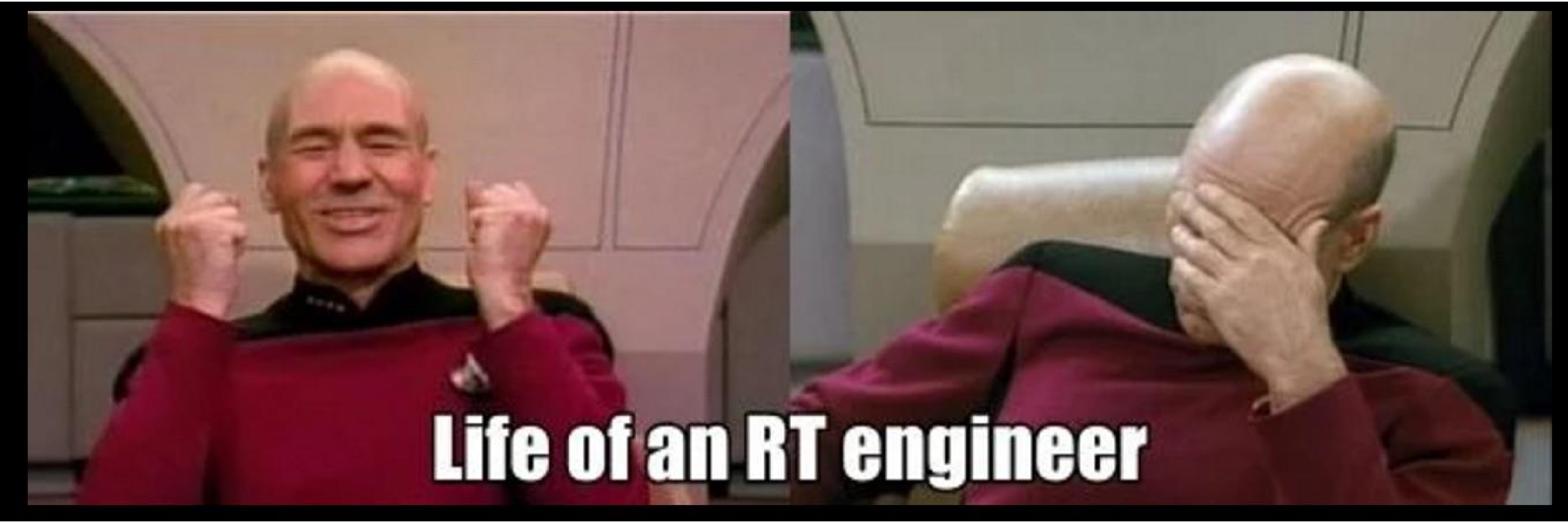
Debugging Real-Time issues in Linux

Embedded Linux Conference, Europe

Joel Fernandes
joel@linuxinternals.org


Life of an RT engineer

Real time - terminology

For purposes of this task:

Period: Time Interval between which RT task will be released at fixed rate

Deadline: Time from when an event occurs to when RT tasks finishes response

For periodic Real time tasks like audio, Period = Deadline

Key concepts - interrupt handling

- Interrupt controller (GIC, APIC etc) sends a hardware signal
- Processor switches mode, banking registers and disabling irq
- Generic Interrupt vector code is called
- Saves the context of the interrupted activity (any context not saved by HW)
- Identify which interrupt occurred, calls relevant ISR

Return:

- Restore context
- Switch processor back to the mode before being interrupted
- Reenable interrupts

Interrupt handling - no hard IRQ nesting in Linux

Tglx removes interrupt nesting officially in hard IRQ handlers..

"The following patch series removes the IRQF_DISABLED functionality from the core interrupt code and runs all interrupt handlers with interrupts disabled."

http://lwn.net/Articles/380536/

Why? Stack overflows is one reason.

Key concepts - kernel preemption (wikipedia def)

Kernel preemption is a method used mainly in monolithic and hybrid kernels where all or most device drivers are run in kernel space, whereby the scheduler is permitted to forcibly perform a context switch (i.e. preemptively schedule; on behalf of a runnable and higher priority process) on a driver or other part of the kernel during its execution, rather than co-operatively waiting for the driver or kernel function (such as a system call) to complete its execution and return control of the processor to the scheduler.

Key concepts - kernel preemption

Cases of Kernel Preemption:

- High priority task wakes up as a result of an interrupt
- Time slice expiration
- System call results in task sleeping

No preemption happens when:

- Preemption explicitly disabled
- Interrupts explicitly disabled
- Spinlock critical sections unless using PREEMPT_RT patchset
- raw spinlock critical sections

Real time - periodic tasks

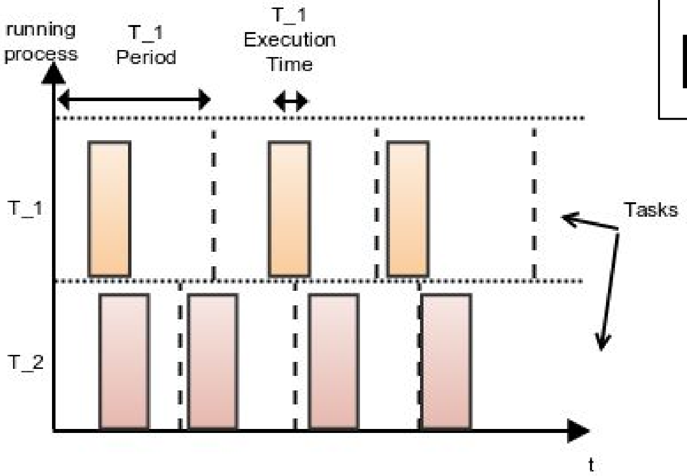


Fig. (1a) - Example of working periodic real-time tasks with deadlines equal to periods. Not deadline misses.

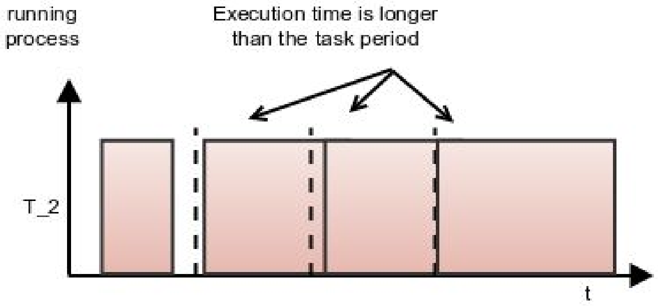


Fig. (1b) - Example of problematic periodic real-time task with deadline misses due to too long execution time.

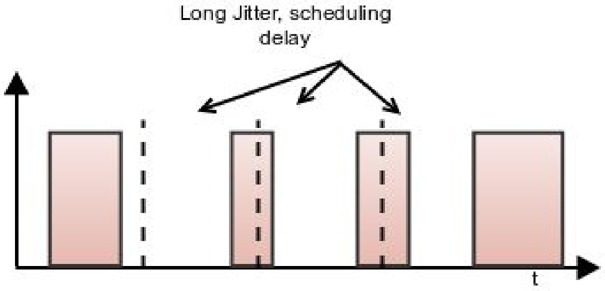


Fig. (1c) - Example of problematic periodic real-time task with deadline misses due to too jitter (scheduling latency).

