Linux kernel debugging: going beyond printk messages
ABOUT THIS DOCUMENT

✗ This document is available under Creative Commons BY-SA 4.0.
https://creativecommons.org/licenses/by-sa/4.0/

✗ The source code of this document is available at:
https://e-labworks.com/talks/elce2019
$ WHOAMI

- Embedded software developer for more than 20 years.
- Principal Engineer of Embedded Labworks, a company specialized in the development of software projects and BSPs for embedded systems.
- Active in the embedded systems community in Brazil, creator of the website Embarcados and blogger (Portuguese language).
  https://sergioprado.org
- Contributor of several open source projects, including Buildroot, Yocto Project and the Linux kernel.
THIS TALK IS NOT ABOUT...

× printk and all related functions and features (pr_ and dev_ family of functions, dynamic debug, etc).

× Static analysis tools and fuzzing (sparse, smatch, coccinelle, coverity, trinity, syzkaller, syzbot, etc).

× User space debugging.

× This is also not a tutorial! We will talk about a lot of tools and techniques and have fun with some demos!
DEBUGGING STEP-BY-STEP

1. Understand the problem.

2. Reproduce the problem.

3. Identify the source of the problem.

4. Fix the problem.

5. Fixed? If so, celebrate! If not, go back to step 1.
TYPES OF PROBLEMS

- We can consider as the top 5 types of problems in software:
  - Crash.
  - Lockup.
  - Logic/implementation error.
  - Resource leak.
  - Performance.
TOOLS AND TECHNIQUES

To address these issues, there are some techniques and tools we could use:

- Our brain (aka knowledge).
- Logs and dump analysis (post mortem analysis).
- Tracing/profiling.
- Interactive debugging.
- Debugging frameworks.
## PROBLEMS vs TECHNIQUES

<table>
<thead>
<tr>
<th></th>
<th>Crash</th>
<th>Leak</th>
<th>Sound</th>
<th>Leak</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>printk()</td>
<td>☑</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>
# PROBLEMS vs TECHNIQUES

<table>
<thead>
<tr>
<th></th>
<th>Crash</th>
<th>Lockup</th>
<th>Logic</th>
<th>Leak</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge</td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Smiley" /></td>
</tr>
<tr>
<td>Logs</td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Neutral" /></td>
<td><img src="image" alt="Neutral" /></td>
<td><img src="image" alt="Sad" /></td>
<td><img src="image" alt="Sad" /></td>
</tr>
<tr>
<td>Tracing</td>
<td><img src="image" alt="Neutral" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Neutral" /></td>
<td><img src="image" alt="Neutral" /></td>
<td><img src="image" alt="Smiley" /></td>
</tr>
<tr>
<td>Interactive debugging</td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Neutral" /></td>
<td><img src="image" alt="Sad" /></td>
</tr>
<tr>
<td>Debugging frameworks</td>
<td><img src="image" alt="Sad" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Sad" /></td>
<td><img src="image" alt="Smiley" /></td>
<td><img src="image" alt="Neutral" /></td>
</tr>
</tbody>
</table>
Embedded Linux Conference Europe 2019

Kernel oops analysis
KERNEL OOPS

- **Kernel oops** is a way for the Linux kernel to communicate the user that a certain error has occurred.

- When the kernel detects a problem, it kills any offending processes and prints an oops message in the log, including the current system status and a **stack trace**.

- Different kind of errors could generate a kernel oops, including an illegal memory access or the execution of invalid instructions.

- The official Linux kernel documentation about handling oops messages is available at Documentation/admin-guide/bug-hunting.rst.
KERNEL PANIC

- After a system has experienced an oops, some internal resources may no longer be operational.

- A kernel oops often leads to a kernel panic when the system attempts to use resources that have been lost.

