Kernel Debug Stories for Arm

Logistics

Questions welcome! There is time allocated for Q&A at the end of today's session but you can ask relevant questions in the chat box as we go.

Slides can be downloaded from:

https://drive.google.com/drive/folders/1uQqtQBYz59XSy7Bzm_v6s7Wqlg_eEeoo

https://people.linaro.org/~leo.yan/training/webinar_kernel_debug_stories/ (If you cannot access Google drive)

Biography: Daniel Thompson

I started my career at STMicroelectronics where I joined the toolset team as a real time operating system engineer. As the set top box industry pivoted from custom RTOS to GNU/Linux I moved along with it.

During the following years I worked across the set-top-box software stack from networking to audio/video codecs and from C libraries to software architecture. For my final years at ST I was brought in as tech lead to bring a project to replace the entire ST driver stack back on track. During that time my role was largely consultative and involved supporting and helping team members to be more productive and to solve problems quickly. Switching to a consultative role led, among other things, to my interest in training and mentoring.

I joined Linaro in 2014, initially in the Kernel Working Group and later as the tech lead for the Support and Solutions Engineering team. My work at Linaro is varied but all of it is centred around my background in tools and operating systems, whether that is my role leading the support team, as the Linaro training manager or as the upstream KGDB maintainer.

Biography: Leo Yan (严念)

I started my first job for Hisilicon Ltd. where I worked on middleware and porting graphic system on Linux system for two years.

Then I joined Marvell mobile group. My first project in Marvell was to develop TrustZone software for xScale Armv6/v7 SoCs, then moved on to work on drivers (audio driver), and reusing my knowledge of TrustZone monitor I engaged tasks for Arm big.LITTLE and Arm trusted firmware on Armv8 SoCs. At that period, I also got some sense for CPU power modeling and tuning.

In 2015 I joined Linaro and I work in the Support and Solutions Engineering team. My work includes support members via LDTS, delivering training and engage several open source projects: EAS/IPA, 96boards Hikey/Hikey960 related development, and enabling Arm CoreSight/SPE with perf for Linux mainline kernel.

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 two hours to describe all of them!

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 study.

Linaro

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
  - I’m sure this used to work
  - My board just randomly failed
- The Wrap up (and free gift)

Linaro

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 can “damage” 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 **automate failing early**
  - Most tools report failures via `printk()`: very defensively coded so it doesn’t fail easily
  - `git grep` makes it easy to map a `printk()` to the corresponding source code
- 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*
  - `VMAP_STACK` - *enable virtually mapped stacks (with guard pages)*
- 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, uses an eighth of available memory, heaps are less RAM efficient and kernel runs ~3x slower than normal)*
  - KHWASAN - *KASAN with hardware assistance (arm64 only with TBI, MTE)*

Failing early - PROVE_LOCKING

When debugging we turn on `PROVE_LOCKING` because we suspect a deadlock... but why didn't that already happen during QA? Debugging could also drive process improvement!

Thread #1	**Thread #2**
`lock(subsys)`	`lock(driver)`
`lock(driver)`	`lock(subsys)`
	<deadlock>

Timing windows for deadlock is small so this is unlikely to be discovered during development or by sub-system testing.

Likely to be discovered during final QA or in the field. Fault will be hard to diagnose.

Thread #1

```
lock(subsys)
lock(driver) // remember this lock sequence
unlock(driver)
unlock(subsys)
lock(driver)
lock(subsys) // remember this lock sequence
```

<PROVE_LOCKING reports bug>

With `PROVE_LOCKING` we no longer have to hit the deadlock timing window to show that a deadlock is possible, we just need our workload to exercise both code paths.

Linaro

Kernel hardware assisted address sanitizer (KHWASAN)

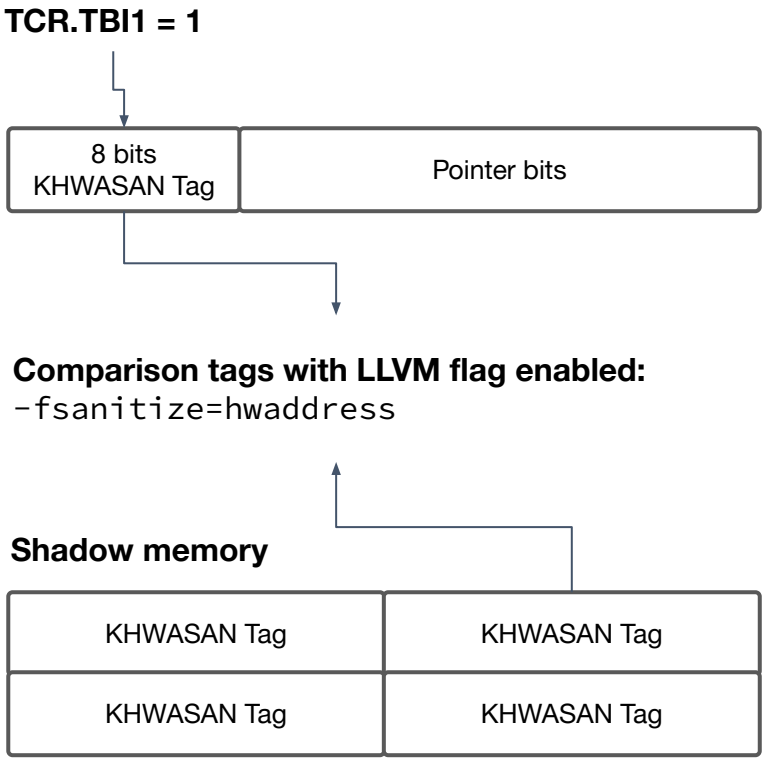
KHWASAN is memory debug method implementable in either kernel or user space: by using the Top Byte Ignore arm64 CPU feature, we can store the a tag within pointers and, by using compiler instrumentation, we can verify the tag against shadow memory before a pointer is dereferenced.

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 does accept some imprecision:

1. Won't catch some small out-of-bounds accesses, such as the last byte of a slab object (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.



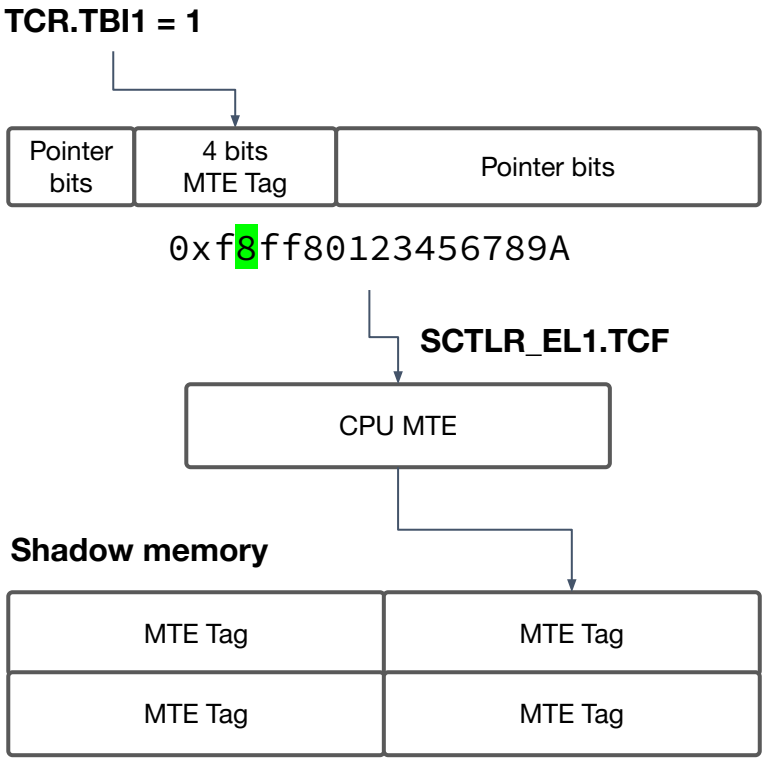
KHWASAN with Arm Memory Tagging Extension

From Armv8.5 and Armv9 start to support Arm memory tagging extension (MTE), which provides a hardware mechanism for memory tagging checking and reports tag mismatching by CPU.

