little hacking trace tools can target on-chip SRAM
  - Memory contents may survive reset if the bootloader brings up the DDR controller fast
  - Caches frustrate post-reset memory recovery (need to hack cache flushes into tracing code)
- Bus debug registers
  - Many SoCs contains registers to help understand bus hangs but…
  - … they are almost always secret so I can't help, you need to discuss this in-house

ftrace

My board just stopped dead

- ftrace could let us know what was happening just before the failure, if only we could see the trace buffer
- We’ve already talked JTAG debuggers… and they are already **cache coherent**
- Debug tools hate caches
  - **RAM** often **survives a reset** if the bootloader reconfigure things **fast**
  - Accessing RAM from other bus initiators may not be cache coherent
  - Modern L2 (and L3) **caches are** large, will be **destroyed by a reset**

Ramoops

My board just stopped dead

Trace

Ramoops is general framework to dump logs into persistent RAM, the RAM is reserved at boot time and with non-cacheable mapping; the ramoops buffer can survive after a restart.

It can support to dump console message, oops and panic log, and support function tracing:

```
ramoops.mem_address=0x21f00000
ramoops.mem_size=0x100000
ramoops.record_size=0x20000
ramoops.console_size=0x20000
ramoops.ftrace_size=0x20000
```

The function tracing can introduce serious performance degradation: testing ‘hackbench’ the completion time extends from 0.6s to 2m4s!

Post-mortem

After system reset (e.g. triggered by watchdog), we need to mount ‘pstore’ virtual file system to retrieve dump data.

```
mount -t pstore pstore /mnt
```

The function tracing data can be used for analysis and find out suspicious point.

```
CPU:7 ts:312658 ffff00000809b664  ffff00000865d0e8
__iounmap <- plat_dis_clock+0x5c/0x6c
CPU:7 ts:312659 ffff0000082156c4  ffff00000809b688
vunmap <- __iounmap+0x38/0x48
CPU:7 ts:312660 ffff000008215504  ffff0000082156e4
__vunmap <- vunmap+0x34/0x48)
```

Coresight

My board just stopped dead

Trace

CoreSight trace doesn’t have to be captured on the device itself (if it is cache flush hacks will be required). Instead it can rerouted off-chip and captured with specialist hardware.

Trace implementation varies widely between manufacturers; need to talk to your SoC experts.

If your internal tools are immature consider integrating OpenCSD to decode the CoreSight trace information. OpenCSD licensing (BSD 3-clause) permits wide reuse of this code.

Post-mortem

The cell that implements trace will, if powered, receive PC trace events and store them in its register state. This happens even when the trace is not being written to memory.

Last PC before failure is stored here and can be extracted from these registers.

CORESIGHT_CPU_DEBUG: Other processors can extract this too (no cache problems)! This kernel config allows Linux SMP partners to watch each other (enhanced LOCKUP_DETECTOR).

```
coresight-cpu-debug f6590000.debug: CPU[0]:
coresight-cpu-debug f6590000.debug:  EDPRSR:  00000001 (Power:On DLK:Unlock)
coresight-cpu-debug f6590000.debug:  EDPCSR:  [<ffff00000808f22c>] handle_IPI+0x1ac/0x1b8
coresight-cpu-debug f6590000.debug:  EDCIDSR: 00000000
coresight-cpu-debug f6590000.debug:  EDVIDSR: 90000000 (State:Non-secure Mode:EL1/0 Width:64bits VMID:0)
```

Memory barrier

I can't reproduce but my customer can

ARM and ARM64 use weak memory model, if the memory mapping is device type, it isn't the same thing with strong order type so it might be out of order for registers accessing.

In kernel, it isn't suggested to define the register variable with `volatile`, insteadly we should use below APIs for register accessing:

```
__raw_writel()/__raw_readl()
writel_relax()/readl_relax()
writel()/readl()
```

`readl()/writel()` variants are most safe APIs with endian conversion and memory barriers.

Below code have potential issues:
`reg_b` maybe is written to the device early than **`reg_a`**, this is caused by the early acknowledge;

Device register accessing may have conflict with normal memory, so variable **`i`** may get the stale value;

```
volatile u32 *reg_a = (u32 *)0xfe000000;
volatile u32 *reg_b = (u32 *)0xfe000004;
volatile u32 *reg_c = (u32 *)0xfe000008;
u32 *mem_data = (u32 *)0xc5000000;
int i;

*reg_a = 0x12345678;
*reg_b = 0x87654321;

while (*reg_c == 0);

i = *mem_data;
```

Graphic by Jost, CC0-PD

Linaro Limited Lifetime Warranty

This training presentation comes with a **lifetime warranty**.

Everyone here today can send **questions** about today's session and **suggestions** for **future topics** to support@linaro.org .

Members can also use this address to get support on any other Linaro output. Engineers from **club** and **core** members can also contact support to discuss how Solutions and Support Engineering can help you with additional services and training.