- In a kernel panic, the execution of the kernel is interrupted and a message with the reason of the kernel panic is displayed in the kernel logs.
KERNEL OOPS

```
# cat /sys/class/gpio/gpio504/value
[  23.688107] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[  23.696431] pgd = (ptrval)
[  23.699167] [00000000] *pgd=28bd4831, *pte=00000000, *ppte=00000000
[  23.705596] Internal error: Oops: 17 [#1] SMP ARM
[  23.710316] Modules linked in:
[  23.719060] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[  23.725606] PC is at mcp23sxx_spi_read+0x34/0x84
[  23.730241] LR is at _regmap_raw_read+0xFC/0x384
[  23.741142] sp : d8c6da48 ip : 00000009 fp : d8c6da6c
[  23.746375] r10: 00000040 r9 : d8a94000 r8 : d8c6db30
[  23.751608] r7 : c12ed9d4 r6 : 00000001 r5 : c0539c10 r4 : c1208988
[  23.758145] r3 : d8789f41 r2 : 2afbo7c1 r1 : d8789f40 r0 : 00000000
[...]
[  24.164250] Backtrace:
[  24.166720] [<c0539c10>] (mcp23sxx_spi_read) from [<c067d894>] (_regmap_raw_read+0xFc/0x384)
[  24.177714] [<c067d798>] (_regmap_raw_read) from [<c067db64>] (_regmap_bus_read+0x48/0x70)
[  24.196372] [<c067db1c>] (_regmap_bus_read) from [<c067c1a4>] (_regmap_read+0x74/0x200)
[  24.210056] [<c067c130>] (_regmap_read) from [<c067c37c>] (regmap_read+0x4c/0x6c)
[  24.227931] [<c067c330>] (regmap_read) from [<c053a24c>] (mcp23s08_get+0x58/0xa4)
[  24.241096] [<c053a1f4>] (mcp23s08_get) from [<c053e764>]
[  24.255650] [<c053e724>] (gpiod_get_raw_value_commit) from [<c05401f0>] (gpiod_get_value_canslee
[  24.276913] [<c05401c0>] (gpiod_get_value_cansleep) from [<c054a68>] (value_show+0x34/0x5c)
[  24.288949] [<c0544a34>] (value_show) from [<c06580d0>] (dev_attr_show+0x2c/0x5c)
[  24.302118] [<c06580a4>] (dev_attr_show) from [<c0343a78>] (sysfs_kf_read+0x58/0xd8)
[...]
```
ADDR2LINE

- The addr2line tool is capable of converting a memory address into a line of source code:

$ arm-linux-addr2line -f -e vmlinux 0xc0539c44
mcp23sxx_spi_read
/home/sprado/elce/linux/drivers/pinctrl/pinctrl-mcp23s08.c:357
The `faddr2line` kernel script will translate a stack dump function offset into a source code line:

```
$ ./scripts/faddr2line vmlinux mcp23sxx_spi_read+0x34
mcp23sxx_spi_read+0x34/0x80: mcp23sxx_spi_read at drivers/pinctrl/pinctrl-mcp23s08.c:357
```
$ arm-linux-gdb vmlinux

(gdb) list *(mcp23sxx_spi_read+0x34)
0xc0539c44 is in mcp23sxx_spi_read (drivers/pinctrl/pinctrl-mcp23s08.c:357).
352   u8 tx[2];
353
354   if (reg_size != 1)
355       return -EINVAL;
356
357   tx[0] = mcp->addr | 0x01;
358   tx[1] = *((u8 *) reg);
359
360   spi = to_spi_device(mcp->dev);
$ arm-linux-gdb vmlinux

(gdb) disassemble /m mcp23sxx_spi_read
Dump of assembler code for function mcp23sxx_spi_read:
349     {
    0xc0539c10 <+0>:     mov     r12, sp
    0xc0539c14 <+4>:     push    {r4, r11, r12, lr, pc}
    0xc0539c18 <+8>:     sub     r11, r12, #4
    0xc0539c1c <+12>:    sub     sp, sp, #20
    0xc0539c20 <+16>:    push    {lr}            ; (str lr, [sp, #-4]!)
                           ; (str lr, [sp, #-4]!)

[...]

357     tx[0] = mcp->addr | 0x01;
    0xc0539c3c <+44>:    mov     r0, #0
    0xc0539c44 <+52>:    ldrb    r1, [r0]
    0xc0539c54 <+68>:    orr     r1, r1, #1
    0xc0539c58 <+72>:    strb    r1, [r11, #-26] ; 0xfffffffffe6

[...]
PSTORE

- Pstore is a generic kernel framework for persistent data storage and can be enabled with the CONFIG_PSTORE option.

- With pstore you can save the oops and panic logs through the CONFIG_PSTORE_RAM option, allowing you to retrieve log messages even after a soft reboot.