MTE relies on Top Byte Ignore and uses 4 bits for memory tagging, and it provides dedicated instructions for tagging operations (generate random tag value, store and load tag in the shadow memory, etc). It's configurable for tag checking modes: sync mode (precise but slow), async mode (imprecise but fast) and mixed mode (sync for reads, async for writes).

MTE is expected to avoid the significant extra resource:

- Memory tags will use 3% of RAM based on 4-bits tag for each 16 bytes
- In worst case scenario for sync mode, it downgrades 10% performance.



Failing early *in production* - KFENCE

KFENCE - “**kernel electric fence**”. Named in **homage to** electric fence, **a userspace memory debugger written** by Bruce Perens when he was working for Pixar **in the late eighties**.

KFENCE (and electric fence) **uses the MMU** to detect use-after-free, invalid-free and out-of-bounds access **by placing inaccessible pages** after allocated memory blocks and by removing recently freed memory blocks from the address space.

This technique is **powerful but** when applied to all allocations this technique results in **significant memory overhead** because every allocation requires at least 4K RAM and 8K of address space.

KFENCE is low-overhead, making production usage possible, because it **does not instrument all allocations**. Instead it instruments a subset of allocations and is, in some ways, similar to a sampling profiler. It only triggers when a timer has expired and the total instrumented allocations is “small” (by default small means <255 objects, <2MB usages).

“KFENCE trades performance for precision. The main motivation behind KFENCE's design, is that with enough total uptime KFENCE will detect bugs in code paths not typically exercised by non-production test workloads.” - KFENCE documentation

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
  - I'm sure this used to work
  - My board just randomly failed
- The Wrap up (and free gift)

Linaro

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"`

Linaro

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 and a half line of text at 115200 takes ~3ms to output
  - Heavy logging to console is therefore very intrusive ⇒ beware of heisenbugs
  - Disable console (quiet) if it is possible to access the log buffer via `dmesg` (and set a large value for `LOG_BUF_SHIFT`/`log_buf_len=`)
- Understand dev_debug()/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.

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.

```
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.
```

Try:

```
git grep REGMAP_ALLOW_WRITE_DEBUGFS
```

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
```


Linaro

ftrace - Function tracing

I can't reproduce but my customer can

- ftrace is fairly lightweight… just a few nops when it is inactive
- Once enabled in Kconfig it can be configured using debugfs

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

- Configuration can be supplied using kernel command line

```
ftrace=function ftrace_filter=mydrv_*
ftrace=function_graph ftrace_graph_notrace=rcu*,*lock,*spin*
```

- Trace information can be extracted in many different ways
  - Accessible from userspace - can be configured and gathered with a shell script
  - Some trace can be automatically routed to printk - `tp_printk`
  - Trace is dumped automatically by kernel failure handlers - `ftrace_dump_on_oops`
  - Can be examined using kdb - useful if you have no other scrollback mechanism and need a pager
- Trace navigation: `trace_printk()`, trace-cmd, 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=ttyS0,115200 …
                … 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.

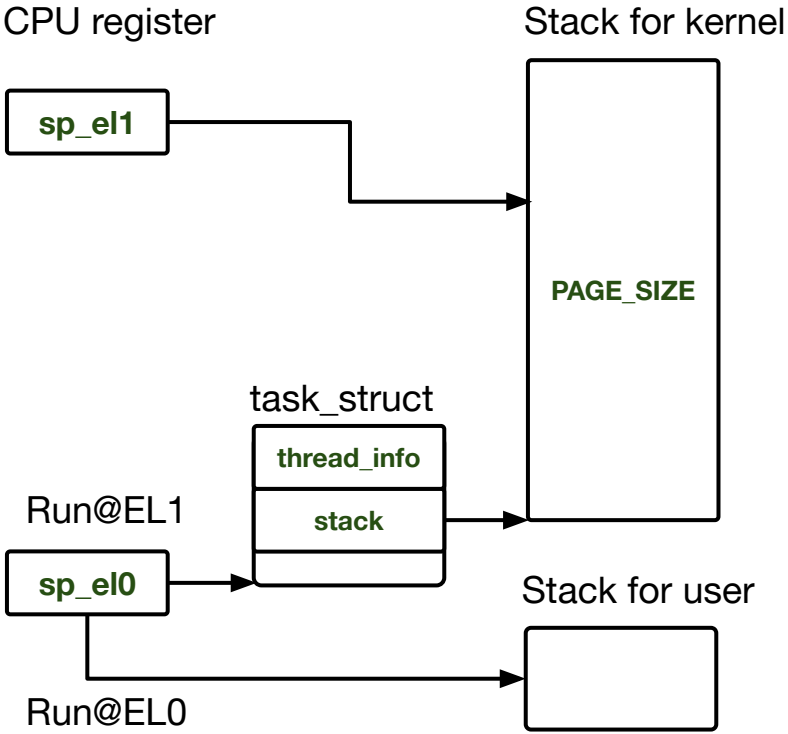
ARM64 memory map (4kb page + 4 level)

#	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	0xffff7dfffe00000	16MB	DEVICE_nGnRE
6	vmemmap		0xffff7e0000000000	0xffff800000000000	2048G	NORMAL: struct page array
7	directly mapped kernel memory		0xffff800000000000	0xffffffffffffffff	128TB	NORMAL: kmalloc

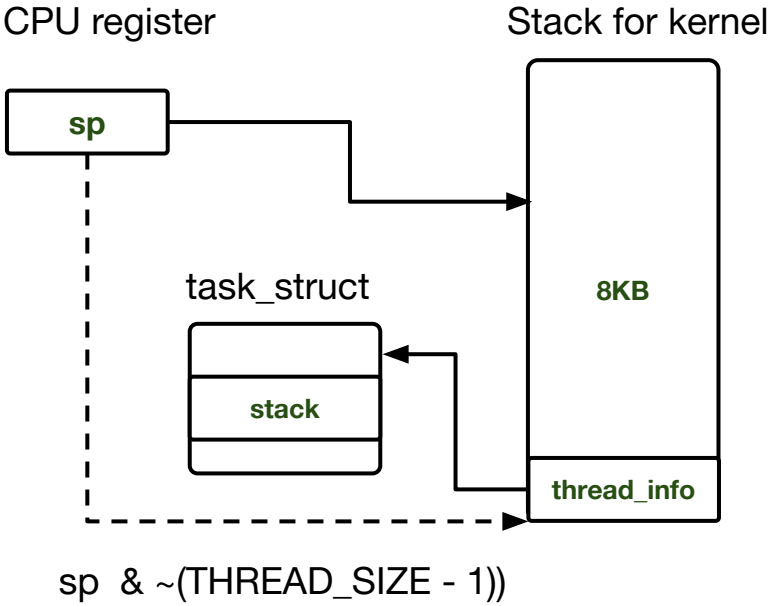
Kernel stack

I can't reproduce but my customer can

/arch/arm64



/arch/arm



Linaro

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
```

