1. Nothing confidential in any of the slides. Everything is public information, all code is open source.

2. Lots of good information but no claims of any accuracy -- do your own research.
KERNEL DEBUGGING INTRO

- Usually no magic formula, requires creative detective work.

- "You can't depend on your eyes when your imagination is out of focus." - Mark Twain
Talk is not about the creative part of the detective work, but what's available - the choice is yours

"You can't connect the dots looking forward; you can only connect them looking backward. So you have to trust that the dots will somehow connect in your future. “ -- Steve Jobs

Won't be covering intro-level sw debugging but rather diving straight into the dots

Many ways to arrive at same result:
  ○ My first kernel patch 14 years ago came out of analyzing wireshark traces and hypothesizing the issue.
    Not using any specific kernel debug tools
ATTITUDE TO TRY NEW THINGS

- Experiment when you hit a real problem:
  - "It always seems impossible until it's done." -- Nelson Mandela
KERNEL DEBUGGING USING GDB

Why use live GDB?
- Understanding code flow
- Dumping data structures and assembly
- Debugging hangs

Why not use live GDB?
- Issue is not reproducible.
- Don’t know what to look for.
- Cannot run gdb in environment.
  - Note: You can still use gdb on a crash dump though!
KERNEL DEBUGGING USING GDB

I will only explain / demo:

- Qemu + gdb

Same principles, slightly different ways of connection/setup:

- KGDB / KDB
- gdb + OpenOCD
- local gdb + gdbserver on a remote host
KERNEL DEBUGGING USING GDB

Starting qemu gdb server

qemu-system-x86_64 -s -S
  # -S    wait for client at startup (you must type 'c' in the monitor).
  # -s    open a gdbserver on TCP port 1234

Starting gdb client

# In the kernel root

gdb ./vmlinux
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x000000000000fff0 in exception_stacks ()
(gdb) c
Continuing.
KERNEL DEBUGGING USING GDB

WITHOUT CONFIG_DEBUG_INFO

(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
0xffffffff81e8ccbf in default_idle ()
(gdb) bt
#0 0xffffffff81e8ccbf in default_idle ()  
    No line info!
#1 0xffffffff81e8cf8c in default_idle_call ()
#2 0xffffffff810da469 in do_idle ()
#3 0x0000000000000000 in ?? ()
(gdb)
KERNEL DEBUGGING USING GDB

KASLR active (default)

gdb ./vmlinux

(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x000000000000fff0 in exception_stacks ()
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
0xfffffffffa168ccbf in ?? ()
(gdb) bt
#0 0xfffffffffa168ccbf in ?? ()
#1 0xfffffffffa168cf8c in ?? ()
#2 0xfffffffffa08da469 in ?? ()
#3 0x0000000000000000 in ?? ()
KERNEL DEBUGGING USING GDB

WITHOUT KASLR (nokaslr boot param) and CONFIG_DEBUG_INFO

(gdb) c
Continuing.

^C
Thread 1 received signal SIGINT, Interrupt.
default_idle () at arch/x86/kernel/process.c:711
    711 raw_local_irq_disable();

(gdb) bt
#0  default_idle () at arch/x86/kernel/process.c:711
#1  0xffffffff81e8cf8c in default_idle_call ()
    at kernel/sched/idle.c:97
#2  0xffffffff810da469 in cpuidle_idle_call ()
    at kernel/sched/idle.c:170

[...]
KERNEL DEBUGGING USING GDB

Finding the current function on all CPUs

(gdb) info threads
   Id  Target Id              Frame
   *1  Thread 1.1 (CPU#0 [halted]) default_idle () at arch/x86/kernel/process.c:711
   2   Thread 1.2 (CPU#1 [halted]) native_irq_disable () at ./arch/x86/include/asm/irqflags.h:37
   3   Thread 1.3 (CPU#2 [halted]) native_irq_disable () at ./arch/x86/include/asm/irqflags.h:37
   4   Thread 1.4 (CPU#3 [halted]) native_irq_disable () at ./arch/x86/include/asm/irqflags.h:37