- By default, logs are stored in a reserved region of RAM, but other storage devices can be used, such as flash memory.
CONFIGURING PSTORE

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;

    ramosps: ramosps@0b00000000 {
        compatible = "ramoops";
        reg = <0x20000000 0x200000>; /* 2MB */
        record-size = <0x4000>; /* 16kB */
        console-size = <0x4000>; /* 16kB */
    }
};
```
**USING PSTORE**

- To access the logs you should mount the `pstore` file system:
  ```bash
  # mount -t pstore pstore /sys/fs/pstore/
  ```

- Saved logs can be accessed through files exported by `pstore`:
  ```bash
  # ls /sys/fs/pstore/
  dmesg-ramoops-0  dmesg-ramoops-1
  ```

- The documentation of this feature is available in the kernel source code at `Documentation/admin-guide/ramoops.rst`. 
KDUMP

- Kdump uses kexec to quickly boot to a dump-capture kernel whenever a dump of the system kernel's memory needs to be taken (for example, when the system panics).

- When the system kernel boots, we need to reserve a small section of memory for the dump-capture kernel, passing a parameter via kernel command line.

  crashkernel=64M

- Using the kexec -p command from kexec-tools we can load the dump-capture kernel into this reserved memory.
On a kernel panic, the new kernel will boot and you can access the memory image of the crashed kernel through `/proc/vmcore`.

This exports the dump as an ELF-format file that can be copied and analysed with tools such as GDB and crash.

More information is available in the Linux kernel source code at `Documentation/kdUMP/kdUMP.txt`. 
Interactive debugging
KERNEL DEBUGGING WITH GDB

✗ Problem 1: How to use the kernel to debug itself?

✗ Problem 2: source code and development tools are on the host and the kernel image is running on target.

✗ Solution: client/server architecture. The Linux kernel has a GDB server implementation called KGDB that communicates with a GDB client over network or serial port connection.
KERNEL DEBUGGING WITH GDB

Host

- arm-linux-gdb
- Kernel image with debug symbols (vmlinux)

serial or ethernet connection

Target

- KGDB
- Linux kernel (zImage)
KGDB


- Supports serial port communication (available in the mainline kernel) and network communication (patch required).

- Available in the mainline Linux kernel since version 2.6.26 (x86 and sparc) and 2.6.27 (arm, mips and ppc).

- Enables full control over kernel execution on target, including memory read and write, step-by-step execution and even breakpoints in interrupt handlers!
KERNEL DEBUGGING WITH GDB

There are three steps to debug the Linux kernel with GDB:

1. Compile the kernel with KGDB support.
2. Configure the Linux kernel on the target to run in debug mode.
3. Use the GDB client to connect to the target via serial or network.
1. ENABLING KGDB

To use KGDB, you must recompile the Linux kernel with the following options:

- CONFIG_KGDB: enables support for KGDB.
- CONFIG_KGDB_SERIAL_CONSOLE: Enables KGDB communication I/O driver over the serial port.
- CONFIG_MAGIC_SYSRQ: Enables magic sysrq key functionality to put the kernel in debug mode.
- CONFIG_DEBUG_INFO: Compiles the kernel with debug symbols.
- CONFIG_FRAME_POINTER: Helps to produce more reliable stack traces.
2. KERNEL IN DEBUG MODE

- The Linux kernel can be put in KGDB mode at boot time via kernel command line option or at run time through files available in /proc.

- To configure KGDB at boot time, use the boot parameters kgdboc and kgdbwait as shown below:
  
  kgdboc=ttymxc0,115200 kgdbwait

- At run time, we can use the commands below to put the kernel in debug mode:

  # echo ttymxc0 > /sys/module/kgdboc/parameters/kgdboc
  # echo g > /proc/sysrq-trigger
3. CONNECTING TO THE TARGET (A)

- On the host, run the GDB client passing the kernel image with debugging symbols:
  
  ```
  $ arm-linux-gdb vmlinux
  ```

- At the GDB command line, configure the serial port and connect to the target:
  
  ```
  (gdb) set serial baud 115200
  (gdb) target remote /dev/ttyUSB0
  ```
AGENT PROXY

✗ If you are using the serial port for both console and KGDB debugging, you will need to use a proxy to manage the serial communication.

✗ A very simple and functional proxy is available in the Linux kernel repository.

$ git clone https://kernel.googlesource.com/pub/scm/utils/kernel/kgdb/agent-proxy
$ cd agent-proxy/
$ make
3. CONNECTING TO THE TARGET (B)

- To start debugging through the serial port using a proxy, first run the proxy program:

  
  `$/agent-proxy 5550^5551 0 /dev/ttyUSB0,115200`

- Open a terminal and run the `telnet` command connect to the target console:

  
  `$/ telnet localhost 5550`

- In another terminal, connect to the target:

  
  `$/ arm-linux-gdb vmlinux
  (gdb) target remote localhost:5551`
AGENT PROXY

Host
- console
- arm-linux-gdb
- Agent Proxy
- Kernel image with debug symbols (vmlinux)

Target
- KGDB
- Kernel Linux (zImage)

Serial port

ARM-Linux-GDB
The kernel provides a collection of helper scripts that can simplify the kernel debugging process.

When enabled in the `CONFIG_GDB_SCRIPTS` config option, it will add Linux awareness debug commands to GDB (lx-).

The documentation is available in the kernel source code at `Documentation/dev-tools/gdb-kernel-debugging.rst`. 
(gdb) apropos lx-
lx-cmdline -- Report the Linux Commandline used in the current kernel
lx-cpus -- List CPU status arrays
lx-dmesg -- Print Linux kernel log buffer
lx-fdtdump -- Output Flattened Device Tree header and dump FDT blob to the filename
lx-iomem -- Identify the IO memory resource locations defined by the kernel
lx-ioports -- Identify the IO port resource locations defined by the kernel
lx-list-check -- Verify a list consistency
lx-1smmod -- List currently loaded modules
lx-mounts -- Report the VFS mounts of the current process namespace
lx-ps -- Dump Linux tasks
lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded modules
lx-version -- Report the Linux Version of the current kernel
KDB

- KDB is a KGDB frontend integrated in the Linux kernel.
- It provides a command line interface integrated in the Linux kernel, allowing you to perform typical debugger operations such as step, stop, run, set breakpoints, disassembly instructions, etc.
- For a long time was available through a set of patches, but was integrated into the kernel mainline in version 2.6.35.
- Does not work at source level, only assembly/machine instruction level!
ENABLING KDB

✗ To use KDB, just compile the kernel with CONFIG_KGDB_KDB enabled.

✗ With this functionality enabled, when the kernel enters in debug mode, the KDB command line interface will automatically be displayed in the console:

[0]kdb>
## KDB HELP