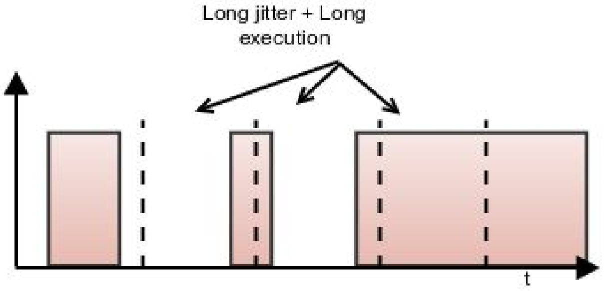
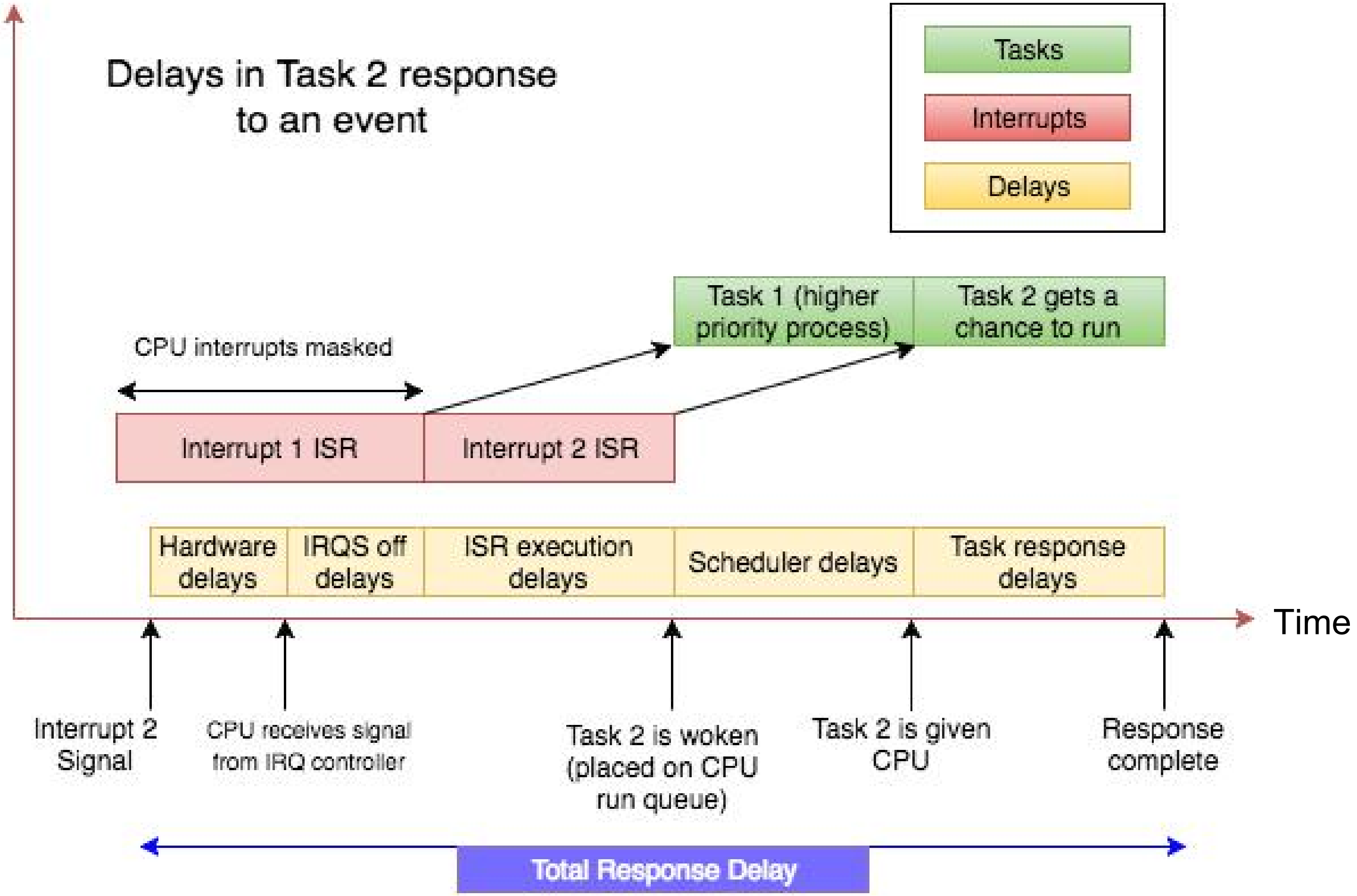


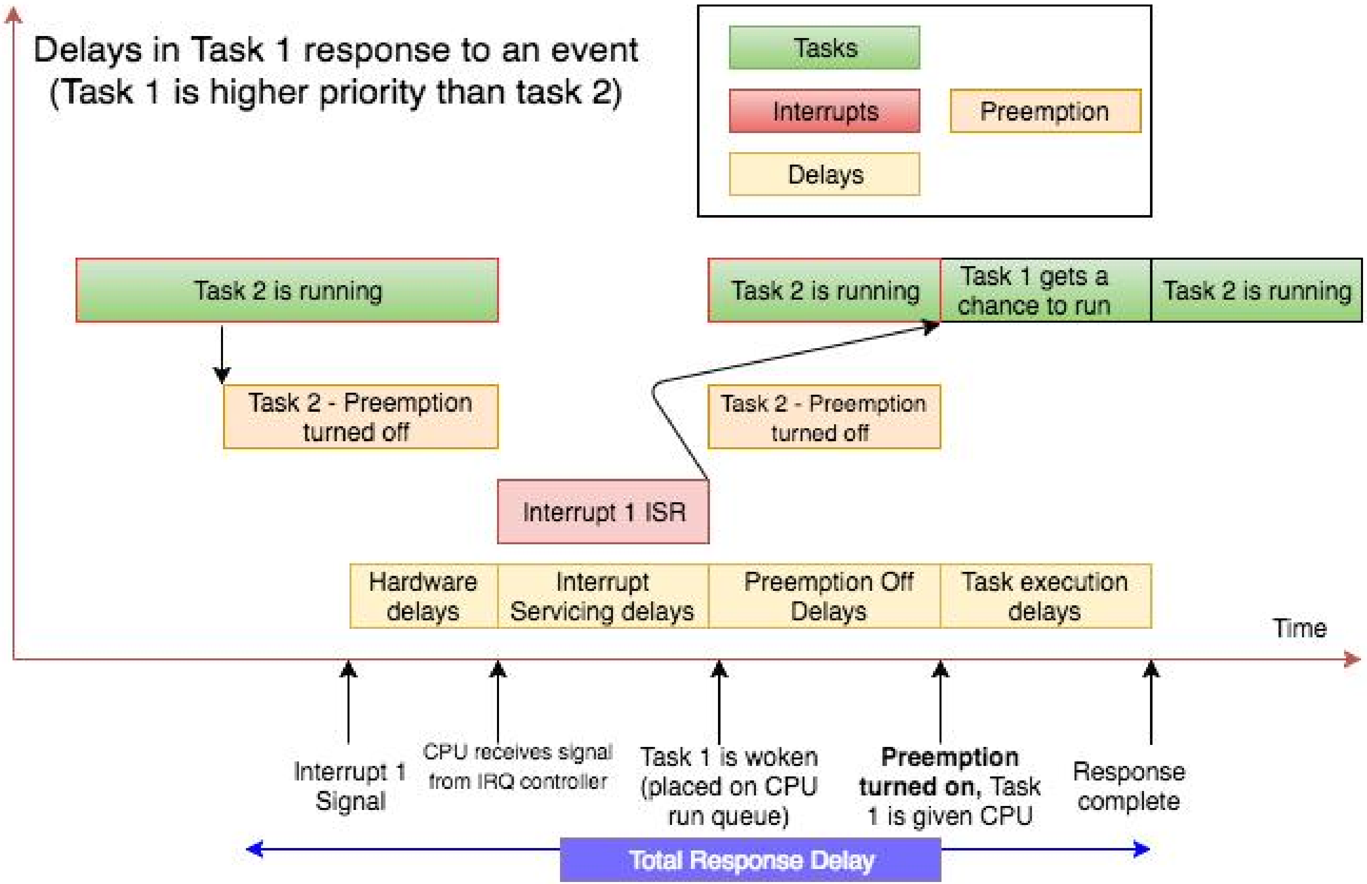
Fig. (1d) - Example of problematic periodic real-time task with deadline misses due to too jitter combined with too long execution time.

Image Source: Eduardo Valentin <edubezval@gmail.com>

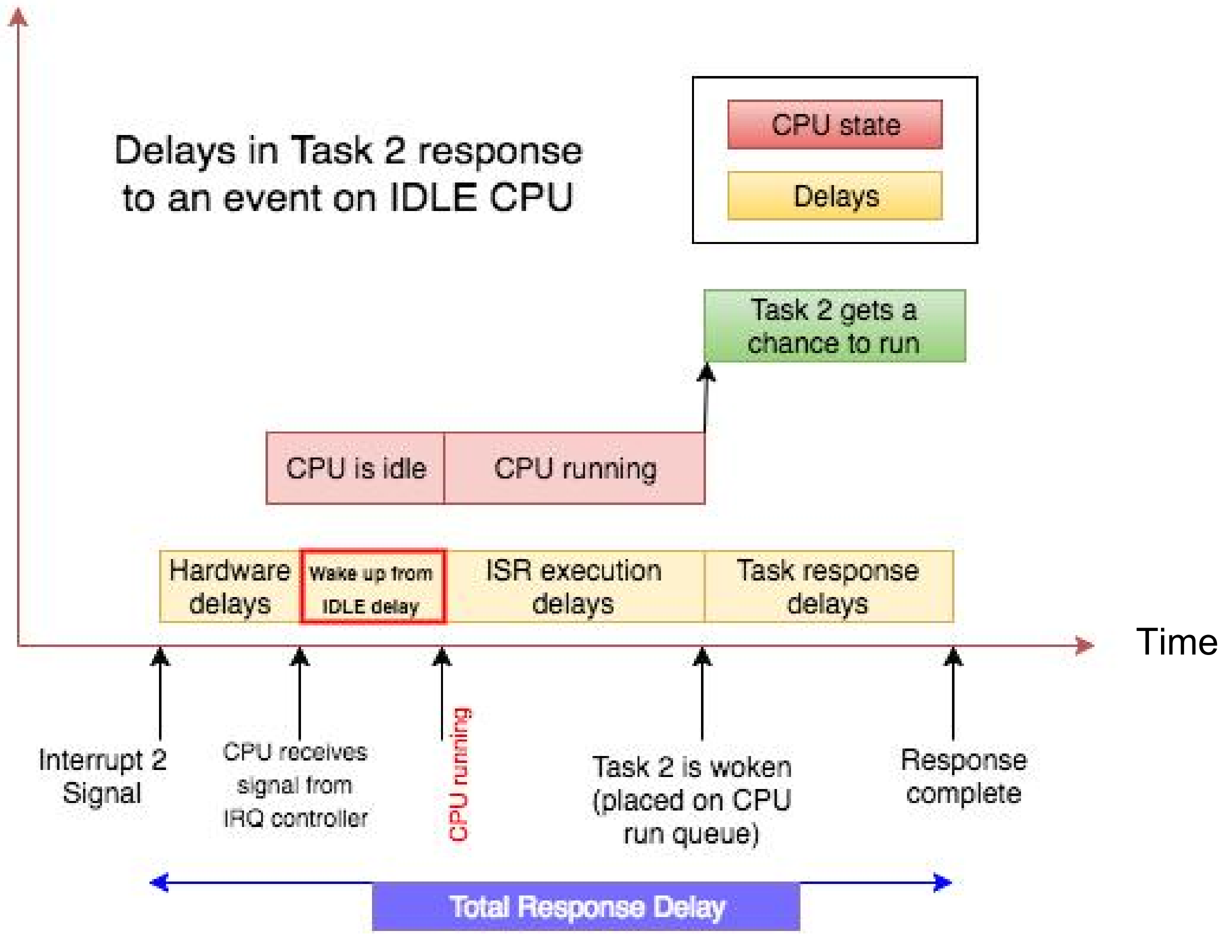
Life of a Wake up (Interrupt 2, Task 2 responds)



Life of a Wake up with Preempt Off delays



Life of a Wake up on an IDLE CPU



Key concepts - spinlocks and mutexes

Spinlocks - spin till you get the lock

Good for small critical sections, sections where you can’t sleep.