And then:
(gdb) thread 2/3/4
Test: Setting a breakpoint on panic

In one window, boot with qemu and run: `echo c > /proc/sysrq-trigger`

GDB window:

(gdb) break panic  # Or hbreak
Thread 1 hit Breakpoint 1, panic (fmt=fmt@entry=0xffffffff826ba290 "sysrq triggered crash\n")
   at kernel/panic.c:277
277     {
(gdb)
KERNEL DEBUGGING USING GDB

Test: Setting a breakpoint on panic

List shows you even the code for panic:

```
(gdb) list
272     *     Display a message, then perform cleanups.
273     *
274     *     This function never returns.
275     */
276     void panic(const char *fmt, ...)
277     {
278         static char buf[1024];
279         va_list args;
280         long i, i_next = 0, len;
281         int state = 0;
282         int old_cpu, this_cpu;
```
Test: Setting a breakpoint on panic

Disas shows you the assembler of the panic function. The “=>” arrow is where the IP is.

(gdb) disas

Dump of assembler code for function panic:

=> 0xffffffff81085ca0 <+0>: endbr64
  0xffffffff81085ca4 <+4>: push %rbp
  0xffffffff81085ca5 <+5>: mov %rsp,%rbp
  0xffffffff81085ca8 <+8>: push %r14
  0xffffffff81085caa <+10>: push %r13
  0xffffffff81085cac <+12>: push %r12
  0xffffffff81085cae <+14>: push %rbx
  0xffffffff81085caf <+15>: mov %rdi,%rbx
  0xffffffff81085cb2 <+18>: sub $0x50,%rsp
### Test: Setting a breakpoint on panic

**Dump all the registers at the breakpoint** *(notice rip is same that disas showed)*

```
(gdb) info registers
rax          0x0                0
rbx          0x0                0
rcx          0xffffffffdfff     4294959103
rsi          0x1                1
rdi          0xffffffff826ba290  -2106875248
rbp          0x4                0x4 <fixed_percpu_data+4>
rsp          0xffffffffc90000317e08 0xffffffffc90000317e08
r8           0xffffffffdfff     4294959103
r12          0x63               99
r13          0x0                0
r14          0xffffffff82281fc0  -2111299648
r15          0x0                0
rip          0xffffffff81085ca0  0xffffffff81085ca0 <panic>
```
KERNEL DEBUGGING USING GDB

Test: Setting a breakpoint on panic

Look at all args passed to panic
(gdb) info args
fmt = 0xffffffff826ba290 "sysrq triggered crash\n"
Test: Setting a breakpoint on panic

# Look at all local vars at the BP
(gdb) info locals
buf = '\000' <repeats 1023 times>
args = {{gp_offset = 2170601861, fp_offset = 4294967295,
    overflow_arg_area = 0xffffffff8160c7d5 <__handle_sysrq+165>,
    reg_save_area = 0x2 <fixed_percpu_data+2>}}
i = <optimized out>
i_next = <optimized out>
len = <optimized out>    # NOTE: As you step through program, these become available.
state = <optimized out>
old_cpu = <optimized out>
this_cpu = <optimized out>
_crash_kexec_post_notifiers = <optimized out>
KERNEL DEBUGGING USING GDB

gdb -tui mode

Demo

- Set breakpoint on panic
- Show single stepping by “next”
- Show enter function preempt_disable_notrace by “step”
- Show step out by “finish”
KERNEL DEBUGGING USING GDB

Real example of recent use of gdb

- RCU code crashing after 1-2 hours. Reproducible.
- No kernel crash, just hang.
- Thought it might actually be a Qemu bug.
- Hang happened in stop machine where all CPUs are hung with interrupts disabled.
- No chance of kernel watchdog to crash the system.