```
[0]kdb> help

<table>
<thead>
<tr>
<th>Command</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>md</td>
<td><code>&lt;vaddr&gt;</code></td>
<td>Display Memory Contents, also mdWcN, e.g. md8c1</td>
</tr>
<tr>
<td>mdr</td>
<td><code>&lt;vaddr&gt; &lt;bytes&gt;</code></td>
<td>Display Raw Memory</td>
</tr>
<tr>
<td>mdp</td>
<td><code>&lt;paddr&gt; &lt;bytes&gt;</code></td>
<td>Display Physical Memory</td>
</tr>
<tr>
<td>go</td>
<td><code>[&lt;vaddr&gt;]</code></td>
<td>Continue Execution</td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td>Display Registers</td>
</tr>
<tr>
<td>rm</td>
<td><code>&lt;reg&gt; &lt;contents&gt;</code></td>
<td>Modify Registers</td>
</tr>
<tr>
<td>ef</td>
<td><code>&lt;vaddr&gt;</code></td>
<td>Display exception frame</td>
</tr>
<tr>
<td>bt</td>
<td><code>[&lt;vaddr&gt;]</code></td>
<td>Stack traceback</td>
</tr>
<tr>
<td>btp</td>
<td><code>&lt;pid&gt;</code></td>
<td>Display stack for process <code>&lt;pid&gt;</code></td>
</tr>
<tr>
<td>btc</td>
<td></td>
<td>Backtrace current process on each cpu</td>
</tr>
<tr>
<td>btt</td>
<td><code>&lt;vaddr&gt;</code></td>
<td>Backtrace process given its struct task address</td>
</tr>
<tr>
<td>env</td>
<td></td>
<td>Show environment variables</td>
</tr>
<tr>
<td>set</td>
<td></td>
<td>Set environment variables</td>
</tr>
<tr>
<td>help</td>
<td></td>
<td>Display Help Message</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td>Display Help Message</td>
</tr>
<tr>
<td>cpu</td>
<td><code>&lt;cpunum&gt;</code></td>
<td>Switch to new cpu</td>
</tr>
<tr>
<td>kgdb</td>
<td></td>
<td>Enter kgdb mode</td>
</tr>
<tr>
<td>ps</td>
<td>`[&lt;flags&gt;</td>
<td>A]`</td>
</tr>
<tr>
<td>pid</td>
<td><code>&lt;pidnum&gt;</code></td>
<td>Switch to another task</td>
</tr>
<tr>
<td>reboot</td>
<td></td>
<td>Reboot the machine immediately</td>
</tr>
<tr>
<td>lsmod</td>
<td></td>
<td>List loaded kernel modules</td>
</tr>
</tbody>
</table>
```

[...]

Embedded Linux Conference Europe 2019

Tracing
TRACING

- There are two main types of tracing: static tracing and dynamic tracing.

- Static tracing is implemented through static probes added in the source code. They have a low processing load, but traced code is limited and defined at compile time.

- Dynamic tracing is implemented through dynamic probes injected into code, allowing to define at runtime the code to be traced. It has a certain processing load, but the range of source code to be traced is much larger.

- Linux kernel tracing documentation is available in the source code at Documentation/trace/.
GCC -PG

(gdb) disassemble gpiod_direction_input
Dump of assembler code for function gpiod_direction_input:
  0xc04faeb8 <+0>:   mov r12, sp
  0xc04faebc <+4>:   push   {r4, r5, r6, r7, r11, r12, lr, pc}
  0xc04faec0 <+8>:   sub r11, r12, #4
  0xc04faec4 <+12>:  push   {lr}           ; (str lr, [sp, #-4]!)
  0xc04faec8 <+16>:  bl   0xc01132e8 <__gnu_mcount_nc>
  0xc04faecc <+20>:  ldr   r1, [pc, #280]; 0xc04fafec <gpiod_directio...
  0xc04faed0 <+24>:  mov r5, r0
  0xc04faed4 <+28>:  bl   0xc04fa924 <validate_desc>
  0xc04faed8 <+32>:  subs  r4, r0, #0
  0xc04faedc <+36>:  b1e  0xc04fa928 <gpiod_direction_input+112>
  0xc04faee0 <+40>:  ldr   r3, [r5]
  0xc04faee4 <+44>:  ldr   r0, [r3, #492]; 0x1ec
  0xc04faee8 <+48>:  ldr   r1, [r3, #496]; 0x1f0
