BUD17-TR04: Kernel Debug Stories

Daniel Thompson & Leo Yan
Linaro Solutions and Support 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 55 minutes 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 independent study by trainees.

Linaro connect
Budapest 2017

ENGINEERS AND DEVICES WORKING TOGETHER

Overview

- The Basics
  - Tracing, profiling and stop-the-world
  - Memory layout
  - 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 Advert
  - We could talk about debugging all day (but they only gave us an hour)
- The Wrap Up (and free gift)

Tracing, profiling and stop-the-world

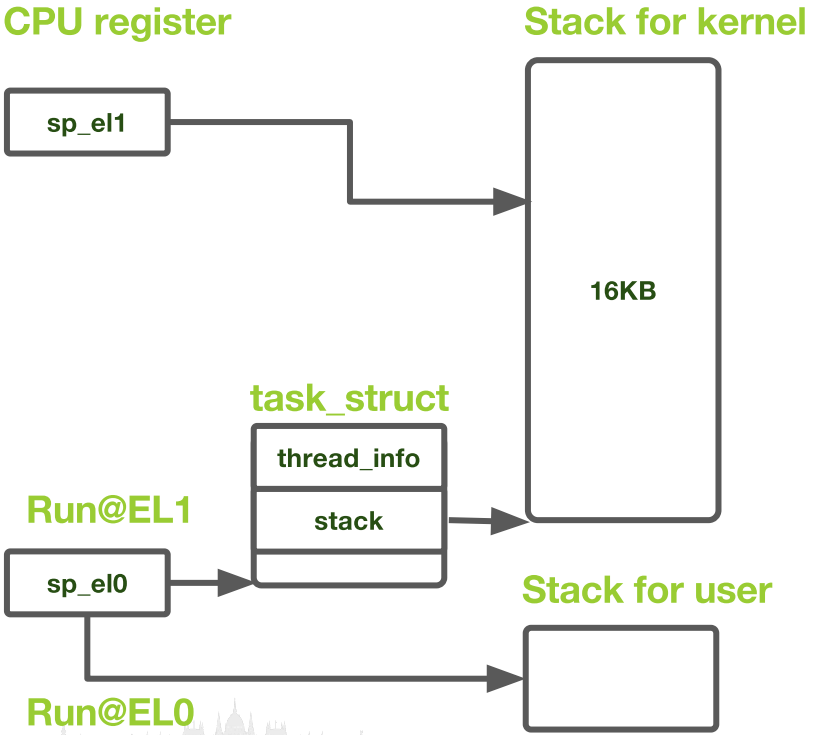
- Tracing
  - Gathering of **events** during system execution
  - Events often have a **timestamp** to assist interpretation
  - `printk()` is a (low performance) form of tracing
- Profiling
  - Gathering of **statistics** during system execution
  - Threads that dominate CPU, L2 cache-miss, etc.
  - 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

ARM64 memory map (4kb page + 4 level)

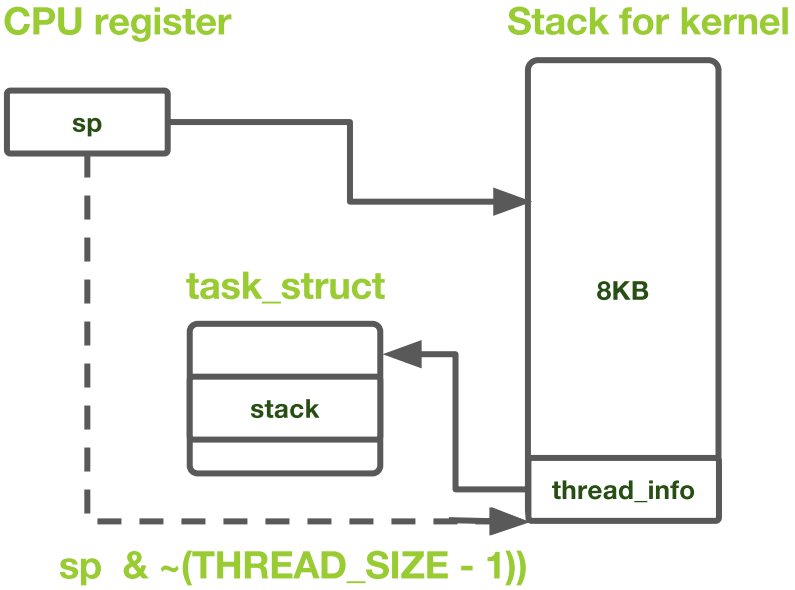
#	Name		Start address	End address	Size	Memory mapping attribution
1	User		0x0	0x0000ffffffffffff	256TB	NORMAL
2	modules		0xffff000000000000	0xffff000008000000	128MB	NORMAL
3	Vmalloc	map_vm_area	0xffff000008000000	0xffff00000807ffff	512KB	DEVICE_nGnRnE: DMA coherent memory region or ioremap NORMAL: vmalloc
		.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