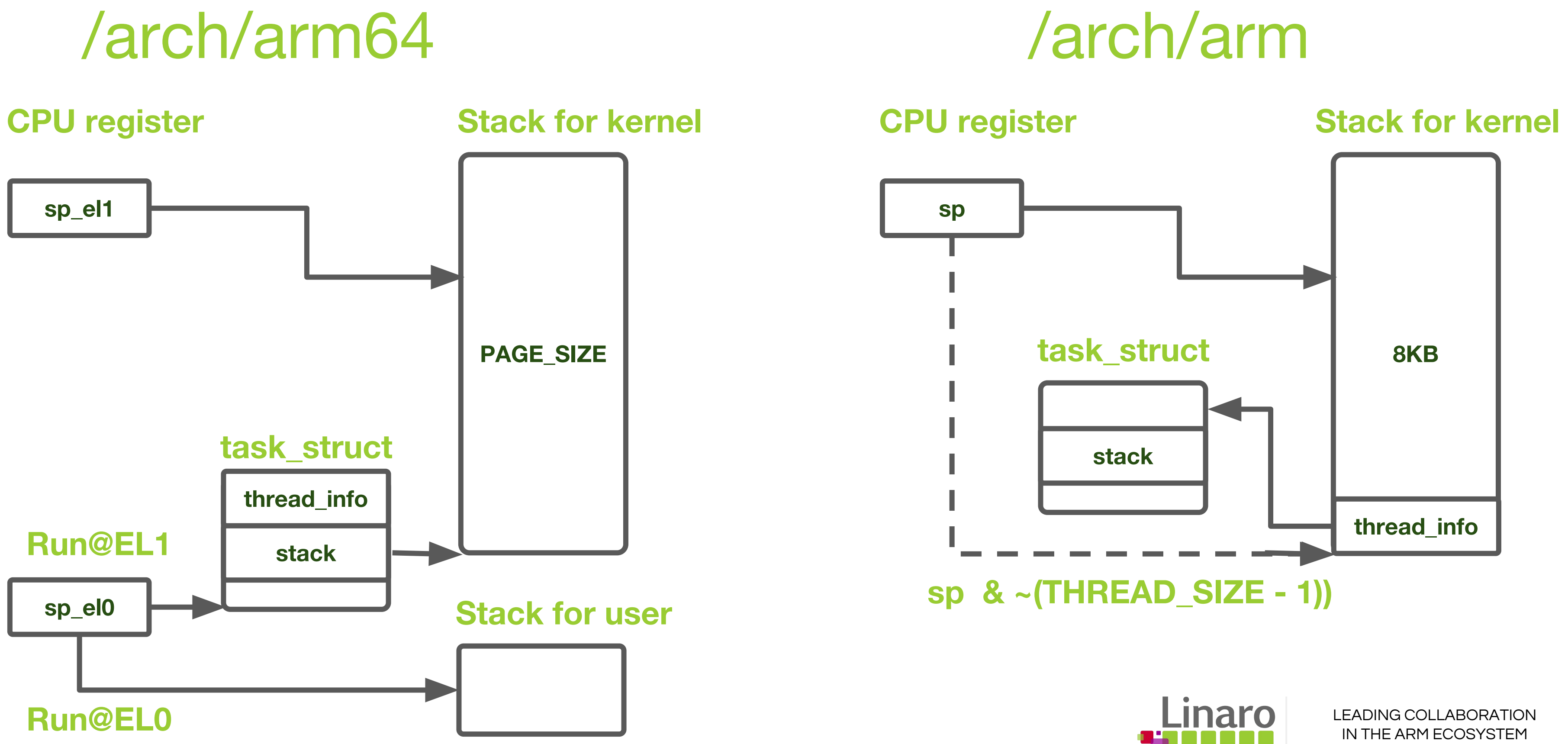
Thanks to Andrew Hennigan for introducing me to the idea of placing a guarantee on training.

Thank You

For further information: www.linaro.org

Backup

Kernel stack



ARM64 memory map (4kb page + 4 level)

#	Name		Start address	End address	Size	Memory mapping attribution
1	User		`0x0`	`0x0000ffffffffffff`	`256TB`	NORMAL
2	modules		`0xffff000000000000`	`0xffff000008000000`	`128MB`	NORMAL
3	Vmalloc	map_vm_area	`0xffff000008000000`	`0xffff00000807ffff`	`512KB`	DEVICE_nGnRnE: DMA coherent memory region or ioremap NORMAL: vmalloc
		.text .rodata .init .data/.bss	`0xffff000008080000` *KASLR causes this address to change*	Kernel image specific	`10MB+`	NORMAL
		map_vm_area	`Kernel image specific`	`0xffff7dffbfff0000`	`~126TB`	DEVICE_nGnRnE: DMA coherent memory region or ioremap NORMAL: vmalloc
4	fixed		`0xffff7dfffe7fd000`	`0xffff7dfffec00000`	`4MB + 12KB`	DEVICE_nGnRE
5	PCI I/O		`0xffff7dfffee00000`	`0xffff7dffffe00000`	`16MB`	DEVICE_nGnRE
6	vmemmap		`0xffff7e0000000000`	`0xffff800000000000`	`2048G`	NORMAL: struct page array
7	directly mapped kernel memory		`0xffff800000000000`	`0xffffffffffffffff`	`128TB`	NORMAL: kmalloc

Convert binary code to instructions

Step 1: If the problem is related with a runtime modified instruction sequence, we may need to decode the `Code:` section of an oops trace.

```
[  469.688341] Code: 52800020 b9025020 d5033e9f
d2800001 (39000020)
[  469.694446] ---[ end trace ea89eb9b9e0b2b48 ]---
```

Step 2: Directly assemble the binary code.

```
     .text
     .globl foo
Foo:
     .word 0x52800020
     .word 0xb9025020
     .word 0xd5033e9f
     .word 0xd2800001
     .word 0x39000020
```

Step 3: Use objdump to dump AArch64 instructions.

```
$ aarch64-linux-gnu-gcc -c -o foo.o foo.s
$ aarch64-linux-gnu-objdump -d foo.o

foo.o:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <foo>:
   0:      52800020  mov  w0, #0x1
   4:      b9025020  str  w0, [x1,#592]
   8:      d5033e9f  dsb  st
   c:      d2800001  mov  x1, #0x0
  10:      39000020  strb w0, [x1]
```