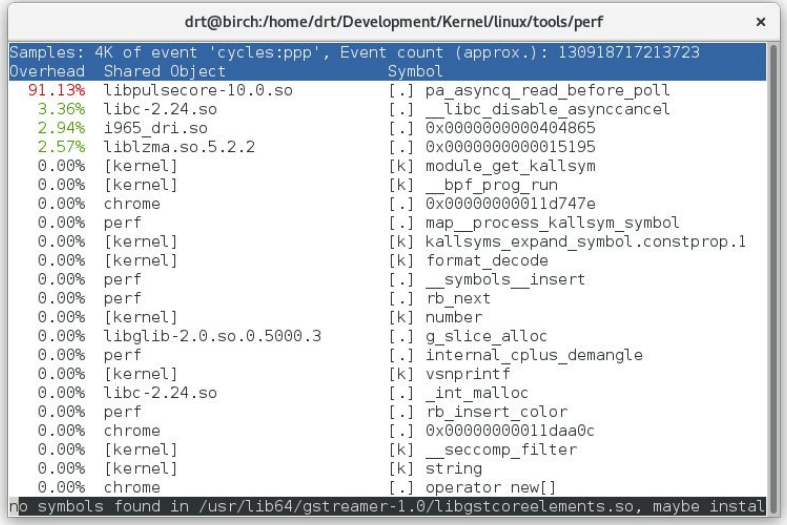
perf

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



My widget missed its deadline

Lock statistics

Lock statistics (`CONFIG_LOCK_STAT`) is a **lock profiler** for Linux. It reuses same hooks as the lockdep hooks to provide per-class lock statistics to help identify heavily contended locks and cross-CPU data sharing.

```
echo 1 > /proc/sys/kernel/lock_stat
# Allow workload to run
echo 0 > /proc/sys/kernel/lock_stat
more /proc/lock_stat
grep : /proc/lock_stat | head
```

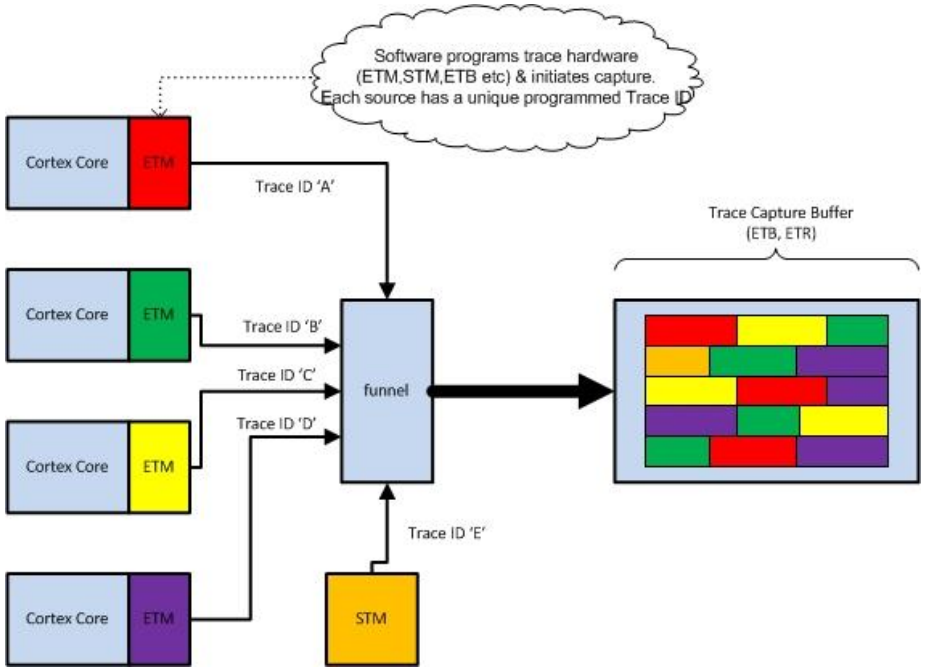
```
lock_stat version 0.4                              # Versioned for automatic tooling
-----------------------------------------------------------------------------------------
class name     con-bounces    contentions    waittime-min   waittime-max   ...
-----------------------------------------------------------------------------------------

unix_table_lock:       110            112            0.21          49.24   ...
---------------
unix_table_lock         45    [<ffffffff8150ad8e>] unix_create1+0x16e/0x1b0
unix_table_lock         47    [<ffffffff8150b111>] unix_release_sock+0x31/0x250
unix_table_lock         15    [<ffffffff8150ca37>] unix_find_other+0x117/0x230
unix_table_lock          5    [<ffffffff8150a09f>] unix_autobind+0x11f/0x1b0
---------------
unix_table_lock         39    [<ffffffff8150b111>] unix_release_sock+0x31/0x250
unix_table_lock         49    [<ffffffff8150ad8e>] unix_create1+0x16e/0x1b0
unix_table_lock         20    [<ffffffff8150ca37>] unix_find_other+0x117/0x230
unix_table_lock          4    [<ffffffff8150a09f>] unix_autobind+0x11f/0x1b0

.........................................................................................

&mm->mmap_sem-W:        26             44            0.26         939.10   ...
&mm->mmap_sem-R:        17             60            1.31      299502.61   …
---------------
&mm->mmap_sem            1    [<ffffffff811502a7>] khugepaged_scan_mm_slot+0x57/0x280
&mm->mmap_sem           12    [<ffffffff815351c4>] __do_page_fault+0x1d4/0x510
```

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.