Preemption is disabled after spinlock is acquired (unless RT patchset is used)

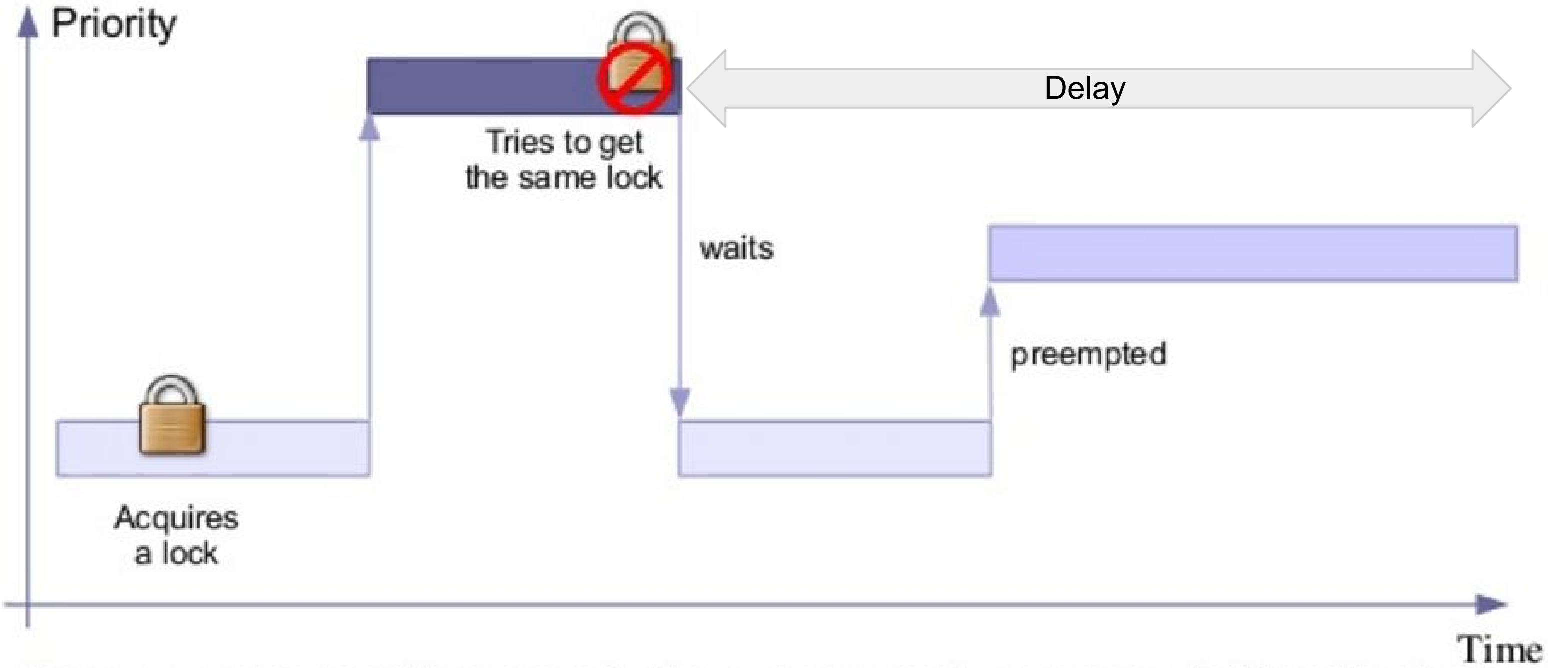
Mutexes - sleep if you cannot get the lock

Good for sections where you can sleep.

Critical sections are preemptible

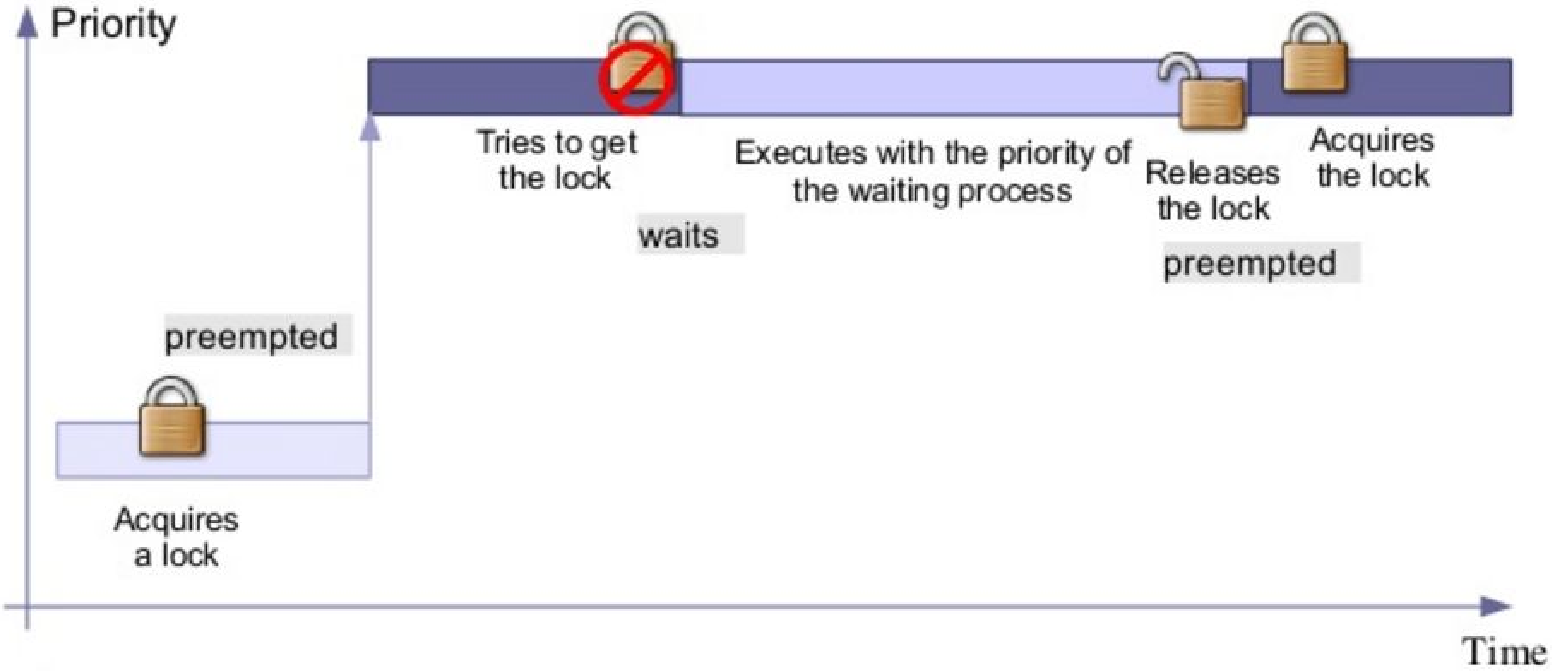
Don’t need to spin and waste CPU (caveat: Linux mutex uses OSQ lock which will spin in some cases, with RT patchset though: mutex lock uses rtmutex)

Key concepts - RT priority inversion



Issue: a process with more priority can preempt a process holding the lock.
The top priority process could wait for a very long time.
This happened with standard Linux before version 2.6.18.