/arch/arm64



/arch/arm



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_SPINLOCK` - *timeout and log if cannot acquire a spinlock*
  - `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)*

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. By this point `DEBUG_SPINLOCK` and `DEBUG_MUTEXES` are probably turned off making the fault 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
connect
Budapest 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Overview

- The Basics
  - Tracing, profiling and stop-the-world
  - Memory layout
  - 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 Advert
  - We could talk about debugging all day (but they only gave us an hour)
- The Wrap up

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… but look here if everything looks borked)
`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

Also known as "how to read an oops"

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
```

ftrace - Function tracing

I can't reproduce but my customer can

- ftrace is fairly lightweight… when it is disabled
- 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** is still being upstreamed, **expected in v4.12**. Full arm64 support in kexec-tools is also not yet 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 and 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 - expected in v4.11*

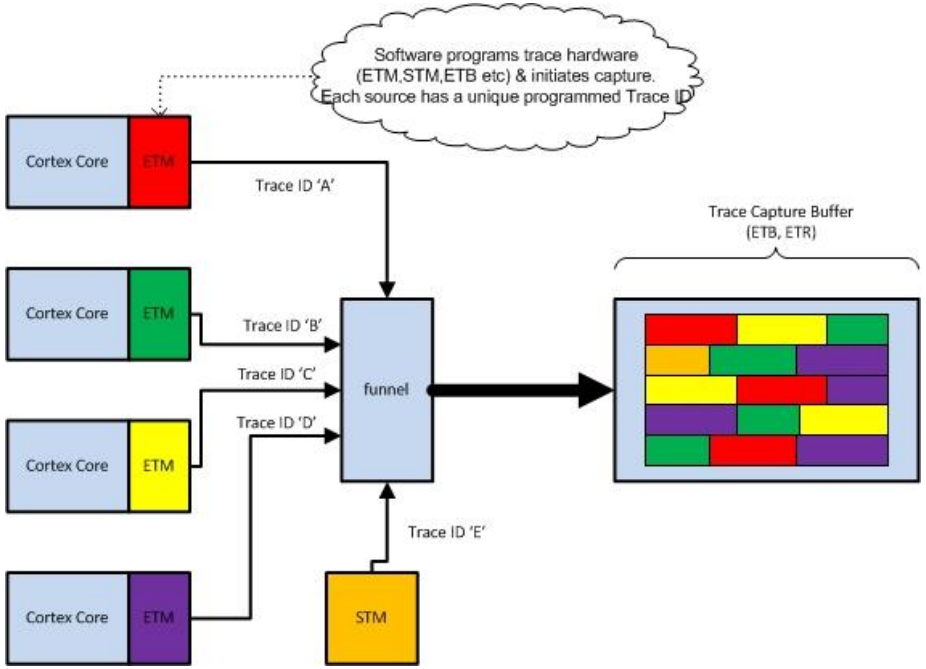
Coresight and OpenCSD

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.



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 - Let’s hack!

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**
- Hacking ftrace to allocate uncached memory is possible but…
- … many ARM cores have terrible performance for uncached memory (and the code is nasty)
- Easier and more performant to hack in cache management calls to the tracing logic

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.