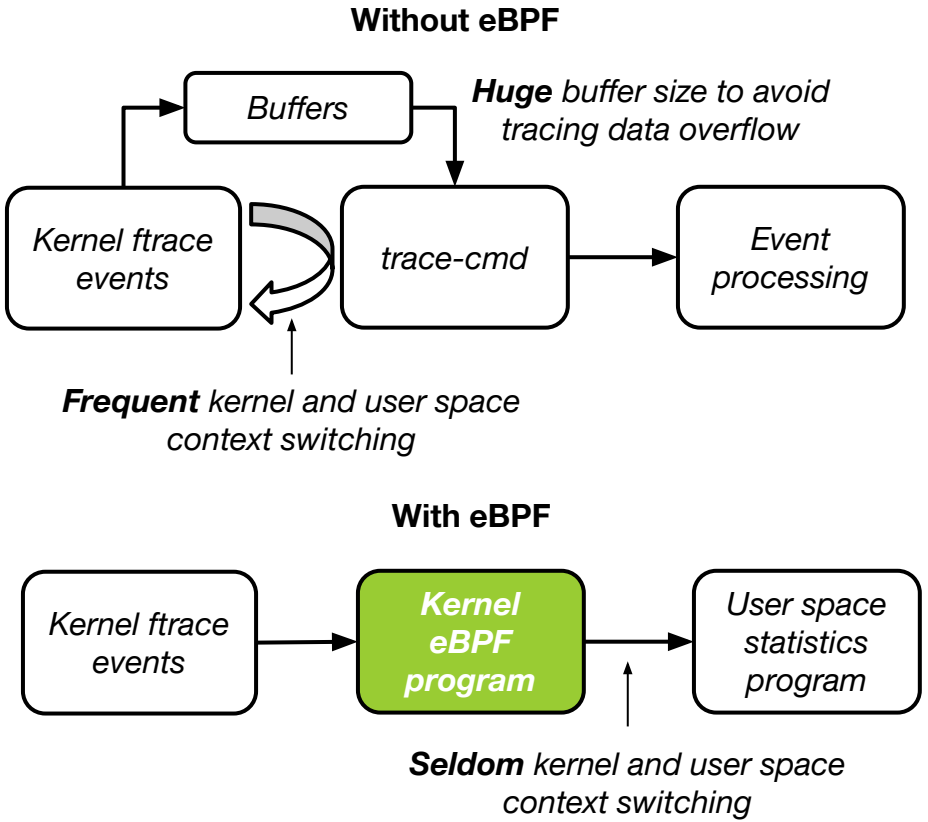
Linaro

eBPF

From the name of eBPF, it's likely to see the meaning for network packet filtering. In fact eBPF is widely used for debugging purpose, e.g. it's extended for Kernel event dynamic filtering (e.g. filter kprobe or ftrace events).

For efficiently processing the interested kernel event, we can write C based program and pass its bytecodes into kernel; in the kernel the eBPF virtual machine is actually an interpreter to convert the eBPF bytecodes to CPU native instructions.

My widget missed its deadline



eBPF tooling

My widget missed its deadline

To efficiently process the kernel event stream, eBPF allows us to write program to summarize trace information without tracing.

eBPF relies on additional tooling for deployment. As well as examples in C, eBPF also offers convenient tooling such as ply/BCC/bpftrace/Systemtap, etc.

Example by using the raw C code to statistic CPU states based on ftrace events

```
CPU states statistics:
     state(ms)  cstate-0    cstate-1    cstate-2    pstate-0    pstate-1    pstate-2    pstate-3    pstate-4
     CPU-0      767         6111        111863      561         31          756         853         190
     CPU-1      241         10606       107956      484         125         646         990         85
     CPU-2      413         19721       98735       636         84          696         757         89
     CPU-3      84          11711       79989       17516       909         4811        5773        341
     CPU-4      152         19610       98229       444         53          649         708         1283
     CPU-5      185         8781        108697      666         91          671         677         1365
     CPU-6      157         21964       95825       581         67          566         684         1284
     CPU-7      125         15238       102704      398         20          665         786         1197
```

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 `earlycon` and `initcall_debug` if the problem occurs during boot
  - If triage analysis identifies `clk_disable_unused()` then set `clk_ignore_unused`
  - Enable (at least) `DEBUG_ATOMIC_SLEEP` and `LOCKUP_DETECTOR` (maybe even re-introduce `DEBUG_SPINLOCK`)
  - 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 is included in OpenOCD (master branch only) 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

Linaro

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
- Post-mortem trace tools hate caches
  - **RAM** often **survives a reset** if the bootloader reconfigure things **fast**
  - Modern L2 (and L3) **caches are** large, and will be **destroyed by a reset**
  - Accessing RAM from other bus initiators may not be cache coherent
  - (We've already talked JTAG debuggers... and they are already **cache coherent**)
- Need to store trace where we can find it later...

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
```

Full 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)
```


Linaro

Aside: How to read an oops

My board just stopped dead

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
```

Aside: How to read an oops - cont.

My board just stopped dead

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:
```

Aside: How to read an oops - cont.

My board just stopped dead

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
```


Linaro

Aside: AArch32 procedure call standard

My board just stopped dead

PC (r15)	**Program counter**
LR (r14)	**Link register (look here if the PC looks borked)**
SP (r13)	**Stack pointer**
r12	Intra-procedure-call scratch register
r11	Frame pointer (or additional temporary)
r9	Platform register (or additional callee-saved)
r4-r8, r10	Temporary registers (callee-saved)
r0..r3	**Parameter/result registers**

Linaro

Aside: AArch64 procedure call standard

My board just stopped dead

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 (caller-saved)
r8	Indirect result location register (for >16 byte results)
r0..r7	**Parameter/result registers**

Aside: AArch64 procedure call standard

My board just stopped dead

pc is good (kernel will automatically looks up symbols)

lr tells us we have already made a function call

x29 is a pointer to the stack (might be a frame record)

x8 is not a pointer (structure returns are pretty rare in the kernel)

x0-x7 could be arguments but let's check the C…

```
static void sysrq_handle_crash(int key)
{
      char *killer = NULL;


      rcu_read_unlock();
      panic_on_oops = 1;  /* force panic */
      wmb();
      *killer = 1;
}
```

```
[  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
```


Linaro

Aside: AArch64 procedure call standard

My board just stopped dead

pc is good (kernel will automatically looks up symbols)

lr tells us we have already made a function call

x29 is a pointer to the stack (might be a frame record)

x8 is not a pointer (structure returns are pretty rare in the kernel)

x0-x7 could be arguments but let's check the C…

```
static void sysrq_handle_crash(int key)
{
	char *ki

	rcu_read
	panic_on
	wmb();
	*killer =
}
```

You can use gdb to look up the symbol offset individually but if working with large stack traces then `scripts/decode_stacktrace.sh` can be used to convert to `__func__ (__FILE__:__LINE__)` form.

```
[  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 =
               0008c610bdb)
               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
```


Linaro

Aside: Convert binary code to instructions

My board just stopped dead

```
wychelm$ aarch64-linux-gnu-gdb build-arm64/vmlinux -q \
> -ex "disass sysrq_handle_crash" -ex quit
Reading symbols from build-arm64/vmlinux...done.
Dump of assembler code for function sysrq_handle_crash:
   0xffff0000085d4140 <+0>:     stp     x29, x30, [sp, #-16]!
   0xffff0000085d4144 <+4>:     mov     x29, sp
   /* rcu_read_unlock(); */
   0xffff0000085d4148 <+8>:     bl      0xffff00000813faf0
                                        <__rcu_read_unlock>
   /* panic_on_oops = 1; */
   0xffff0000085d414c <+12>:    adrp    x1, 0xffff000009280000
                                        <xen_dummy_shared_info+976>
   0xffff0000085d4150 <+16>:    mov     w0, #0x1
   0xffff0000085d4154 <+20>:    str     w0, [x1, #1448]
   /* wmb(); */
   0xffff0000085d4158 <+24>:    dsb     st
   /* *killer = 1; */
   0xffff0000085d415c <+28>:    mov     x1, #0x0
   0xffff0000085d4160 <+32>:    strb    w0, [x1]
   0xffff0000085d4164 <+36>:    ldp     x29, x30, [sp], #16
   0xffff0000085d4168 <+40>:    ret
End of assembler dump.
```

```
[  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: 00000000005f5e0ff
[  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
```


Linaro

Aside: 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-as -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]
```