GDB !!!
KERNEL DEBUGGING USING GDB

Real example of recent use of gdb

- Thanks to gdb, I was able to find out that one of the CPUs was getting a “timer interrupt storm” and hanging system.
- At the time of the hang, I did the following gdb commands:
  - info threads (to look at the function on all CPU - stop_machine)
  - thread N (switch to CPU N for backtraces)
  - backtrace (invoke the backtrace)
- See thread:
  https://lore.kernel.org/all/20230810221416.GB562211@google.com/
KERNEL DEBUGGING USING GDB

Real example of recent use of GDB

(gdb) bt
#0 0xffffffff81050ab8 in native_apic_mem_write (reg=896, v=<optimized out>) at ./arch/x86/include/asm/apic.h:110
#1 0xffffffff8104ab9b in lapic_timer_shutdown (evt=<optimized out>) at arch/x86/kernel/apic/apic.c:490
#2 0xffffffff81106576 in __clockevents_switch_state (state=CLOCK_EVT_STATE_ONESHOT_STOPPED, dev=0xffffffff88801f49af80) at kernel/time/clockevents.c:131
#3 clockevents_switch_state (dev=dev@entry=0xffffffff88801f49af80, state=state@entry=CLOCK_EVT_STATE_ONESHOT_STOPPED) at kernel/time/clockevents.c:151
#4 0xffffffff811084cb in tick_program_event (expires=9223372036854775807, force=<optimized out>) at kernel/time/tick-oneshot.c:31
#5 0xffffffff810f7708 in hrtimer_interrupt (dev=<optimized out>) at kernel/time/hrtimer.c:1824
#6 0xffffffff8104b115 in local_apic_timer_interrupt () at arch/x86/kernel/apic/apic.c:1085
#7 __sysvec_apic_timer_interrupt (regs=<optimized out>) at arch/x86/kernel/apic/apic.c:1102
#8 0xffffffff81c5e721 in sysvec_apic_timer_interrupt (regs=0xffffc90000117dc8) at arch/x86/kernel/apic/apic.c:1096
Backtrace stopped: Cannot access memory at address 0xffffc9000013d008
KERNEL DEBUGGING: STACK QUALITY

Enable CONFIG_FRAME_POINTERS for the best stack:

```c
[ 44.644614] <IRQ>
[ 44.644615] __const_udelay+0x3e/0x50 // Missing without FP
[ 44.644618] ipi_handler+0x17/0x20 [ipist] // Missing without FP
[ 44.644628] __flush_smp_call_function_queue+0xf2/0x420
[ 44.644696] ? tick_nohz_stop_idle+0x4b/0x70
[ 44.644700] generic_smp_call_function_single_interrupt+0x17/0x20
[ 44.644702] __sysvec_call_function_single+0x31/0xd0
[ 44.644707] sysvec_call_function_single+0x73/0xa0
[ 44.644745] </IRQ>
```

Without this, both the kernel and gdb depend on vmlinux’s debug info for a good stack. Kernel embeds ORC (concise DWARF for runtime) and uses it for unwinding.
TRICK: WHAT'S A FUNCTION DOING

trace-cmd record -p function_graph -g kfree --max-graph-depth 3

Examples:

1. trace-cmd-139 [003] 205.462836: funcgraph_entry:                   | kfree()
2. trace-cmd-139 [003] 205.462838: funcgraph_entry:                   |    __kmem_cache_free()
3. trace-cmd-139 [003] 205.462838: funcgraph_entry:        0.856 us   |      fixup_red_left()
4. trace-cmd-139 [003] 205.462840: funcgraph_entry:      + 22.222 us  |      stack_trace_save()
5. trace-cmd-139 [003] 205.462863: funcgraph_entry:        0.781 us   |    }
6. trace-cmd-139 [003] 205.462865: funcgraph_exit:       + 27.273 us  |  }
7. trace-cmd-139 [003] 205.462866: funcgraph_exit:        30.262 us   |}

NOTE: This has a bug though where it sometimes shows unrelated functions, I am discussing with ftrace maintainers
TRICK: SUSPECT FUNCTION IS TOO SLOW!

Use function graph tracer!

Examples:

1. trace-cmd record -p function_graph -l vfs_read

   -l filters function tracing to only those.
TRICK: NAILING HIGH SCHEDULER LATENCIES

Use perf sched! I literally used this 50% of the time working on a core scheduling feature.