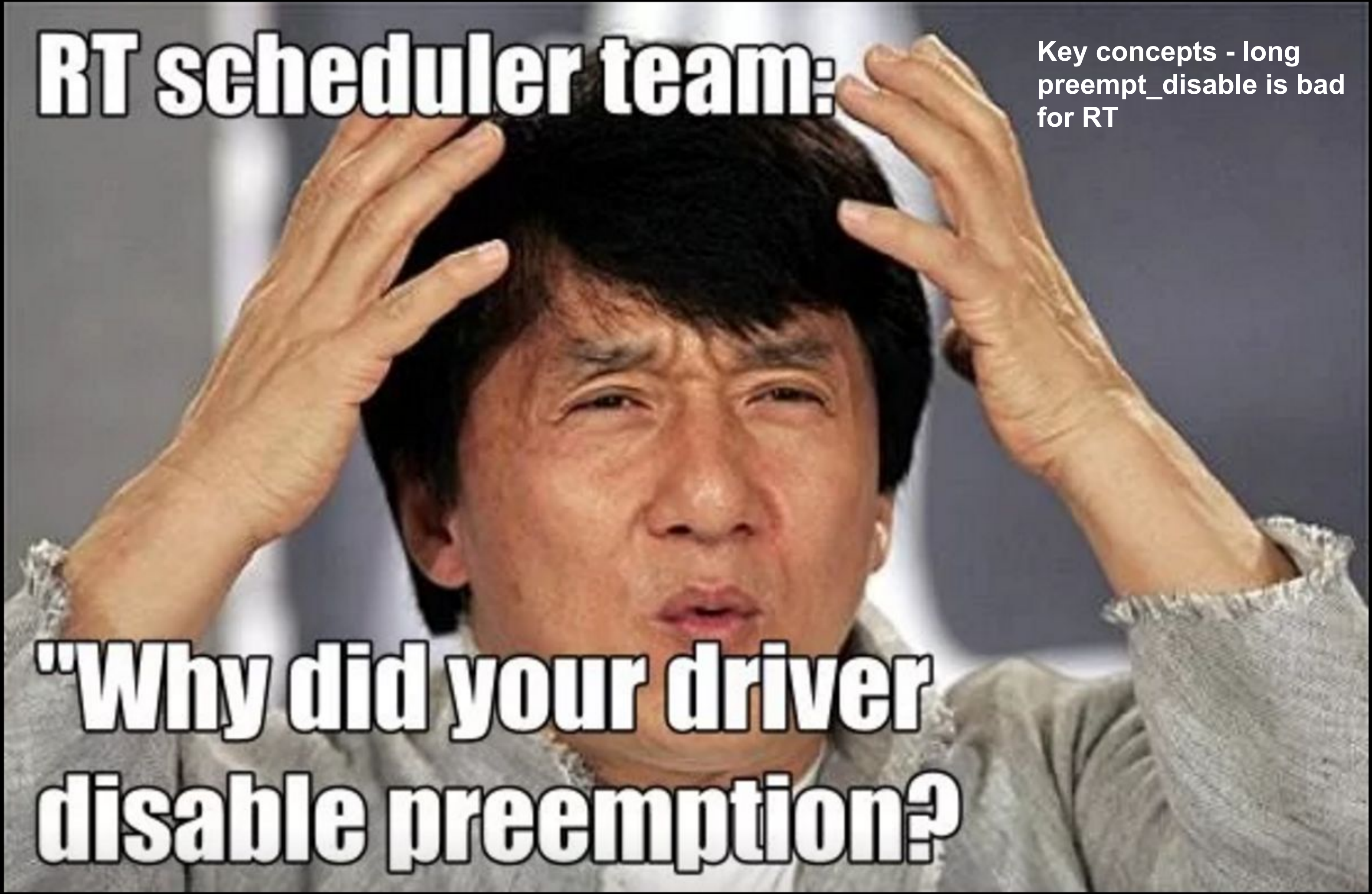
Key concepts - RT priority inversion



Solution: *priority inheritance*. The process holding the lock inherits the priority of the process waiting for the lock with the greatest priority.

Key concepts - RT priority inversion

rtmutex, spin_lock, mutex code with CONFIG_PREEMPT_RT_FULL have priority-inhertiance capabilities.



Key concepts - long preempt_disable is bad for RT

Key concepts - long preempt_disable is bad for RT

preemption is disabled after acquiring spinlock (after raw_spin_lock if RT patchset is used)

Preemption off for long time is a problem (high prio task cannot run)

- Use RT patchset for solving lock preempt-off issues, more on that next slide.
- If you have to disable preemption, use need_resched() to check if higher prio task needs CPU to break out of preempt off section.

RT Patchset - rt.wiki.kernel.org

- Spinlock API uses rt_spin_lock - which sleeps while spinning
- Spinlock critical sections are preemptible
- Mutex uses rtmutex which has PI support and doesn’t optimistically spin
- Convert IRQ top halfs to use IRQ threads

This talk is not about RT Patchset and its features!

Key concepts - scheduler behavior

Scheduler needs to put woken up task on CPU, otherwise we’ve latency

Things preventing that:

- Process priority:
  - Low prio task waits on the rq while high prio given cpu
- Process scheduling class:
  - task is in scheduling class like SCHED_OTHER instead of SCHED_FIFO
- SCHED_FIFO and SCHED_RR always scheduled before SCHED_OTHER/SCHED_BATCH

Note that IRQ threads are SCHED_FIFO tasks with priority 50. **Being threads** their priority can be changed so that other RT tasks have higher priority.

Real time issues - Categories

- Kernel issues
  - Too much time spent in kernel mode
  - Preemption turned off
  - IRQs turned off
  - Spinlocks used where not necessary etc
- Application issues
  - App takes too much time in its period
  - Compiler issues: suboptimal code
  - Poor design - eg. lack of parallel code, cache misses
  - CPU frequency
  - Wrong scheduling priority, policy
  - Page faults in timing critical code
- Hardware issues
  - Bus accesses or interconnect take too long
  - CPU takes too long to come out of idle
  - Interrupt routing misconfigured

Real time issues - kernel - irqs and preempt off

- Interrupts disabled on local CPU for too long
  - Has the effect of locking the CPU to tasks and ISRs
  - An interrupt that wakes up a task can’t execute its ISR to wake it up
- preemption disabled on local CPU for too long
  - Has the effect of locking CPU to other tasks
  - Acceptable if preempt off section checks need_resched()


One does not simply
disable interrupts!

Real time issues - kernel - irqs off examples

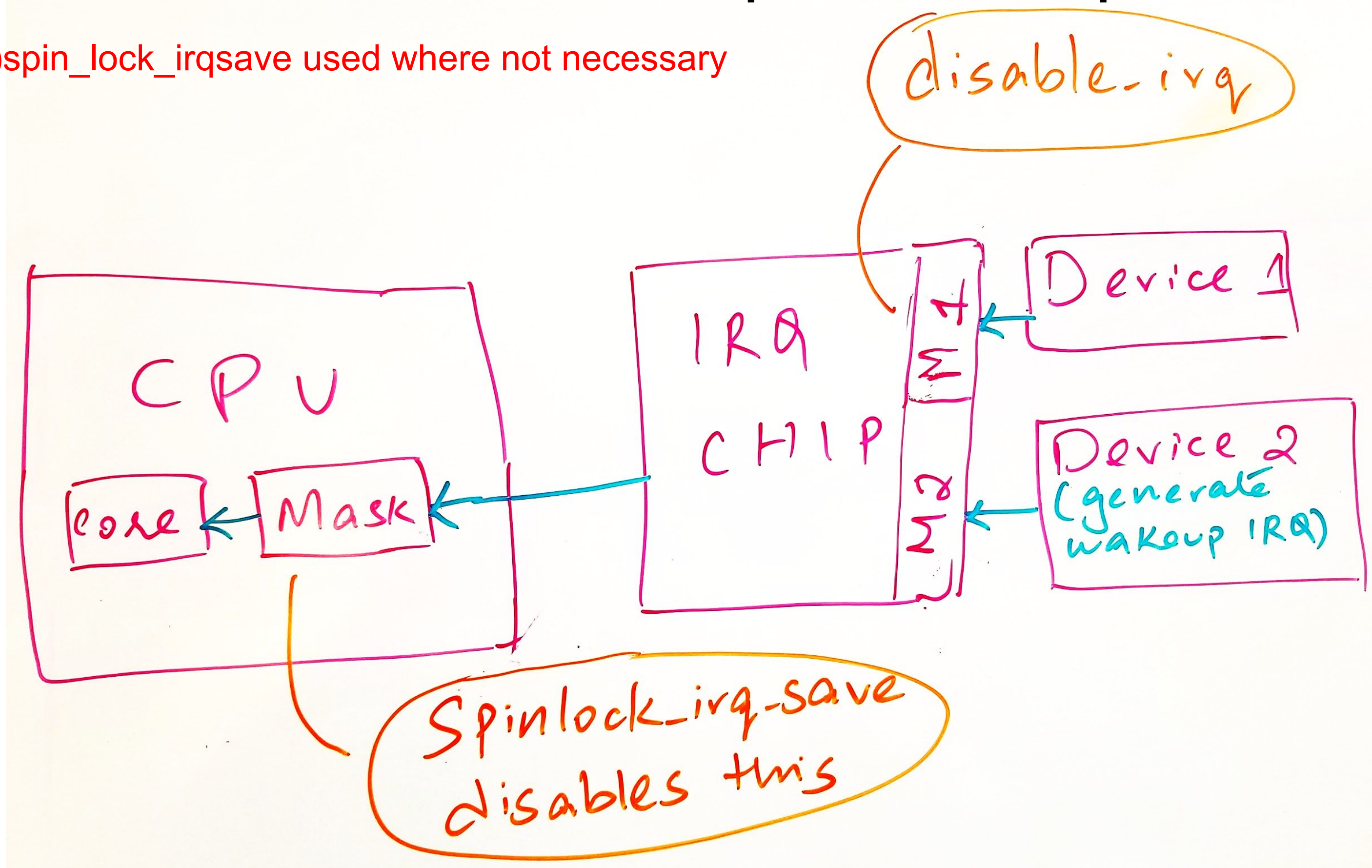
Ex1: (raw_)spinlock_irq_save used where not necessary