Coresight

My board just stopped dead

Trace

CoreSight trace doesn't have to be captured on the device itself. 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:  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)
```

The story - I’m sure this used to work

“I ran this test a couple of weeks ago although I haven’t run it recently because I haven’t integrated it into our CI loops yet. I’m sure this used to work fine.”

Scope:

- The scope here is fairly narrow although we don’t know how complex the test it.
- The reporter was performing ad-hoc testing and may not have been very rigorous about configuration management.

Notes:

- The kernel source code is strongly version controlled using git
- Other parameters may be much less well controlled:
  - Kernel configuration
  - Compiler and toolset
  - Userspace

git bisect

I'm sure this used to work

```
git bisect start <bad_sha1> <good_sha1>
git bisect run compile_kernel_and_run_test.sh
```

`compile_kernel_and_test.sh` must return one of four return values:

0:	kernel compiled and test passed
1-124,126,127:	kernel compiled, test failed
125:	kernel failed to compile, test not run
Other:	Abort bisection (catastrophic fail)

Even if it looks difficult to write `compile_kernel_and_test.sh` try anyway!

Manual bisection using git good/bad/skip is error prone

If a manual step is required (power cycle, press button, etc) then get the shell script to prompt you when you need to to perform it but automate everything else.

ktest.pl

I'm sure this used to work

`ktest.pl` is a Perl script included in the kernel sources (tools/testing/ktest). It can be used to automatically run tests on targets board that can be remotely rebooted.

It can be used as a slightly higher level framework for running git bisect, replacing `compile_kernel_and_test.sh`). However the framework can also perform other types of empirical testing, for example it can bisect .config files.

Sadly there is no `Documentation/` but `tools/testing/ktest/sample.conf` and `tools/testing/ktest/examples` are well commented!

Kernel triva: ktest.pl was written by Steven Rostedt, who we also have to thank for ftrace!

The story - My board just randomly failed

“I wasn’t doing anything special, my board just failed for no reason. I guess the system was pretty busy with some of the workloads, but all the workloads run OK some of the time.”

Scope:

- The scope here is (deliberately) very wide
- It’s **not** a compiler bug until you can point at the bad opcode and explain why the compiler was forbidden to emit it
- It’s **not** a hardware bug until you have proved it (or at least got powerful evidence)

Notes:

- There’s a lot of problem solving to be done here but most it isn’t really “kernel debug”
- Kernel engineers are often exposed to weakly validated hardware, especially for commodity interfaces (USB, PCIe, WiFi)
- Effect of SMP race conditions and memory corruption can be hard to tell apart

No tricks... just hard work

My board just randomly failed

Investment ahead of time

Get kernel drivers (or bootloader) to display contents of every reset reason register you can find (WDT, thermal, PMIC)

Ensure DVFS and other power supply changes can be traced and disabled

Write a pstore driver (or configure kdump)

Don’t write SMP bugs…

Automate everything

Debugging when problems appear

Maintain a summary-of-investigation and retain all logs, traces and memory dumps.

Always seek to replicate on a different board (and with a different power supply).

All the usual “fail early” techniques should be deployed.

MEMTEST (kernel boot time memory test) can provide sanity testing but always-on test running in userspace or on a co-processor is often better.

Core dumps (kdump and co-processor dumps) can be useful to prove hardware did not act correctly but only if they fail early enough.

Aside: Memory barrier

My board just randomly failed

ARM architecture has a weak memory model; the memory versus I/O access order is not guaranteed.

Kernel defines default accessors with inserting barriers, which are most safe APIs with endian conversion and memory barriers:

```
writel()/readl()
ioreadX()/iowriteX()
```

Kernel also provide the relaxed accessors:

```
writel_relaxed()/readl_relaxed()
__raw_writel()/__raw_readl()
readsX()/writesX()
ioreadX_rep()/iowriteX_rep()
```

Below code have potential issue is dma channel enabling maybe is written to the device early than dma description, especially these two accessions are not in the same endpoint so the out of order is caused by the early acknowledge;

```
volatile u32 *dma_desc = (u32 *)0xa0000000;
volatile u32 *dma_ena  = (u32 *)0xfe000004;