```
[  277.592103] PCSR of cluster0 cpu1 is 0xffff0000080c4f78
[  277.592103] PCSR of cluster0 cpu1 is 0xffff0000080c4f78
[  277.600833] PCSR of cluster0 cpu2 is 0xffff00000809a934
[  277.609564] PCSR of cluster0 cpu3 is 0xffff00000809a934
```

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.

Let's hack: Other processors can extract this too (no cache problems)! Also RFC patches (written by Leo) exist to allow Linux SMP partners to watch each other (enhanced LOCKUP_DETECTOR):
http://archive.arm.linux.org.uk/lurker/message/20170213.061135.c97560eb.en.html

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 some 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 registers you can find (WDT, thermal, PMIC).

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

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.

Linaro
connect
Budapest 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Overview

- The Basics
  - Tracing, profiling and stop-the-world
  - Memory layout
  - 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 Advert
  - We could talk about debugging all day (but they only gave us an hour)
- The Wrap up

The Advert

“We could talk about debugging all day”...

… and we’d love to include the technologies and fine detail we just couldn’t cram into today’s presentation.

As well as focusing on the stories we’d have the time to focus on how to use each tool and the technology behind it. We’d also have time to discuss ways your company's SoC-level debug technologies could complement the Linux features we’ve discussed today.

Linaro connect
Budapest 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

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.

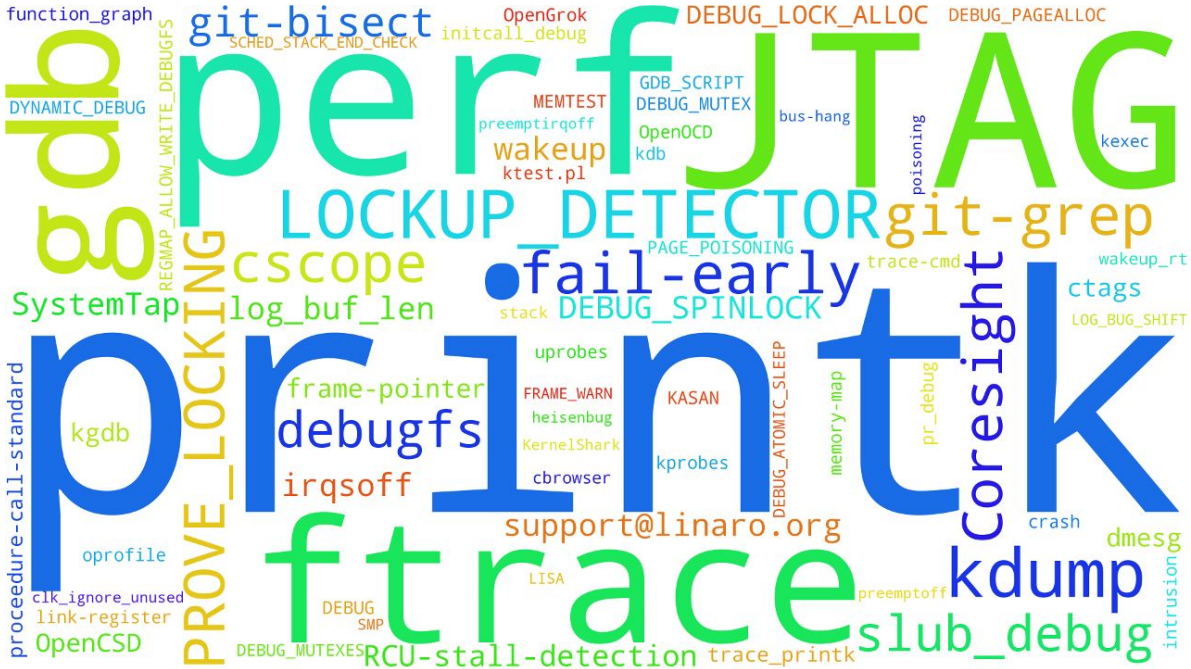
Linaro
connect
Budapest 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Overview

- The Basics
  - Tracing, profiling and stop-the-world
  - Memory layout
  - 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 Advert
  - We could talk about debugging all day (but they only gave us an hour)
- The Wrap up

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: http://linaro.co/bud17-kdb

Thank You

#BUD17

For further information: support@linaro.org or www.linaro.org

BUD17 keynotes and videos on: connect.linaro.org