Output from irqsoff tracer:

```
=> started at: atomisp_css2_hw_load
=> ended at:   atomisp_css2_hw_load

 => _raw_spin_unlock_irqrestore
 => atomisp_css2_hw_load
 => ia_css_device_load
 => sp_dmem_load
 => ia_css_pipeline_has_stopped
 => ia_css_stream_has_stopped
 => __destroy_stream.isra.4
 => __destroy_streams.constprop.13
 => atomisp_css_stop
```

Real time issues - kernel - irqs off examples

Ex1: (raw_)spin_lock_irqsave used where not necessary



Real time issues - kernel - irqs off examples

Ex1: (raw_)spin_lock_irqsave used where not necessary

```
static void atomisp_css2_hw_load(hrt_address addr, void *to, uint32_t n)
{
     unsigned long flags;
     unsigned i;
     char *_to = (char *)to;
     unsigned int _from = (unsigned int)addr;

     spin_lock_irqsave(&mmio_lock, flags);
    // replace with:
    // disable_irq_nosync(irq)
    // spin_lock(&mmio_lock)
     raw_spin_lock(&pci_config_lock);
     for (i = 0; i < n; i++, _to++, _from++)
          *_to = _hrt_master_port_load_8(_from);
     raw_spin_unlock(&pci_config_lock);
     spin_unlock_irqrestore(&mmio_lock, flags);
}
```

Real time issues - kernel - irqs off examples

Ex1: (raw_)spin_lock_irqsave used where not necessary

Notes:

- With CONFIG_PREEMPT_RT_FULL, spin_lock_irqsave doesn’t disable interrupts. API name maintained for non-RT kernel cases. raw_spin_lock_irqsave still does

- With RT patchset, interrupts run as threads, so irqsave* spinlock variants may not be needed.

Real time issues - kernel - irqs off examples

Ex2: top half handlers taking a long time

Linux doesn’t support hard IRQ nesting, local interrupts are off when executing a hard interrupt handler also known as a top half.

Example 2: SST irq top half takes too long. This steals CPU time from the SST thread and other tasks in the system.

Here's function graph tracer output showing top half irq time issue

(intel_sst_interrupt_mrfld took 3ms)

```
                                                  |  handle_irq_event() {
                                                  |    _raw_spin_unlock() {
                                       0.162 us   |      preempt_count_sub();
                                       1.649 us   |    }
                                                  |    handle_irq_event_percpu() {
                                                  |      intel_sst_interrupt_mrfld() {
                                                  |        _raw_spin_lock_irqsave() {
                                       0.100 us   |          preempt_count_add();
                                       0.975 us   |        }
1329.411728: funcgraph_entry:                     |        _raw_spin_unlock_irqrestore() {
1329.411728: funcgraph_entry:          0.100 us   |          preempt_count_sub();
1329.411729: funcgraph_exit:           0.975 us   |        }
1329.411731: funcgraph_entry:                     |        sst_create_ipc_msg() {
1329.411731: funcgraph_entry:          0.362 us   |          kmem_cache_alloc_trace();
1329.411732: funcgraph_entry:                     |          __kmalloc() {
1329.411733: funcgraph_entry:          0.113 us   |            kmalloc_slab();
1329.411735: funcgraph_exit:           2.450 us   |          }
1329.411736: funcgraph_exit:           4.562 us   |        }
1329.411737: funcgraph_entry:                     |        _raw_spin_lock_irqsave() {
1329.411737: funcgraph_entry:          0.100 us   |          preempt_count_add();
1329.411738: funcgraph_exit:           0.900 us   |        }
1329.411738: funcgraph_entry:                     |        _raw_spin_unlock_irqrestore() {
1329.411739: funcgraph_entry:          0.087 us   |          preempt_count_sub();
1329.411740: funcgraph_exit:           0.925 us   |        }
1329.411740: funcgraph_entry:                     |        intel_sst_clear_intr_mrfld() {
1329.411741: funcgraph_entry:                     |          _raw_spin_lock_irqsave() {
1329.411741: funcgraph_entry:          0.100 us   |            preempt_count_add();
1329.411742: funcgraph_exit:           1.013 us   |          }
1329.411749: funcgraph_entry:                     |          _raw_spin_unlock_irqrestore() {
1329.411749: funcgraph_entry:          0.100 us   |            preempt_count_sub();
1329.411750: funcgraph_exit:           0.900 us   |          }
1329.411751: funcgraph_exit:           9.936 us   |        }
1329.411752: funcgraph_exit:       ! 3167.993 us  |      }
```

Real time issues - kernel - irqs off examples

Ex2: top half handlers taking a long time

Turns out SST was having a hard time accessing the bus… more on that later.

After using function graph tracer to narrow down, I used timestamps + instrumentation to learn that the PCI register space was causing this function to take a long time.

```
memcpy_fromio(&fw_tstamp,
((void *)(sst_drv_ctx->mailbox + sst_drv_ctx->tstamp)
+ (str_id * sizeof(fw_tstamp))),
sizeof(fw_tstamp));
```

Had to disable PCIe power management features to fix these - more on that later.

Real time issues - kernel - irqs off examples

Ex2: top half handlers taking a long time

Tricks to find top half latency issues

Trick 1: Use function_graph tracer with thresholds set to say 1ms, and depth to 3 on the graph function `handle_irq_event`

Output on next slide

Real time issues - kernel - irqs off examples

Using ftrace graph to easily find top half handlers taking forever

```
echo 3 > /d/tracing/max_graph_depth
echo 1000 > /d/tracing/tracing_thresh
trace-cmd record -p function_graph -g handle_irq_event

cat /d/tracing/trace_pipe

 0) ! 3165.056 us |      } /* i2c_dw_isr */
 0) ! 3168.043 us |    } /* handle_irq_event_percpu */
 0) ! 3170.017 us |  } /* handle_irq_event */
 0) ! 2757.569 us |      } /* cherryview_irq_handler */
 0) ! 2763.906 us |    } /* handle_irq_event_percpu */
 0) ! 2766.343 us |  } /* handle_irq_event */
 0) ! 3188.289 us |      } /* atomisp_isr [atomisp_css2401a0_v21] */
 0) ! 3209.286 us |    } /* handle_irq_event_percpu */
 0) ! 3214.460 us |  } /* handle_irq_event */
```

Real time issues - kernel - irqs off examples

Using ftrace graph to easily find top half handlers taking forever

`Trick 2: Use kretprobes`

Idea:

1. Install a dynamic probe at handle_irq_entry function entry
2. At the entry probe handler, get the IRQ name and timestamp
3. At the exit probe handler, get another timestamp
4. Find entry/exit time difference and print warning if too high

Real time issues - kernel - irqs off examples

Using ftrace graph to easily find top half handlers taking forever

Trick 2: Use kretprobes

```
/* the entry_handler to timestamp function entry */
static int entry_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        struct irqprobe_data *data;
        struct irq_desc *desc;
        char *str;

        data = (struct irqprobe_data *)ri->data;
        data->entry_stamp = ktime_get();
        str = data->funcname;

        desc = PT_REGS_PARM1(regs);
        str[0] = 0;

        if (desc->action)
                sprint_symbol_no_offset(str, (unsigned long)desc->action->handler);

        return 0;
}
```

Real time issues - kernel - irqs off examples

Using ftrace graph to easily find top half handlers taking forever

Trick 2: Use kretprobes

```
static int ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        int retval = regs_return_value(regs);
        struct irqprobe_data *data = (struct irqprobe_data *)ri->data;
        s64 delta;
        ktime_t now;

        now = ktime_get();
        delta = ktime_to_ns(ktime_sub(now, data->entry_stamp));
        if (delta > 1000 * 1)
        pr_err("IRQ: %s took %lld ns to execute\n",
                        data->funcname,
                        (long long)delta);
        return 0;
}
```

Real time issues - kernel - irqs off examples

Using ftrace graph to easily find top half handlers taking forever

Trick 2: Use kretprobes

```
static char func_name[NAME_MAX] = "handle_irq_event";
static struct kretprobe irq_kretprobe = {
        .handler                = ret_handler,
        .entry_handler          = entry_handler,
        .data_size              = sizeof(struct irqprobe_data),
        /* Probe up to 20 instances concurrently. */
        .maxactive              = 20,
};

static int __init kretprobe_init(void)
{
        int ret;

        irq_kretprobe.kp.symbol_name = func_name;
        ret = register_kretprobe(&irq_kretprobe);
        if (ret < 0) {
                printk(KERN_INFO "register_kretprobe failed, returned %d\n",
                                ret);
                return -1;
        }
```

Real time issues - kernel - irqs off examples

Using ftrace graph to easily find top half handlers taking forever

Trick 2: Use kretprobes

```
$ # threshold set to 1ms
$ insmod /lib/modules/thardirq.ko

[ 1002.153168] IRQ: i2c_dw_isr took 3062713 ns to execute
[ 1002.183965] IRQ: i2c_dw_isr took 3158637 ns to execute
[ 1002.202206] IRQ: i2c_dw_isr took 3188238 ns to execute
[ 1002.656567] IRQ: i2c_dw_isr took 3176875 ns to execute
[ 1002.854593] IRQ: i2c_dw_isr took 3161238 ns to execute
[ 1003.157660] IRQ: i2c_dw_isr took 3024987 ns to execute
[ 1003.253201] IRQ: i2c_dw_isr took 3044400 ns to execute
[ 1082.237671] IRQ: sdhci_irq took 1209488 ns to execute
[ 1082.253001] IRQ: sdhci_irq took 1229225 ns to execute
[ 1082.274314] IRQ: sdhci_irq took 1229712 ns to execute
[ 1082.302393] IRQ: i2c_dw_isr took 3167850 ns to execute
[ 1213.410491] IRQ: dhdpcie_isr [bcmdhd] took 533287 ns to execute
```