*dma_desc = DMA_SRC_32BIT | DMA_DST_32BIT;
*dma_ena  = DMA_ENA_BIT;
```

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. So need to use barrier for safe accessing.

Overview

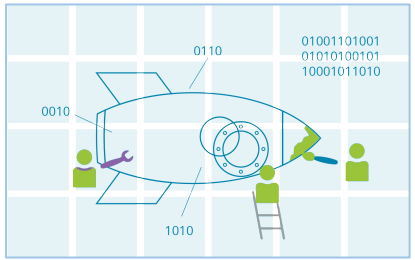
- The Basics
  - Tracing, profiling and stop-the-world
  - Failing early
- The Stories
  - I can't reproduce but by customer can (and I hate flying)
  - My XYZ missed its deadline
  - My board just stopped dead
  - I'm sure this used to work
  - My board just randomly failed
- The Wrap up (and free gift)

Linaro

Linaro Developer Services helps companies build, test, deploy and maintain great products on Arm

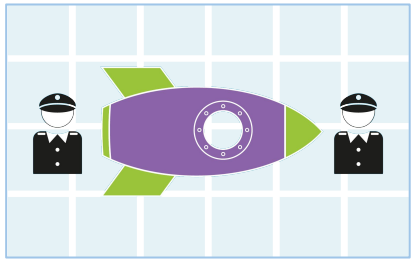
Leverage our Arm expertise

As part of Linaro, Developer Services has some of the world's **leading Arm Software experts**. All of this expertise and experience is made available to you for your projects.



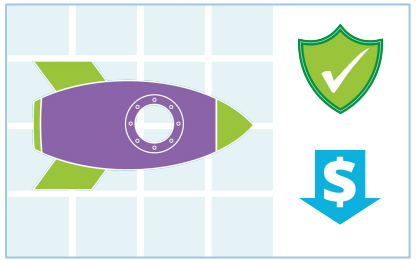
Secure your product

Specialists in security and Trusted Execution Environment (TEE) on Arm, we leverage open source to ensure you **benefit from the latest upstream features and security fixes**.



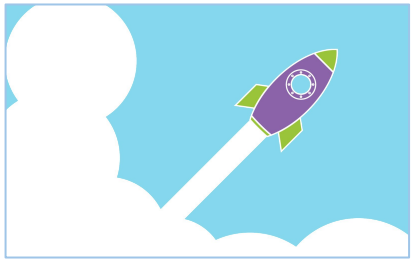
Maintain quality cost-effectively

We offer continuous integration (CI) and automated validation for your product software, ensuring the **highest possible quality**. We upstream code to **reduce the cost and effort needed to maintain your product**.



Build, Test and deploy faster

We support every aspect of product delivery, from building secure board support packages (BSPs), product validation and long-term maintenance - we help get your **products to market faster**.



Linaro Training & Educational Services

Linaro provides **hands-on training for software developers** who **work on Arm**

Expert Instructors who are **real world engineers** deliver hands-on training

Training delivered **online or onsite**

Training structured to your **team needs** with mentoring available

Training catalogue courses include:

- Advanced kernel debugging techniques and tools
- Upstream Kernel Development
- Arm Trusted Firmware, Secure boot and PSCI
- Power Management - Energy Aware Scheduler (EAS) Introduction and tuning
- Introduction to OP-TEE
- Building Custom Systems with OE/Yocto
- Automatic Validation with LAVA

Linaro

Linaro Developer Technical Support (LDTS)

Support for Developers delivered via web and e-mail

Hands-on engineering expertise to help with technical questions, challenges and problems

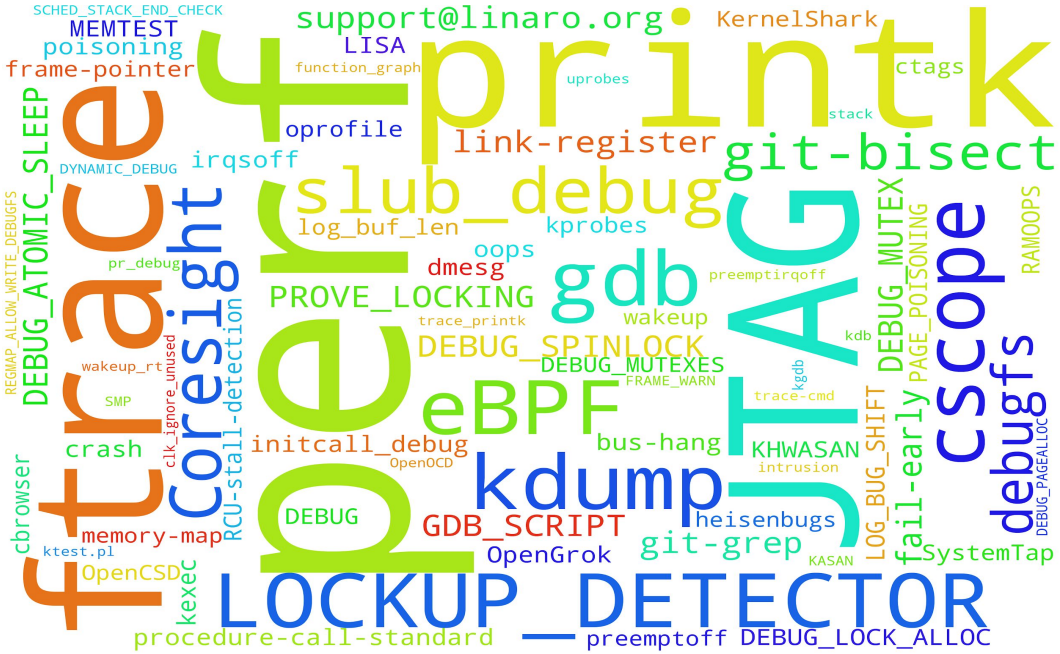
Team maintains renowned **kernel debugging skills**

100% online based
As simple as support@linaro.org

Area of expertise include:

Kernel	LAVA & Testing
BT	Boot
96Boards	KVM
Firmware	Upstreaming & Open Source
Wifi	
OP-TEE	Zephyr RTOS
Toolchain (GNU & LLVM)	Performance Optimization
Arm	Yocto/OE
QEMU	BSP

The Wrap Up



The **Kernel hacking** menu is huge and we just couldn't include everything! We have **hidden** some **secret gems** above that we couldn't cover today. **Download** as a **poster** from: Shared folder

Linaro

Linaro Limited Lifetime Warranty

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

Everyone here today **can ask** questions **about** the **material** we have **presented** now or at any point in the future.

Raise you question by e-mailing support@linaro.org . Be sure to **say what** training course **you attended**.

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

Graphic by Jost, CC0-PD

Lab sessions

Linaro

Briefing

- This week covered a lot of different tools
  - Today the exercises will focus on the tools that we would **not** cover as part of other modules.

- The lab work for Kernel Debug Stories is not prescriptive
  - Don’t rush… take time to look around and to think about if, when and how you could use the feature

- This week all exercises can be undertaken on real hardware and any target rootfs
  - You will need to know how to build and configure a kernel for your hardware and also be able to modify the kernel command line
  - If you prefer to work using the VM then follow the preparation guide.

Preparation: Install host packages

For Debian/Ubuntu:

```
$ sudo apt-get install build-essential flex bison libelf-dev libssl-dev
```

For Fedora and RHEL/CentOS/RockyLinux:

```
$ sudo dnf install gcc flex make bison openssl-devel elfutils-libelf-devel ncurses-devel
```

(or perhaps sudo *yum* install gcc …)

Linaro

Preparation: Downloadable content

The development tools and images used for the lab exercises can be downloaded from:

https://drive.google.com/drive/folders/1uQqtQBYz59XSy7Bzm_v6s7Wqlg_eEeoo

https://people.linaro.org/~leo.yan/training/webinar_kernel_debug_stories/ (If you cannot access Google drive)

You will need to download:

- SDK installer (a very large sh script)
  - The SDK is distribution neutral and provides a reference toolchain together with additional utilities such as the QEMU system emulator!
- The kernel image (`Image-...`) and root filesystem (`core-image-...-ext4.xz`)
- The QEMU configuration file (...qemuconf...)

Decompress the root filesystem:

```
unxz -k -T0 core-image*.rootfs.ext4.xz
```

Preparation: Install the development tools

```
$ sh poky-glibc-x86_64-core-image-minimal-aarch64-qemuarm64-toolchain-3.1.5.sh
Poky (Yocto Project Reference Distro) SDK installer version 3.1.5
==================================================================
Enter target directory for SDK (default: /opt/poky/3.1.5): $HOME/devtools-for-training
You are about to install the SDK to "/home/drt/devtools-for-training". Proceed [Y/n]? y
Extracting SDK.....................................................................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the
environment setup script e.g.
 $ . /home/drt/devtools-for-training/environment-setup-aarch64-poky-linux