Dummy bug added to context switch path
(for demo: pass --boot-args 'rcutree.enable_cs_bug=1')

```c
int inc_this;
void rcu_note_context_switch(..) {
    ...
    // Every 10000 context switches, spin for 5ms
    if (nr_cs++ % 1000 == 0) {
        int i;
        for (i = 0; i < 110000000UL; i++) {
            WRITE_ONCE(inc_this, i);
            cpu_relax();
        }
    }
    ...
}
```
TRICK: NAILING HIGH SCHEDULER LATENCIES

Use perf sched! I literally used this 50% of the time to upstreaming a scheduling feature.

perf sched record -- timeout 10 find /
perf sched latency --sort max
TRICK: NAILING HIGH SCHEDULER LATENCIES

Use perf sched! I literally used this 50% of the time to upstreaming a scheduling feature.

```
perf sched record -- timeout 10 find /
perf sched latency --sort max
```

```
[root@qemubox /]$ perf sched latency --sort max

<table>
<thead>
<tr>
<th>Task</th>
<th>Runtime ms</th>
<th>Switches</th>
<th>Avg delay ms</th>
<th>Max delay ms</th>
<th>Max delay start</th>
<th>Max delay end</th>
</tr>
</thead>
<tbody>
<tr>
<td>find:115</td>
<td>5484.467 ms</td>
<td>5</td>
<td>avg: 0.151 ms</td>
<td>max: 0.411 ms</td>
<td>max start: 17.235347 s</td>
<td>max end: 17.235758 s</td>
</tr>
<tr>
<td>rcu_preempt:16</td>
<td>3888.199 ms</td>
<td>2</td>
<td>avg: 0.060 ms</td>
<td>max: 0.119 ms</td>
<td>max start: 22.376060 s</td>
<td>max end: 22.376179 s</td>
</tr>
<tr>
<td>perf:(2)</td>
<td>9.920 ms</td>
<td>2</td>
<td>avg: 0.009 ms</td>
<td>max: 0.017 ms</td>
<td>max start: 21.702475 s</td>
<td>max end: 21.702493 s</td>
</tr>
<tr>
<td>timeout:113</td>
<td>10.210 ms</td>
<td>1</td>
<td>avg: 0.000 ms</td>
<td>max: 0.000 ms</td>
<td>max start: 0.000000 s</td>
<td>max end: 0.000000 s</td>
</tr>
<tr>
<td>migration/2:20</td>
<td>0.000 ms</td>
<td>1</td>
<td>avg: 0.000 ms</td>
<td>max: 0.000 ms</td>
<td>max start: 0.000000 s</td>
<td>max end: 0.000000 s</td>
</tr>
</tbody>
</table>
```
TRICK: NAILING HIGH SCHEDULER LATENCIES

Use perf sched! I literally used this 50% of the time to upstreaming a scheduling feature.

To see the raw perf events in the trace: perf script

```bash
kworker/1:1H-kb  97 [001]  17.226351: sched:sched_stat_runtime: comm=kworker/1:1H pid=97 runtime=258352 [ns]
kworker/1:1H-kb  97 [001]  17.226354: sched:sched_switch: prev_comm=kworker/1:1H prev_pid=97 prev_prio=100
swapper       0 [003]  17.226375: sched:sched_switch: prev_comm=swapper/3 prev_pid=0 prev_prio=120
rcu_preempt    16 [003]  17.226381: sched:sched_stat_runtime: comm=rcu_preempt pid=16 runtime=8736 [ns]
rcu_preempt    16 [003]  17.226382: sched:sched_switch: prev_comm=rcu_preempt prev_pid=16 prev_prio=120
swapper       0 [002]  17.226658: sched:sched_waking: comm=perf-exec pid=113 prio=120 target_cpu=001
swapper       0 [002]  17.226778: sched:sched_waking: comm=kworker/2:1H pid=49 prio=100 target_cpu=002
swapper       0 [002]  17.226781: sched:sched_waking: comm=ksoftirqd/2 pid=21 prio=120 target_cpu=002
swapper       0 [002]  17.226784: sched:sched_switch: prev_comm=swapper/2 prev_pid=0 prev_prio=0
swapper       0 [001]  17.226864: sched:sched_switch: prev_comm=swapper/1 prev_pid=0 prev_prio=0
```