  0xc04faeeec <+52>:  ldr   r2, [r0, #36]  ; 0x24
  0xc04faef0 <+56>:  sub r1, r5, r1
  0xc04faef4 <+60>:  cmp r2, #0
  0xc04faef8 <+64>:  asr r1, r1, #4
  0xc04faefc <+68>:  beq  0xc04fafc0 <gpiod_direction_input+264>
[...]
int gpiod_direction_input(struct gpio_desc *desc)
{
    struct gpio_chip *chip;
    int status = -EINVAL;

    VALIDATE_DESC(desc);
    chip = desc->gdev->chip;

    if (!chip->get || !chip->direction_input) {
        gpiod_warn(desc,
            "%s: missing get() or direction_input() operations\n", __func__);
        return -EIO;
    }

    status = chip->direction_input(chip, gpio_chip_hwgpio(desc));
    if (status == 0)
        clear_bit(FLAG_IS_OUT, &desc->flags);

    trace_gpio_direction(desc_to_gpio(desc), 1, status);

    return status;
}
void input_set_abs_params(struct input_dev *dev, unsigned int axis,
                     int min, int max, int fuzz, int flat)
{
    struct input_absinfo *absinfo;

    input_alloc_absinfo(dev);
    if (!dev->absinfo)
        return;

    absinfo = &dev->absinfo[axis];
    absinfo->minimum = min;
    absinfo->maximum = max;
    absinfo->fuzz = fuzz;
    absinfo->flat = flat;

    dev->absbit[BIT_WORD(axis)] |= BIT_MASK(axis);
}
Several frameworks and tools use these tracing features to instrument the kernel, including:

- Ftrace.
- Trace-cmd.
- Kernelshark.
- SystemTap.
- Perf.
- Kernel live patching.
- And many more!
FTRACE

- Ftrace is the official tracer of the Linux kernel and can be used for debugging and performance/latency analysis.
- It uses static and dynamic kernel tracing mechanisms.
- The trace information is stored in a ring buffer in memory.
- The user interface is via the `tracefs` virtual file system.
ENABLING FTRACE

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <? > for Help, /> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable
USING FTRACE

# mount -t tracefs none /sys/kernel/tracing

# cd /sys/kernel/tracing/

# cat available_tracers
hwlat  blk  function_graph  wakeup_dl  wakeup_rt
wakeup  irqsoff  function  nop
```
# echo function > current_tracer
# cat trace
# tracer: function

#                        _-----=> irqs-off
#                        / _----=> need-resched
#                        | / _---=> hardirq/softirq
#                        || / _--=> preempt-depth
#                        ||| /     delay
# TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
# <idle>-0    [001] d...  23.695208: _raw_spin_lock_irqsave <-hrtimer_next_event_wi...
<idle>-0    [001] d...  23.695209: __hrtimer_next_event_base <-hrtimer_next_event...
<idle>-0    [001] d...  23.695210: __next_base <-__hrtimer_next_event_base
<idle>-0    [001] d...  23.695211: __hrtimer_next_event_base <-hrtimer_next_event...
<idle>-0    [001] d...  23.695212: __next_base <-__hrtimer_next_event_base
<idle>-0    [001] d...  23.695213: __next_base <-__hrtimer_next_event_base
<idle>-0    [001] d...  23.695214: _raw_spin_unlock_irqrestore <-hrtimer_next_event_eve...
<idle>-0    [001] d...  23.695215: get_iowait_load <-menu_select
<idle>-0    [001] d...  23.695217: tick_nohz_tick_stopped <-menu_select
<idle>-0    [001] d...  23.695218: tick_nohz_idle_stop_tick <-do_idle
<idle>-0    [001] d...  23.695219: rcu_idle_enter <-do_idle
<idle>-0    [001] d...  23.695220: call_cpuidle <-do_idle
<idle>-0    [001] d...  23.695221: cpuidle_enter <-call_cpuidle