$ . /home/drt/devtools-for-training/environment-setup-aarch64-poky-linux
$ aarch64-poky-linux-gcc --version
aarch64-poky-linux-gcc (GCC) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
$
```

Linaro

Boot kernel and rootfs image

1. Source the SDK environment script if needed. Note that the SDK environment script is not persistent and must be sourced every time you open a new terminal window.

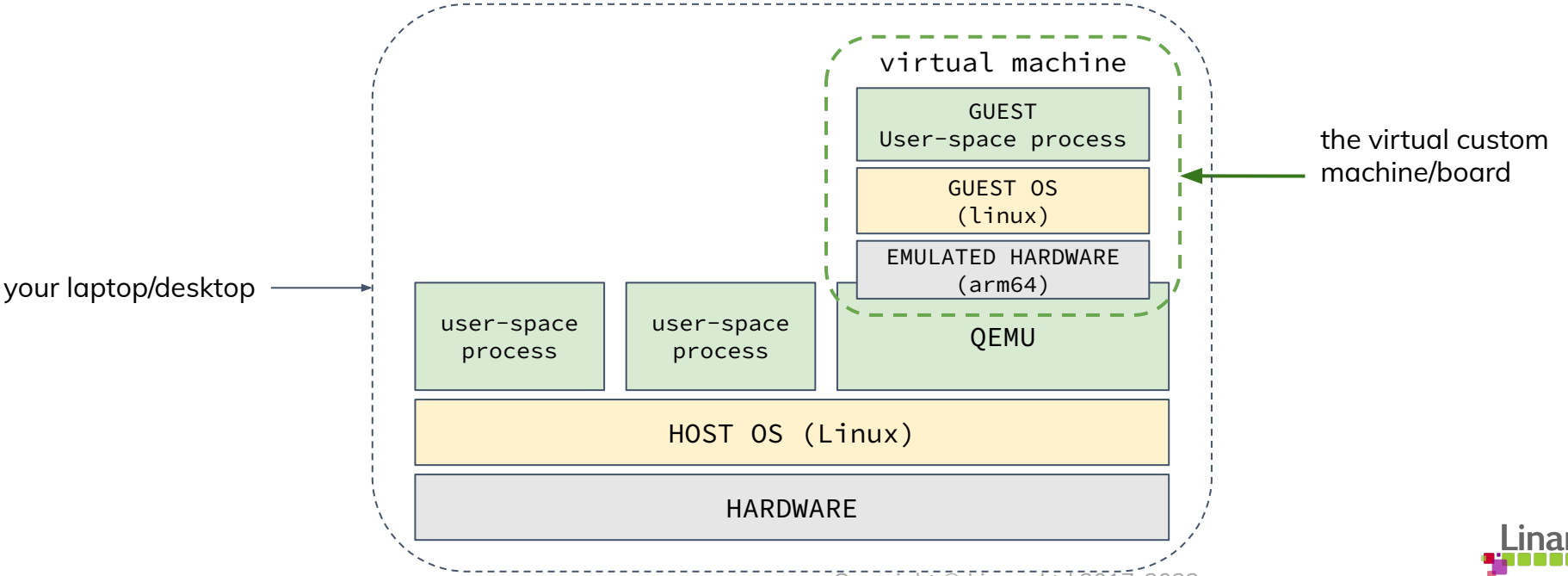
2. Boot (must run from the directory containing the kernel and rootfs):
   `runqemu . slirp nographic`

3. Login as root (there is no password)

4. Use `poweroff` or Ctrl-A X to exit

Aside: QEMU

Qemu is a free and open source emulator providing hardware virtualization. It will allow us to work with an arm64 machine by emulating the processor(s) and its peripherals.



Linaro

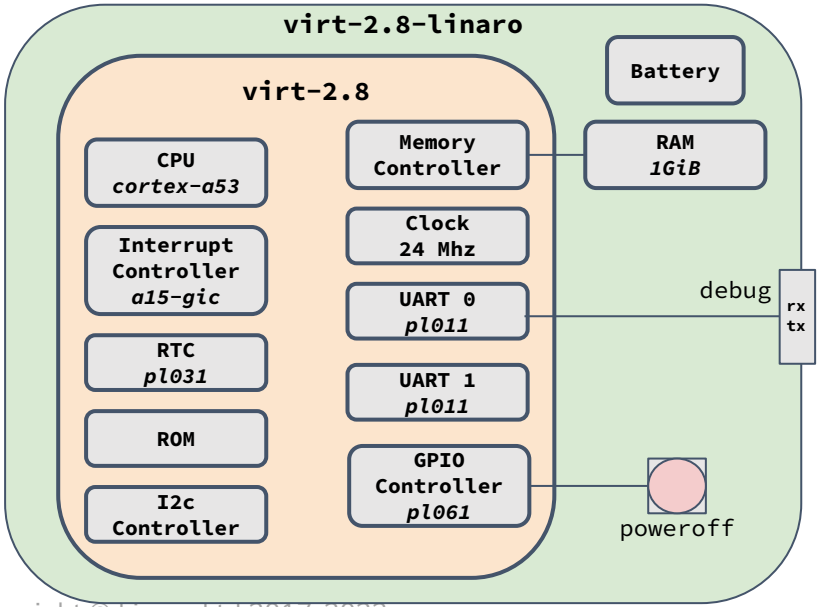
Aside: “Hardware” information

For this course we will use a custom board derived from the virt-2.8 virtual SoC that Qemu provides.

The simplified SoC devices memory mapping is given below:

Device	Address	Size
VIRT_ROM	0x00000000	0x08000000
VIRT_GIC	0x08000000	0x00010000
VIRT_UART (UART 0)	0x09000000	0x00001000
VIRT_RTC	0x09010000	0x00001000
VIRT_I2C	0x0901A000	0x00001000
VIRT_FW_CFG	0x09020000	0x00000018
VIRT_GPIO	0x09030000	0x00001000
VIRT_SECURE_UART (UART 1)	0x09040000	0x00001000
VIRT_MEM	0x40000000	0x08000000

Our custom board is populated with **virt-2.8 SoC**, **1 GiB of RAM**, a **TMP105** temperature sensor, a **poweroff** button and the **UART** is presented on the VM stdin/stdout **as a debug console**.



Aside: Understanding the virtual network

```
runqemu - INFO - Running …/x86_64-pokysdk-linux/usr/bin/qemu-system-aarch64 \
-object rng-random,filename=/dev/urandom,id=rng0 -device virtio-rng-pci,rng=rng0 \
-drive id=disk0,file=…/core-image-akd-qemuarm64.rootfs.ext4,if=none,format=raw \
-device virtio-blk-device,drive=disk0 \
-device qemu-xhci -device usb-tablet -device usb-kbd \
-device virtio-net-device,netdev=net0,mac=52:54:00:12:35:02 \
-netdev user,id=net0,hostfwd=tcp::2222-:22,hostfwd=tcp::2323-:23 \
-machine virt -cpu cortex-a57 -smp 4 -m 256 -serial mon:stdio -serial null \
-nographic -device virtio -gpu-pci \
-kernel …/Image-qemuarm64.bin \
-append 'root=/dev/vda rw  mem=256M ip=dhcp console=tt                sole=hvc0  '
```

Emulate a private network (in **user**space) as part of the qemu process. The emulated network includes your host workstation (10.0.2.2), the VM (10.0.2.15) and a DNS server (10.0.2.3).

The emulated network uses **NAT routing** to bridge to the rest of the network. **hostfwd allows** us to configure **port forwarding**, in this case routing port 2222 on your workstation to port 22 on the VM (port 22 is secure shell). We can use this forwarded port to interact with the target.

Linaro

Preparation: Test the networking and NAT routing

- Boot the VM
  - Use `ip addr` to check the networking is active (the VM is pre-configured to bring up the network interface and launch a DHCP request)
- Open a new Terminal emulator window on your workstation and try:
  - Connecting to the target interactively (lower case -p)

    ```
    ssh -p 2222 root@localhost
    ```
  - Copying files from host to target (upper case -P)

    ```
    scp -P 2222 my_file root@localhost:
    ```
  - Copying files from target to host (upper case -P)

    ```
    scp -P 2222  root@localhost:/proc/config.gz .
    ```
- From the target try:
  - Pinging your workstation

    ```
    ping 10.0.2.2
    ```
  - Test NAT routing (if your workstation network is behind a company firewall replace https://linaro.org with something served internally)

    ```
    wget https://linaro.org
    ```


Linaro

Preparation: Download Linux source code

Work in a different terminal to the one you are using to launch QEMU! As you work through later steps you will need a kernel development terminal and a terminal to launch the VM.