Real time issues - kernel - irqs off examples

Ex2: top half handlers taking a long time

Recommendations:

- Use Threaded IRQ
- Time your hard IRQ sections and make sure they're tiny

Real time issues - kernel - irqs off examples

Ex3: serial console prints in 8250 driver

Serial console prints.. Timestamps show 10ms per print (code execution time between prints was minimal)

```
[   89.966248] atomisp-css2401a0_v21 0000:00:03.0: atomisp_css_isr_thread:no subdev.event:4096
[   89.975753] atomisp-css2401a0_v21 0000:00:03.0: atomisp_css_isr_thread:no subdev.event:4096
[   89.985184] atomisp-css2401a0_v21 0000:00:03.0: atomisp_css_isr_thread:no subdev.event:4096
[   89.994879] atomisp-css2401a0_v21 0000:00:03.0: atomisp_css_isr_thread:no subdev.event:4096
[   90.004302] atomisp-css2401a0_v21 0000:00:03.0: atomisp_css_isr_thread:no subdev.event:4096
[   90.013844] atomisp-css2401a0_v21 0000:00:03.0: atomisp_css_isr_thread:no subdev.event:4096
[   90.023419] atomisp-css2401a0_v21 0000:00:03.0: atomisp_css_isr_thread:no subdev.event:4096
```

Code publicly at ZenFone sources:
https://github.com/ZenfoneArea/android_kernel_asus_zenfone5/blob/master/linux/modules/camera/drivers/media/pci/atomisp2/atomisp_driver/atomisp_compat_css20.c

Real time issues - kernel - irqs off examples

Ex3: serial console prints in 8250 driver

Serial console prints disable interrupts for a long time (seen upto 6ms per line)

```
static void
serial8250_console_write(struct console *co, const char *s, unsigned int count)
{
        struct uart_8250_port *up = &serial8250_ports[co->index];
        struct uart_port *port = &up->port;
        unsigned long flags;
        unsigned int ier;
        int locked = 1;

        touch_nmi_watchdog();

--->    local_irq_save(flags);
        if (port->sysrq) {
```

Real time issues - kernel - irqs off examples

Ex3: serial console prints in 8250 driver

Serial console prints disable interrupts for a long time

Possible solutions:

- Fix the errors/warning (Usually messages are result of errors/warnings)
- Play with the log levels
  - Reduce the log level of the message (use pr_info instead of pr_err)
  - Increase printk log level `(echo <level> > /proc/sys/kernel/printk)`
- Disable serial console - in our final product we disabled this
- Upgrade kernel and use PREEMPT_RT_FULL

Real time issues - kernel - irqs off examples

Ex3: serial console prints in 8250 driver

Note:
Ingo Molnar fixed this already in upstream for -rt

Upstream use spin_lock_irqsave instead of local_irq_save, which ends up not disabling interrupts for CONFIG_PREEMPT_RT_FULL kernels. So if you're using a fairly recent kernel and have PREEMPT_RT_FULL, you shouldn't have this problem.

preemptirqsoff tracer

- Start tracing at start of critical section (preempt disabled or irqs off)
- Stop tracing at stop of critical section (preempt enabled and irqs on)
- Show trace with maximum latency
- Can enable function tracing (default on) to show which function executed in critical section

`More info: Documentation/trace/ftrace.txt`

Real time issues - kernel - real preemptoff issue

Ex4: Lazy max pages

```
static void free_vmap_area_noflush(struct vmap_area *va)
{
        int nr_lazy;

        nr_lazy = atomic_add_return((va->va_end - va->va_start) >> PAGE_SHIFT,
                                    &vmap_lazy_nr);

        /* After this point, we may free va at any time */
        llist_add(&va->purge_list, &vmap_purge_list);

        if (unlikely(nr_lazy > lazy_max_pages()))
                try_purge_vmap_area_lazy();
}
```

Real time issues - kernel - real preemptoff issue

Ex4: Lazy max pages

```
/*
 * Purges all lazily-freed vmap areas.
 *
 * If sync is 0 then don't purge if there is already a purge in progress.
 * If force_flush is 1, then flush kernel TLBs between *start and *end even
 * if we found no lazy vmap areas to unmap (callers can use this to optimise
 * their own TLB flushing).
 * Returns with *start = min(*start, lowest purged address)
 *              *end = max(*end, highest purged address)
 */
static void __purge_vmap_area_lazy(unsigned long *start, unsigned long *end,
                                        int sync, int force_flush)
{
… . . . . . . .……. . .
        if (nr) {
                spin_lock(&vmap_area_lock);
                llist_for_each_entry_safe(va, n_va, valist, purge_list)
                        __free_vmap_area(va);
                spin_unlock(&vmap_area_lock);
        }
        spin_unlock(&purge_lock);
}
```

Real time issues - kernel - real preemptoff issue

Ex4: Lazy max pages

mm/vmalloc.c (line 593)

```
593 static unsigned long lazy_max_pages(void)
594 {
595         unsigned int log;
596
597         log = fls(num_online_cpus());
598
599         return log * (32UL * 1024 * 1024 / PAGE_SIZE);
600 }
```

Real time issues - kernel - real preemptoff issue

Ex4: Lazy max pages.   Preemptirqsoff tracer output (with my tracer fix)

```
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.14.37-x86_64-00190-gddfae4b-dirty
# --------------------------------------------------------------------
# latency: 14707 us, #38619/38619, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: netd-4462 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------


    -----------------
#   => started at: __purge_vmap_area_lazy
#   => ended at:   __purge_vmap_area_lazy
#
#   <...>-4462    2...1    0us : _raw_spin_trylock <-__purge_vmap_area_lazy
```

Real time issues - kernel - real preemptoff issue

Ex4: Lazy max pages.   Preemptirqsoff tracer output (with my tracer fix)

```
<...>-4462    2...3  944us : csd_lock_wait.isra.4 <-smp_call_function_many
<...>-4462    2...3  945us+: csd_lock_wait.isra.4 <-smp_call_function_many
<...>-4462    2...3  948us : preempt_count_sub <-smp_call_function
<...>-4462    2d..2  948us : do_flush_tlb_all <-on_each_cpu
<...>-4462    2...2  949us : preempt_count_sub <-on_each_cpu
<...>-4462    2...1  950us : _raw_spin_lock <-__purge_vmap_area_lazy
<...>-4462    2...1  950us+: preempt_count_add <-_raw_spin_lock
<...>-4462    2...2  952us+: __free_vmap_area <-__purge_vmap_area_lazy
<...>-4462    2...2  954us : kfree_call_rcu <-__free_vmap_area
<...>-4462    2...2  955us : __call_rcu.constprop.63 <-kfree_call_rcu
<...>-4462    2d..2  956us : preempt_count_add <-rcu_is_watching
<...>-4462    2d..3  956us : preempt_count_sub <-rcu_is_watching
<...>-4462    2...2  957us : __free_vmap_area <-__purge_vmap_area_lazy

... rinse repeat..
```

RT patchset : s/spin_lock_irqsave/spin_lock/

```
From include/linux/spin_lock.h

+#ifdef CONFIG_PREEMPT_RT_FULL
+# include <linux/spinlock_rt.h>
+#else /* PREEMPT_RT_FULL */
+

From include/linux/spinlock_rt.h

+#define spin_lock_irqsave(lock, flags)                   \
+       do {                                              \
+               typecheck(unsigned long, flags);          \
+               flags = 0;                                \
+               spin_lock(lock);                          \
+       } while (0)
+
```