[...]```
TRACE-CMD & KERNELSHARK

- Trace-cmd is a command line tool that interfaces with ftrace.
- It can configure ftrace, read the buffer and save the data to a file (trace.dat) for further analysis.
- Kernelshark is a graphical tool that works as a frontend to the trace.dat file generated by the trace-cmd tool.
# trace-cmd record -p function -F ls / 
plugin 'function'
CPU0 data recorded at offset=0x30d000 
737280 bytes in size
CPU1 data recorded at offset=0x3c1000 
0 bytes in size

# ls trace.dat
trace.dat

# trace-cmd report
CPU 1 is empty
cpus=2

ls-175  [000]  43.359618: function:
mutex_unlock <-- rb_simple_write
__fsnotify_parent <-- vfs_write
fsnotify <-- vfs_write
__sb_end_write <-- vfs_write
__f_unlock_pos <-- ksys_write
mutex_unlock <-- __f_unlock_pos
down_read_trylock <-- do_page_fault
__cond_resched <-- do_page_fault
rcu_all_qs <-- __cond_resched
find_vma <-- do_page_fault
vmacache_find <-- find_vma

[...]

mut_ex_unlock <-- rb_simple_write
__fsnotify_parent <-- vfs_write
fsnotify <-- vfs_write
__sb_end_write <-- vfs_write
__f_unlock_pos <-- ksys_write
mutex_unlock <-- __f_unlock_pos
down_read_trylock <-- do_page_fault
__cond_resched <-- do_page_fault
rcu_all_qs <-- __cond_resched
find_vma <-- do_page_fault
vmacache_find <-- find_vma
KERNELSHARK

$ kernelshark trace.dat
DEBUGGING LOCKUPS

# echo ondemand > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor  
# task is hanging in kernel space!

# trace-cmd record -p function_graph -O nofuncgraph-irqs -F echo \  
  ondemand > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor  
  plugin 'function_graph'

# ls
trace.dat.cpu0  trace.dat.cpu1

# trace-cmd restore trace.dat.cpu0 trace.dat.cpu1
first = 2 trace.dat.cpu0 args=2
CPU0 data recorded at offset=0x459000
0 bytes in size
CPU1 data recorded at offset=0x459000
1130496 bytes in size

# ls
trace.dat  trace.dat.cpu0  trace.dat.cpu1
DEBUGGING LOCKUPS

![Image of debugging tool output]

<table>
<thead>
<tr>
<th>#</th>
<th>CPU</th>
<th>Time Stamp</th>
<th>Task</th>
<th>PID</th>
<th>Latency</th>
<th>Event</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>46605</td>
<td>0</td>
<td>34.958900</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
<tr>
<td>46607</td>
<td>0</td>
<td>34.958988</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td>mutex_lock()</td>
</tr>
<tr>
<td>46608</td>
<td>0</td>
<td>34.958990</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
<tr>
<td>46609</td>
<td>0</td>
<td>34.958993</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
<tr>
<td>46610</td>
<td>0</td>
<td>34.958995</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
<tr>
<td>46611</td>
<td>0</td>
<td>34.958997</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_exit</td>
<td></td>
</tr>
<tr>
<td>46612</td>
<td>0</td>
<td>34.959000</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
<tr>
<td>46613</td>
<td>0</td>
<td>34.959002</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
<tr>
<td>46614</td>
<td>0</td>
<td>34.959005</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
<tr>
<td>46615</td>
<td>0</td>
<td>34.959008</td>
<td>&lt;...&gt;</td>
<td>178</td>
<td>....</td>
<td>funcgraph_entry</td>
<td></td>
</tr>
</tbody>
</table>
Embedded Linux Conference Europe 2019