```
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
$ git checkout v5.14
```

To improve download times you may be able to access the kernel from a local mirror within your company or re-use an existing clone you have locally.

To reuse an existing clone without disturbing your existing work try:

```
git worktree add ../linux-akd-training v5.14
```


Linaro

Configure the kernel

If you source the SDK environment scripts then `ARCH` and `CROSS_COMPILE` will be set for you. On most distros you are also able to set these by hand if you prefer (this can be useful for IDE integration):

```
ARCH=arm64
CROSS_COMPILE=$SDKROOT/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-
export ARCH CROSS_COMPILE
```

Change directory into the kernel source directory and configure the kernel:

```
make defconfig
```

Compile the kernel:

```
make -j$(nproc)
```

Kernel and modules are compiled, resulting in a kernel image, **`arch/arm64/boot/Image`** together with kernel modules: **`*.ko`** files spread across the source tree.

Updating the kernel module and boot

Boot the VM using the original command line (e.g. old kernel)

From your kernel development terminal copy the modules to the target

```
make -j `nproc` \
    INSTALL_MOD_PATH=$PWD/overlay INSTALL_MOD_STRIP=1 \
    modules_install
tar --owner root:0 --group root:0 -C overlay -cf - . \
         | ssh -p 2222 root@localhost tar -C / -xvf -
```

The first command will install the modules (with debug information removed) into the `overlay/` directory in the same structure needed on the target. This allows the second command to transfer the modules to the target over the network.

Shutdown the VM and relaunch it with your new self-compiled kernel (change path to KERNEL as needed):

```
KERNEL=linux/arch/arm64/boot/Image runqemu . slirp nographic
```

LAB0 - Recompile the kernel

Linaro

Recompile the kernel

- Enable `DYNAMIC_DEBUG` (**Enable dynamic printk() support**)
  - Found under **Kernel hacking** -> **printk and dmesg options** (`make menuconfig`)
  - Check the online help for other printk related messages
- Rebuild the kernel and the kernel modules and install both to the target device
- This config change is significant and the recompile will take a long time. However the resulting kernel is not required until LAB2 so you can set the kernel building and move on to LAB1.

LAB1 - initcall_debug

Linaro

initcall_debug

- Boot the VM with `earlycon` and `initcall_debug` added to the kernel cmdline:
  `runqemu . slirp nographic bootparams='earlycon initcall_debug'`

- `initcall_debug` enables a lot of debug messages
  - Where did they go and how do you access them without rebooting?
  - Hint: See the printk and dmesg slide

- Add debug (or `loglevel=8`) to the kernel command line and try again:
  `runqemu . slirp nographic bootparams='earlycon initcall_debug debug'`

- If the system crashes and become unresponsive whilst running an init function then `initcall_debug` gives us some idea about what to point our debugger, function tracer or source navigator at in order to learn more

- Q: What does `earlycon` do and why is it often combined it with `initcall_debug`?

LAB2 - DYNAMIC_DEBUG

Linaro

Spot the difference!

- Make sure the kernel built on LAB0 has **finished compiling** (and you have **installed** both the **kernel** and any **modules** to the target)
- Let's experiment with some different values on the kernel command line to configure dynamic debug in different ways.
- Undo the kernel changes from LAB1 (don't launch with `initcall_debug debug`)
- Instead try the following combinations of command line arguments and work out how they differ from each other (rebooting each time):
  - `dyndbg=+p`
  - `dyndbg=+pfm`
  - `dyndbg=+pfm debug`
- Reminder:
  - KERNEL=linux/arch/arm64/boot/Image runqemu . slirp nographic bootparams=

Blocklists versus allowlists

- We can try reducing the quantity of debug output by suppressing messages from the noisiest components
  - +p enables messages, -p disables messages
  - dyndbg is processed left to right and can be semi-colon seperated
  - -append 'root=/dev/vda earlycon **debug dyndbg="+pfm; module <modname> -p**" '
  - Note the change of quote characters for the append
    - To allow spaces into the dyndbg value we must embed double quotes into the kernel command line string; changing quote characters for -apend allows us to avoid hard-to-read escaping
  - Q: How would you silence messages from the `virtio_ring` and `kobject` modules? Try it out!

- It is generally more useful to carefully target messages to be enabled (an allow list) since that avoids irrelevant noise
  - Q: How would you write a command line enables output only for `serial_core` module?

- Try out some of the other techniques mentioned in the kernel documentation: https://www.kernel.org/doc/html/latest/admin-guide/dynamic-debug-howto.html

(____ptrval____)

- Enable the debug output for the kobject module
  ... debug dyndbg="module kobject +pf" '

- Experiment with source navigation (perhaps aided by a search engine) and find out why the pointers are not being displayed
  - Hint: Try booting with `debug_boot_weak_hash` added to the kernel command line

- Note: If you are working with an old kernel you may see “real” pointers instead of (____ptrval____) in the output. If so you may still like to do the web searches to get a glimpse of what is coming!

LAB3 - DebugFS

Linaro

Explore debugfs

- Debugfs is a filesystem
  - Can be explored using standard Unix-like tools: cd, cat, find, grep, echo, tee
  - Try: `find /sys/kernel/debug`

- Pick something “interesting” and use `git grep` and a source navigator to find out what is does!
  - Real hardware is usually much more interesting then QEMU! so if you have access to real hardware try exploring that too!

- Dynamic debug itself has a debugfs interface that can be used to enable/disable dynamic messages without using the kernel command line
  - Locate these interfaces and use them to enable/disable additional log messages
  - Note: *You can also use /proc/sys/kernel/printk to change the system log level to ensure the new messages go the UART as well as the dmesg buffer*

Regmap enabled TMP105 driver for QEMU!

- QEMU! has a fairly “boring” DebugFS directory because the emulated SoC is very simple (fixed clocks, no power regulators, few buses, etc)
- Let’s add a new driver that implements regmap support
- Apply the kernel patch

```
git am 0001-drivers-hwmon-Add-a-simple-TMP105-driver.patch
```

- Reconfigure the kernel
  - Ensure `CONFIG_I2C_VERSATILE` and `CONFIG_SENSORS_TMP105` are enabled
- Rebuild the kernel and boot QEMU using the new kernel

```
KERNEL=linux/arch/arm64/boot/Image runqemu . slirp nographic
```

- Check the driver probed successfully

```
root@qemuarm64:~# dmesg | grep tmp105
[    0.600876] tmp105 0-0048: New tmp105 detected, address = 0x48
```


Linaro

DebugFS and Regmap

- Take a look at `/sys/kernel/debug/regmap`
- Any active driver that uses regmap will appear here
  - Find the driver you are most familiar with and look at its register set
  - If you are working on QEMU! then you can use the TMP105 driver
- Use `watch` to look for changes in register values:

```
watch cat registers
```

- Enable `REGMAP_ALLOW_WRITE_DEBUGFS`
  - It is **not** a Kconfig option
  - Use `git grep` to find it in the kernel sources
  - If you know the hardware well enough, try writing new register values

Challenges

Ideas to explore

- Deeper exploration of dynamic debug
  - Enable/disable debug messages for specific functions (rather than specific modules)
- Add dynamic debug messages to your own drivers (in QEMU!)
  - Use pr_dbg() and dev_dbg() macros and then craft a targeted dyndbg= line to enable your messages
- Use a source navigator to see how dev_dbg() is implemented
  - Elixir is good for online lookup (but doesn't limit your search based on ARCH)
  - Reminder: make cscope is a reasonable default if you don't have a preference
- Expose some debugfs files from one of your own drivers (raw value most recently read from hardware?)
  - API docs are included in the kernel sources (Documentation/filesystems/debugfs.txt)
  - https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt

Thank you