RT patchset : s/spin_lock/rt_spin_lock/

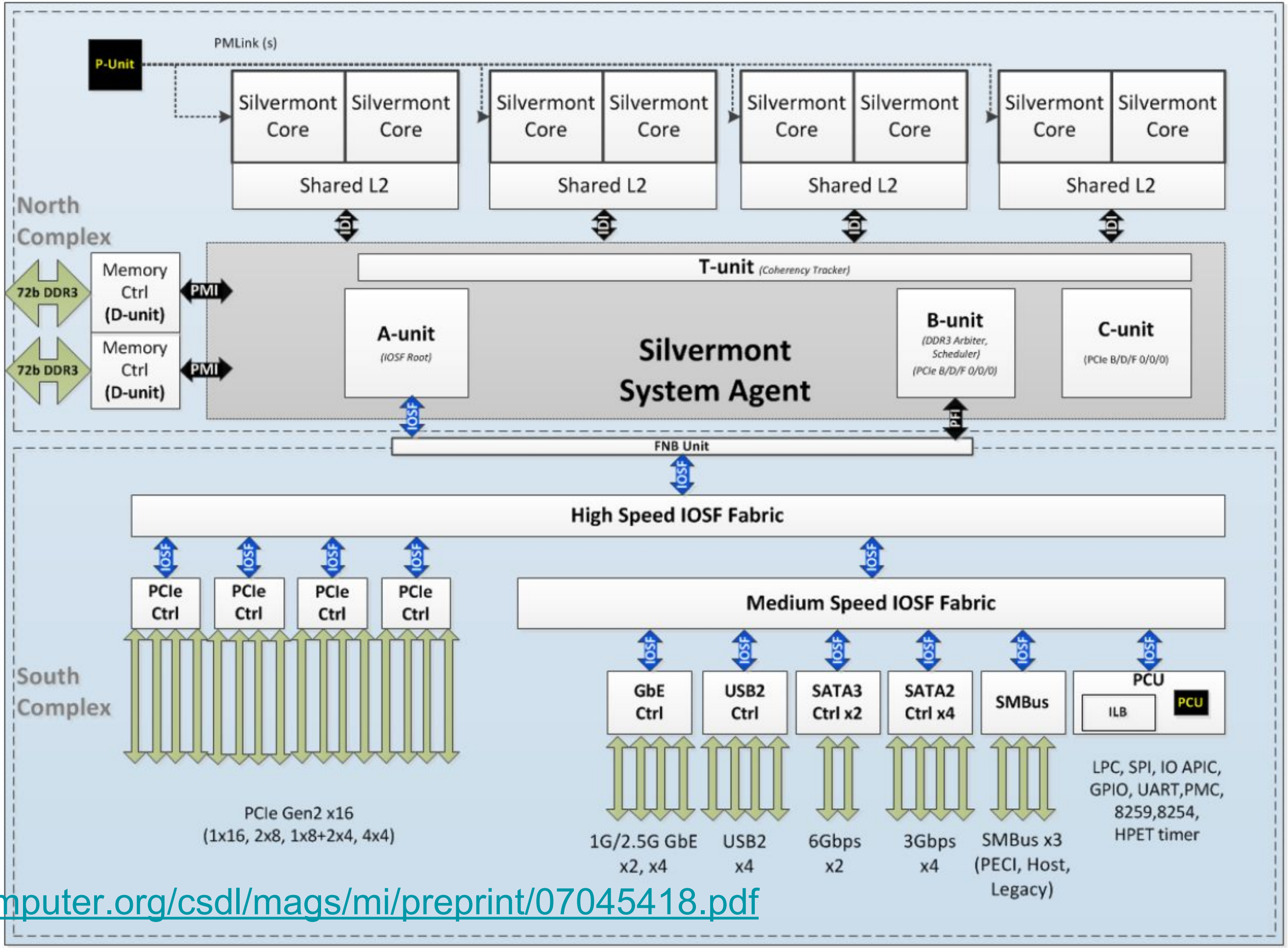
From include/linux/spinlock_rt.h

```
+#define spin_lock(lock)                                         \
+       do {                                            \
+               migrate_disable();                      \
+               rt_spin_lock(lock);                     \
+       } while (0)
+
```

Real time issues - Hardware : Bus related

- Posted transactions: wait for transaction to complete - doesn’t need completion
- Non-Posted transactions: CPU waits for transaction to complete.

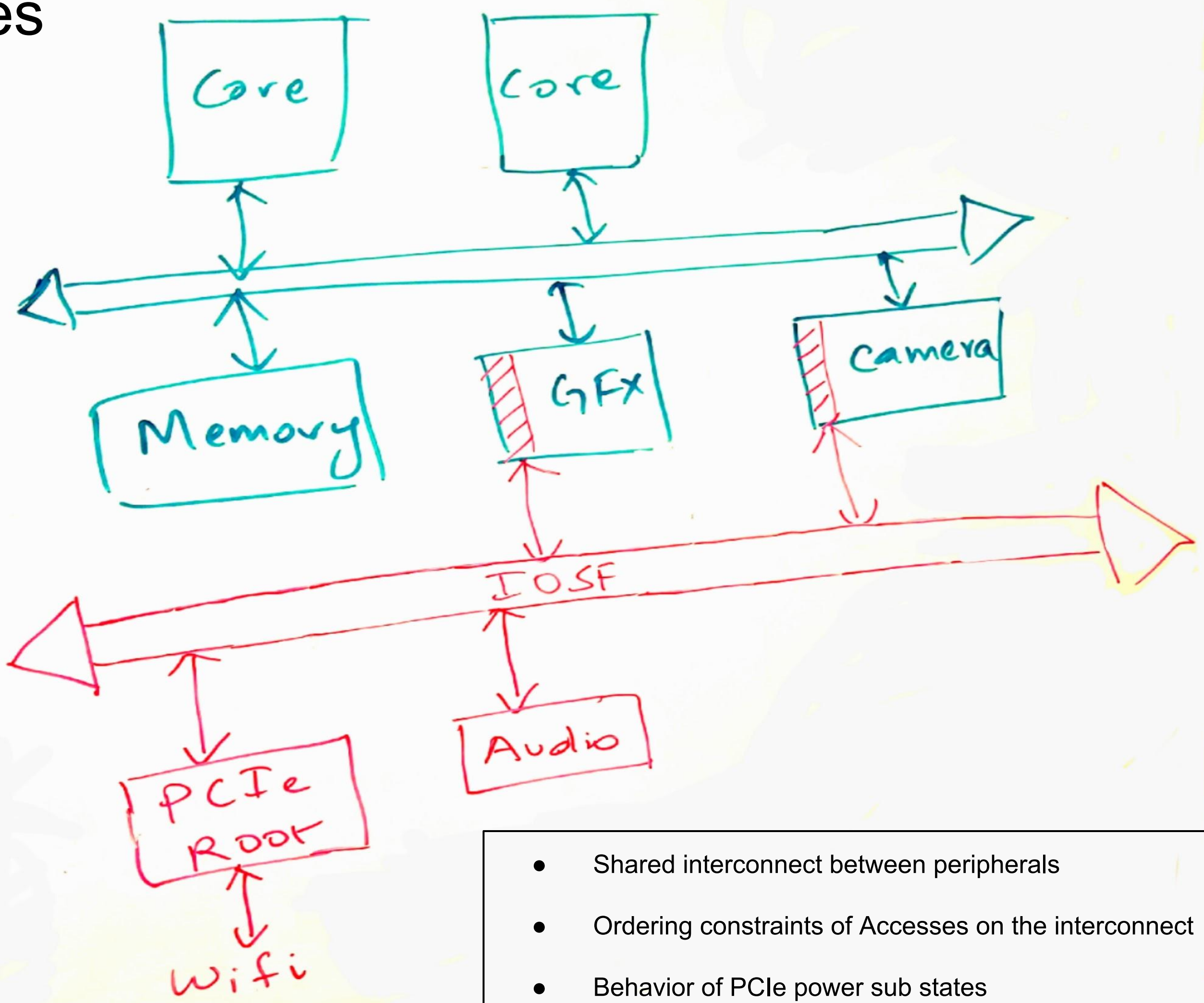
Real time issues - Hardware : Bus related



Source:
https://www.computer.org/csdl/mags/mi/preprint/07045418.pdf

RT Hardware issues

Intel Atom SoC architecture:
http://techreport.com/review/25311/Inside-intel-atom-c2000-series-avoton-processors



- Shared interconnect between peripherals
- Ordering constraints of Accesses on the interconnect
- Behavior of PCIe power sub states

Real time issues - Hardware: : Bus related (wifi recovery)

Broadcom wireless driver sources publicly available at: goo.gl/z1EnJB

```
Function graph tracer with max depth of 1 ...
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 1) ! 3145.770 us |  dhd_bus_cmn_readshared [bcmdhd]();
 1) ! 3159.381 us |  dhd_bus_cmn_readshared [bcmdhd]();
 1)   6.199 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 1) ! 3164.968 us |  dhd_bus_cmn_readshared [bcmdhd]();
 1)   1.775 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 1)   3.124 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 1)   3.237 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 1)   3.149 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 1) + 40.293 us   |  dhd_bus_cmn_readshared [bcmdhd]();
 1)   3.561 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 1)   3.162 us    |  dhd_bus_cmn_readshared [bcmdhd]();
```

Real time issues - Hardware : Bus related (after fixing…)

Broadcom wireless driver sources publicly available at: goo.gl/z1EnJB

```
Function graph tracer with max depth of 1 ...
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0) + 55.129 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 0)   2.087 us     |  dhd_bus_cmn_readshared [bcmdhd]();
 0)   3.774 us     |  dhd_bus_cmn_readshared [bcmdhd]();
 0)   3.949 us     |  dhd_bus_cmn_readshared [bcmdhd]();
 0) + 55.678 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 0)   1.837 us     |  dhd_bus_cmn_readshared [bcmdhd]();
 0)   3.612 us     |  dhd_bus_cmn_readshared [bcmdhd]();
 0)   3.987 us     |  dhd_bus_cmn_readshared [bcmdhd]();
 0) + 55.504 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 0) + 53.005 us    |  dhd_bus_cmn_readshared [bcmdhd]();
 0)   3.312 us     |  dhd_bus_cmn_readshared [bcmdhd]();
```