Debugging frameworks
KERNEL HACKING

```
[printk and dmesg options ---]
  Compile-time checks and compiler options --->
  -.* Magic SysRq key
  (0x1) Enable magic SysRq key functions by default
  [*] Enable magic SysRq key over serial
  -* Kernel debugging
  Memory Debugging --->
  [ ] Code coverage for fuzzing
  [ ] Debug shared IRQ handlers
  [ ] Debug Lockups and Mangs --->
  [ ] Panic on Oops
  (5) panic timeout
  [ ] Collect scheduler debugging info
  [ ] Collect scheduler statistics
  [ ] Detect stack corruption on calls to schedule()
  [ ] Enable extra timekeeping sanity checking
  [*] Debug preemtible kernel
  Lock Debugging (spinlocks, mutexes, etc...) --->
  -* Stack backtrace support
  [ ] Warn for all uses of unseeded randomness
  [ ] kobject debugging
  [ ] Verbose BUG() reporting (adds 70K)
  [ ] Debug linked list manipulation
  [ ] Debug priority linked list manipulation
  [ ] Debug SG table operations
  [ ] Debug notifier call chains
  [ ] Debug credential management
  RCU Debugging --->
  [ ] Force round-robin CPU selection for unbound work items
  [ ] Force extended blk device numbers and spread then
```

<Select>  < Exit >  < Help >  < Save >  < Load >
MAGIC SYSRQ KEY

- It is a key combination you can hit which the kernel will respond to regardless of whatever else it is doing (unless it is completely locked up).
  - On a virtual TTY: [Alt] + [SysRq] + <command-key>.
  - On a serial console: <break> + <command-key>.

- You can also send the command via /proc/sysrq-trigger.
  
  # echo g > /proc/sysrq-trigger

- This feature is enabled via CONFIG_MAGIC_SYSRQ and can be configured/disabled at runtime via /proc/sys/kernel/sysrq.
MAGIC SYSRQ KEY

Some 'command' keys examples:
- s: sync all mounted filesystems.
- b: immediately reboot the system.
- g: enable KGDB.
- z: dump the ftrace buffer.
- l: shows a stack trace for all active CPUs.
- w: dumps tasks that are in uninterruptable (blocked) state.

More information about this feature, including a list of all supported commands, is available in the Linux kernel source code at Documentation/admin-guide/sysrq.rst.
LOCKUPS

✗ The kernel has some options for identifying kernel space lockups in the "Kernel Hacking" configuration menu, showing a kernel oops message when a task hangs in kernel space.

✗ The CONFIG_HARDLOCKUP_DETECTOR option will monitor lockups for more than 10 seconds without letting an interrupt run.

✗ The CONFIG_BOOTPARAM_HARDLOCKUP_PANIC option will cause a hard lockup to panic.
LOCKUPS

* The CONFIG_SOFTLOCKUP_DETECTOR option will monitor lockups for more than 20 seconds without letting other tasks run.
  * The CONFIG_BOOTPARAM_SOFTLOCKUP_PANIC option will cause a soft lockup to panic.

* The CONFIG_DETECT_HUNG_TASK option will identify tasks locked in the Uninterruptible state “indefinitely”.
  * The CONFIG_BOOTPARAM_HUNG_TASK_PANIC option will cause a hung task to panic.
DEBUGGING LOCKUPS

# hwclock -w -f /dev/rtc1
[ 48.041337] watchdog: BUG: soft lockup - CPU#1 stuck for 22s! [hwclock:180]
[ 48.048322] Modules linked in:
[ 48.051396] CPU: 1 PID: 180 Comm: hwclock Not tainted 4.18.9 #51
[ 48.057412] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[ 48.063964] PC is at snvs_rtc_set_time+0x60/0xc8
[ 48.068599] LR is at _raw_spin_unlock_irqrestore+0x40/0x54
[ 48.080367] sp : d949fdff  ip : d949fd78  fp : d949fe2c
[ 48.085599] r10: c0786554  r9 : bef2bc94  r8 : 00000000
[ 48.090832] r7 : d8e71450  r6 : c0bc74a0  r5 : d840b410  r4 : d949fe58
[ 48.097368] r3 : 1e6a8abe  r2 : 1e6a8abe  r1 : 00000000  r0 : 00000000
[ 48.103904] Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
[ 48.111047] CPU: 1 PID: 180 Comm: hwclock Not tainted 4.18.9 #51
[ 48.116805] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[ ...]
[ 48.253808] [<c0009a30>] (__irq_svc) from [<c0516eeec>] (snvs_rtc_set_time+0x60/0xc8)
[ 48.261571] [<c0516eeec>] (snvs_rtc_set_time) from [<c050c358>] (rtc_set_time+0x94/0x1f0)
[ 48.269676] [<c050c358>] (rtc_set_time) from [<c050dee8>] (rtc_dev_ioctl+0x3a8/0x654)
[ 48.277529] [<c050dee8>] (rtc_dev_ioctl) from [<c019e310>] (do_vfs_ioctl+0x944)
[ 48.285291] [<c019e310>] (do_vfs_ioctl) from [<c019ebec>] (ksys_ioctl+0x44/0x68)
[ 48.292701] [<c019ebec>] (ksys_ioctl) from [<c019ec28>] (sys_ioctl+0x1c)
[ 48.299851] [<c019ec28>] (sys_ioctl) from [<c0009000>] (ret_fast_syscall+0x0/0x28)
DEBUGGING LOCKUPS