TRICK: NAILING HIGH SCHEDULER LATENCIES

You can also run perf and trace-cmd at the same time

```bash
perf sched record -- timeout 10 find /
trace-cmd record -e power:cpu_idle
```

- Both perf.data and trace.dat can be analyzed later.
- I find this useful because I can use perf sched to find when the latency starts and ends, and then look at trace-cmd for a more full trace.
**TRICK: NAILING HIGH SCHEDULER LATENCIES**

Or just use `perf` and enable the extra trace events

```
perf sched record -e power:cpu_idle -- timeout 10 find /
```

Demo: “`perf script`” will now show `power:cpu_idle` event along with scheduler events.
KERNEL DEBUGGING USING TRACING

Shotgun debugging to understand code flow.

The idea is sprinkle printk() or trace_printk() all over the code being debugged, by copying and pasting.

Example:

    printk("%s: (%s) (%d)", __func__, __FILE__, __LINE__);
// or
    trace_printk("We are here: (%s) (%d)", __FILE__, __LINE__);
KERNEL DEBUGGING USING TRACING

Shotgun debugging to understand code flow (courtesy: Steven Rostedt)

The idea is sprinkle printk() or trace_printk() all over the code being debugged, by copying and pasting.

Example:

    printk("%s: (%s) (%d)", __func__, __FILE__, __LINE__);
    // or
    trace_printk("We are here: (%s) (%d)", __FILE__, __LINE__);

Let's demo, first apply the shotgun.diff from the demo tree.
KERNEL DEBUGGING USING TRACING

Dumping the stack to understand why something’s happening.

- Real debug in December 2023. I noticed that a thread a sleeping in D-state constantly but I didn’t know why / who’s calling the sleep.
- Look at the stack!

Demo:

- Apply tracedumpstack.diff from the demo tree.
- Boot and cat /sys/kernel/debug/tracing/trace.
KERNEL DEBUGGING: HANGS

Example: An interrupt storm

- Test kernel module sends an IPI-storm from one CPU to another.
- Heavy interrupt activity can hang a system such as network irqs / hardware bugs.

Demo:

1. Show the code (one CPU sends 1000 IPIs every 5ms to another).
2. Run qemu and load ipst.ko into the kernel, hangs in a few seconds.
3. What’s going on the CPUs?
   a. Use gdb to look at stacks (info threads + thread N + bt)
   b. Dump dmesg using gdb (make scripts_gdb and then lx-dmesg)
      i. RCU stalls in dmesg clearly show that the CPU receiving the IPI is hung.
      ii. Gdb’s lx-dmesg shows what the console cannot!
KERNEL DEBUGGING: HANGS

Example: An interrupt storm

- Test kernel module sends an IPI-storm from one CPU to another.
- Heavy interrupt activity can hang a system such as network irqs / hardware bugs.

Demo:
1. Rcu stall detector may not always work, let us use the CPU lockup detectors.
KERNEL DEBUGGING: HANGS

Example: An interrupt storm

- Test kernel module sends an IPI-storm from one CPU to another.
- Heavy interrupt activity can hang a system such as network irqs / hardware bugs.

Demo:

1. Enable the configs:
   CONFIG_LOCKUP_DETECTOR=y
   CONFIG_SOFTLOCKUP_DETECTOR=y
   CONFIG_HARDLOCKUP_DETECTOR=y
   CONFIG_HARDLOCKUP_DETECTOR_BUDDY=y
KERNEL DEBUGGING: HANGS

Example: An interrupt storm

- Test kernel module sends an IPI-storm from one CPU to another.
- Heavy interrupt activity can hang a system such as network irqs / hardware bugs.

Demo:

2. Boot and set the following:
   a. Reduce watchdog threshold (determines how often the watchdog checks for hrtimer interrupts progressing. Set to 2 seconds for demo.
      i. echo 2 > /proc/sys/kernel/watchdog_thresh
      ii. echo 1 > /proc/sys/kernel/nmi_watchdog
KERNEL DEBUGGING USING TRACING

Dumping the trace buffer to the console on OOPs

What’s an OOPs?

1. A detailed error report of something bad happened in the kernel.
   Examples: NULL pointer deref, invalid memory access.
2. What’s not an OOPs? Examples: One-off WARN_ON( ), RCU stalls warnings.
3. Kernel may continue operating even after OOPs.

What’s a Panic?

1. Kernel can no longer recover from the error, has to be halted or rebooted.
KERNEL DEBUGGING USING TRACING

Dumping the trace buffer to the console on OOPs

Boot parameters

1. ftrace_dump_on_oops: Dumps to the console on both OOPs and a panic.
   a. Note: For OOPs, trace dump to console doesn’t shut down the kernel. Because OOPs != PANIC.
2. trace_event
3. trace_buf_size
KERNEL DEBUGGING USING TRACING

Forcing a panic on various “problems”

Various boot parameters can force a panic:

1. `sysctl.kernel.panic_on_oops=1` # OOPs causes a panic
2. `sysctl.kernel.panic_on_warn=1` # Warnings cause a panic
3. `sysctl.kernel.panic_on_rcu_stall=1` # RCU stalls cause a panic.
4. `sysctl.kernel.hardlockup_panic=1` # Panic on hard lockup detection.

Note: The advantage of panicking on problems is you can combine:

1. `panic_on_YYYY`
2. `ftrace_dump_on_oops`

to dump ftrace on problem YYYY.
KERNEL DEBUGGING USING TRACING

Full example: Dump the trace buffer to the console on any warning

Step 1: Qemu bootargs
rq --boot-args 'ftrace_dump_on_oops trace_event=sched:sched_switch, power:cpu_idle
trace_buf_size=1K sysctl.kernel.panic_on_warn=1'

Step 2: Load the wp module and observe console dump
KERNEL DEBUGGING USING TRACING

Full example: Dump the trace buffer to the console on an RCU stall

Step 1: Qemu bootargs
rq --boot-args 'ftrace_dump_on_oops trace_event=sched:sched_switch,power:cpu_idle trace_buf_size=1K sysctl.kernel.panic_on_rcu_stall=1'

Step 2: Load the previous IPI storm module and observe console dump
KERNEL DEBUGGING USING TRACING

Tip: turn off tracing once warnings hit

- Trace may have many useless events after the warning fires.
- Pass `sysctl.kernel.traceoff_on_warning` boot parameter to turn off or pass `tracing_off()`.

Run inside tmux (so I can scroll) enabling lots of event:

1. `rq -q --boot-args 'ftrace_dump_on_oops trace_event=sched:*,irq:* trace_buf_size=1K sysctl.kernel.panic_on_warn=1 sysctl.kernel.traceoff_on_warning=1'`
2. Load the wp.ko module to throw warning.

Last line of console dump shows:

```
[ 20.423015] insmod-116       2...1. 16120678us :
    disable_trace_on_warning: Disabling tracing due to warning
```
KERNEL DEBUGGING USING TRACING

Example of recent debug run with all the boot options:

- https://git.kernel.org/pub/scm/linux/kernel/git/jfern/linux.git/commit/?h=rcu/linux-6.5.y-debug-boost&id=5068c9218a58f5fa85129c5de6b75fc213390b6e
KERNEL DEBUGGING: KASAN

- C-language memory safety issues is a source of bugs.
- Detect memory corruption bugs: UAF, OOB etc.
- Example error reports: https://www.kernel.org/doc/...kasan.html#error-reports
- 2-3x slow down; if you are not convinced just check “perf top” ;-)}
KERNEL DEBUGGING USING TRACING

vscode.

(gdb) bt
OTHER IN-KERNEL DEBUGGING TOOLS

- Lockdep
- PreemptIrqsOff tracer
- KASAN
- KCSAN
- NMI watchdog detector
- Hung task detector
BONUS: USING VS CODE AND CLANGD TO IMPROVE DEVELOPMENT EFFICIENCY.

- Peace of mind when developing for the kernel can avoid bugs.
- Amazing git integration, terminal integration.
Thanks.