$ arm-linux-addr2line -f -e vmlinux 0xc0516eec
snvs_rtc_set_time
/opt/labs/ex/linux/drivers/rtc/rtc-snvs.c:140

$ arm-linux-gdb vmlinux
(gdb) list *(snvs_rtc_set_time+0x60)
0xc0516eec is in snvs_rtc_set_time (drivers/rtc/rtc-snvs.c:140).
135
136    dev_dbg(dev, "After conversion: %ld", time);
137
138    /* Disable RTC first */
139    ret = snvs_rtc_enable(data, false);
140    if (ret)
141        return ret;
142
143    while(1);
MEMORY LEAK

✗ Excessive system memory consumption may be associated with a kernel space memory leak problem.

✗ The kernel has a feature called kmemleak, which can monitor kernel memory allocation routines and identify possible memory leaks.

✗ This feature can be enabled via the CONFIG_DEBUG_KMEMLEAK config option.
With kmemleak enabled, a kernel thread will monitor the memory every 10 minutes and log potential allocated and unfreed memory regions.

```
# ps | grep kmemleak
root    151   2    0    0    800df728 00000000 S kmemleak
```

Information about possible memory leaks will be available in a file called kmemleak inside debugfs:

```
# cat /sys/kernel/debug/kmemleak
```
We can force a memory check and create a list of possible memory leaks by writing `scan` to this file:

```
# echo scan > /sys/kernel/debug/kmemleak
```

To clear the current list of possible memory leaks, we can write `clear` to this file:

```
# echo clear > /sys/kernel/debug/kmemleak
```

Documentation of this feature is available in the kernel source code at `Documentation/dev-tools/kmemleak.rst`. 
USING KMEMLEAK

```bash
# cat /sys/kernel/debug/kmemleak
unreferenced object 0xd9868000 (size 30720):
    comm "sh", pid 179, jiffies 4294943731 (age 19.720s)
    hex dump (first 32 bytes):
        00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        0a 00 07 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 28 6e bf d8 ...
    backtrace:
        [<c015c9e8>] kmalloc_order+0x54/0x5c
        [<c015ca1c>] kmalloc_order_trace+0x2c/0x10c
        [<c03c39ec>] gpiod_set_value_cansleep+0x3c/0x54
        [<c03c827c>] value_store+0x98/0xd8
        [<c042e31c>] dev_attr_store+0x28/0x34
        [<c02112a0>] sysfs_kf_write+0x48/0x54
        [<c021099c>] kernfs_fop_write+0xfc/0x1e0
        [<c0190fa8>] __vfs_write+0x44/0x160
        [<c0191254>] vfs_write+0x44/0x160
        [<c0191490>] ksys_write+0x58/0xbc
        [<c019150c>] sys_write+0x18/0x1c
        [<c0009000>] ret_fast_syscall+0x0/0x28
        [<be829888>] 0xbe829888
```
$ arm-linux-addr2line -f -e vmlinux 0xc03c39ec
gpiod_set_value_cansleep
/opt/labs/ex/linux/drivers/gpio/gpiolib.c:3465

$ arm-linux-gdb vmlinux
(gdb) list *(gpiod_set_value_cansleep+0x3c)
0xc03c39ec is in gpiod_set_value_cansleep (drivers/gpio/gpiolib.c:3465).
3460  void gpiod_set_value_cansleep(struct gpio_desc *desc, int value)
3461  {
3462       might_sleep_if(extra_checks);
3463       VALIDATE_DESC_VOID(desc);
3464       kmalloc(1024*30, GFP_KERNEL);
3465       gpiod_set_value_nocheck(desc, value);
3466  }
3467  EXPORT_SYMBOL_GPL(gpiod_set_value_cansleep);
CONCLUSION

✗ Know your tools!

✗ Use the right tool for the job.

✗ There are many more tools: SystemTap, Perf, eBPF, LTTnG, etc.

✗ Sometimes adding printk() messages may also help! :-)

✗ Debugging is fun!
QUESTIONS?

E-mail    sergio.prado@e-labworks.com
Website   https://e-labworks.com/en