Others peripheral suffer too,
here we see audio was suffering from the same issue:

```
memcpy_fromio(msg->mailbox_data,
 drv->mailbox + drv->mailbox_recv_offset, size);
```

```
                                                  handle_irq_event() {
                                                    _raw_spin_unlock() {
                                                      preempt_count_sub();
                                                    }
                                                    handle_irq_event_percpu() {
                                                      intel_sst_interrupt_mrfld() {
                                                        _raw_spin_lock_irqsave() {
1329.411724: funcgraph_entry:        0.100 us   |         preempt_count_add();
1329.411725: funcgraph_exit:         0.975 us   |       }
1329.411728: funcgraph_entry:                   |       _raw_spin_unlock_irqrestore() {
1329.411728: funcgraph_entry:        0.100 us   |         preempt_count_sub();
1329.411729: funcgraph_exit:         0.975 us   |       }
1329.411731: funcgraph_entry:                   |       sst_create_ipc_msg() {
1329.411731: funcgraph_entry:        0.362 us   |         kmem_cache_alloc_trace();
1329.411732: funcgraph_entry:                   |         __kmalloc() {
1329.411733: funcgraph_entry:        0.113 us   |           kmalloc_slab();
1329.411735: funcgraph_exit:         2.450 us   |         }
1329.411736: funcgraph_exit:         4.562 us   |       }
1329.411737: funcgraph_entry:                   |       _raw_spin_lock_irqsave() {
1329.411737: funcgraph_entry:        0.100 us   |         preempt_count_add();
1329.411738: funcgraph_exit:         0.900 us   |       }
1329.411738: funcgraph_entry:                   |       _raw_spin_unlock_irqrestore() {
1329.411739: funcgraph_entry:        0.087 us   |         preempt_count_sub();
1329.411740: funcgraph_exit:         0.925 us   |       }
1329.411740: funcgraph_entry:                   |       intel_sst_clear_intr_mrfld() {
1329.411741: funcgraph_entry:                   |         _raw_spin_lock_irqsave() {
1329.411741: funcgraph_entry:        0.100 us   |           preempt_count_add();
1329.411742: funcgraph_exit:         1.013 us   |         }
1329.411749: funcgraph_entry:                   |         _raw_spin_unlock_irqrestore() {
1329.411749: funcgraph_entry:        0.100 us   |           preempt_count_sub();
1329.411750: funcgraph_exit:         0.900 us   |         }
1329.411751: funcgraph_exit:         9.936 us   |       }
1329.411752: funcgraph_exit:      ! 3167.993 us |     }
```

Real time issues - Hardware : CPU wakeup from idle

- PM QoS framework
- **`cpuidle_latency_notify`** is called when latency requirement change.
- All cores have to be woken up to calculate new C-state
- Involves sending an IPI (inter-processor interrupt) to all cores to wake-up
- Preemption is turned off until all CPUs wakeup

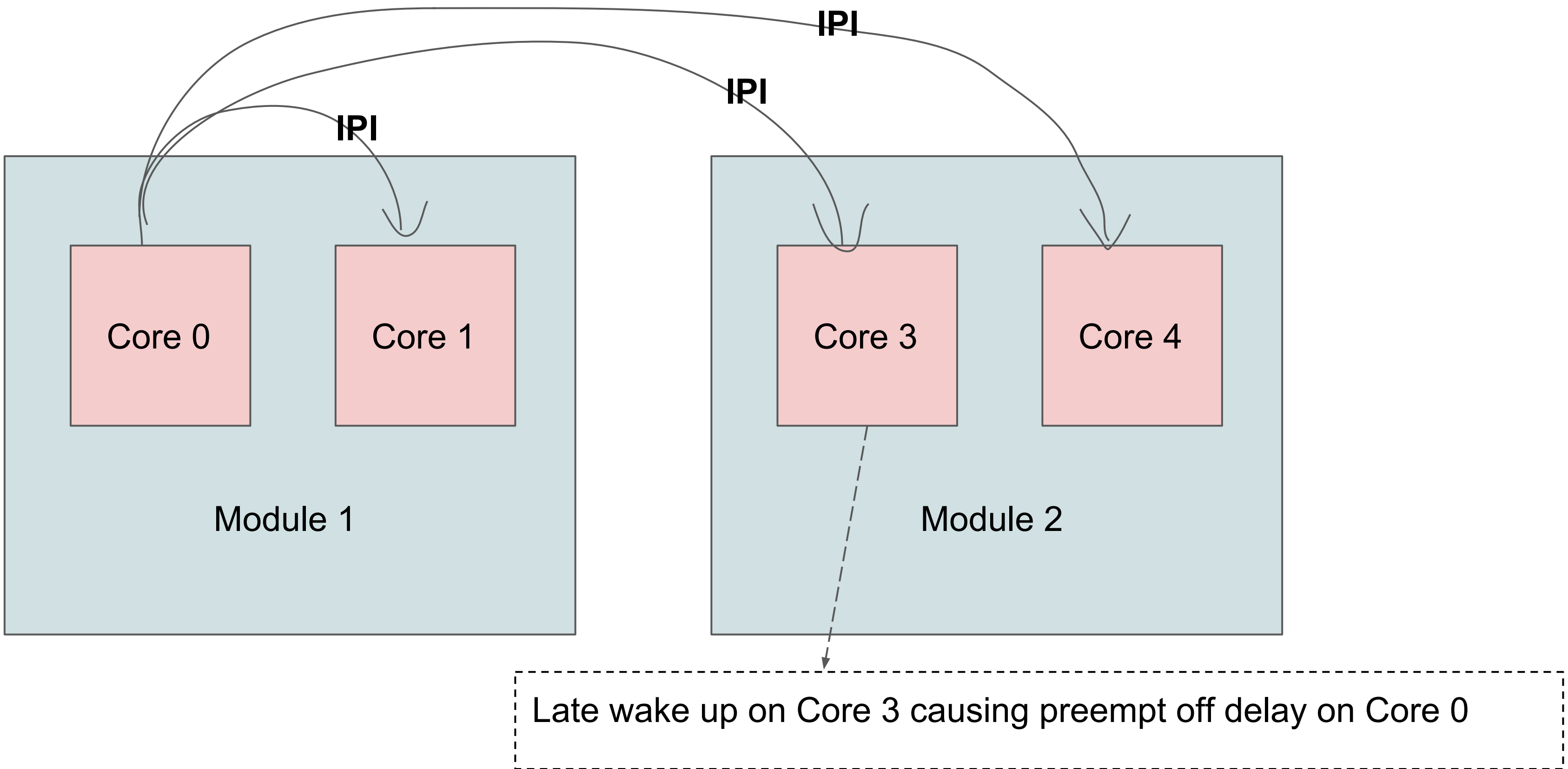
Real time issues - Hardware : CPU wakeup from idle

```
/*
* This function gets called when a part of the kernel has a new latency
* requirement.  This means we need to get all processors out of their C-state,
* and then recalculate a new suitable C-state. Just do a cross-cpu IPI; that
* wakes them all right up.
*/
static int cpuidle_latency_notify(struct notifier_block *b,
                unsigned long l, void *v)
{
        smp_call_function(smp_callback, NULL, 1);
        return NOTIFY_OK;
}


int smp_call_function(smp_call_func_t func, void *info, int wait)
{
        preempt_disable();
        smp_call_function_many(cpu_online_mask, func, info, t);
        preempt_enable();

        return 0;
}
```

Real time issues - Hardware : CPU wakeup from idle

```
mmcqd/0-1408    0...1     0us : smp_call_function <-cpuidle_latency_notify
mmcqd/0-1408    0...1     1us : smp_call_function_many <-smp_call_function
mmcqd/0-1408    0...1     2us : csd_lock_wait.isra.4 <-smp_call_function_many
mmcqd/0-1408    0...1     3us : csd_lock_wait.isra.4 <-smp_call_function_many   ← Ensure its unlocked
mmcqd/0-1408    0...1     3us : csd_lock_wait.isra.4 <-smp_call_function_many
mmcqd/0-1408    0...1     4us : native_send_call_func_ipi <-smp_call_function_many
mmcqd/0-1408    0...1     5us : flat_send_IPI_allbutself <-native_send_call_func_ipi   ← Send IPI to all
mmcqd/0-1408    0...1     6us+: csd_lock_wait.isra.4 <-smp_call_function_many   ← first csd_lock_wait succeeds
mmcqd/0-1408    0...1    10us!: csd_lock_wait.isra.4 <-smp_call_function_many   ← second one always delayed
mmcqd/0-1408    0d..1   121us : smp_apic_timer_interrupt <-apic_timer_interrupt
mmcqd/0-1408    0d..1   122us : irq_enter <-smp_apic_timer_interrupt
...
mmcqd/0-1408    0dNh1 14023us : irq_exit <-do_IRQ
mmcqd/0-1408    0dNh1 14024us : irqtime_account_irq <-irq_exit
mmcqd/0-1408    0dNh1 14024us : preempt_count_sub <-irq_exit
mmcqd/0-1408    0dN.1 14024us : idle_cpu <-irq_exit
mmcqd/0-1408    0dN.1 14025us : rcu_irq_exit <-irq_exit
mmcqd/0-1408    0.N.1 14026us : csd_lock_wait.isra.4 <-smp_call_function_many   ← Next one after long time
mmcqd/0-1408    0.N.1 14027us : preempt_count_sub <-smp_call_function
mmcqd/0-1408    0.N.1 14028us : smp_call_function <-cpuidle_latency_notify
mmcqd/0-1408    0.N.1 14029us+: trace_preempt_on <-cpuidle_latency_notify
mmcqd/0-1408    0.N.1 14098us : <stack trace>
```

Real time issues - Hardware : CPU wakeup from idle



Real time - application issues

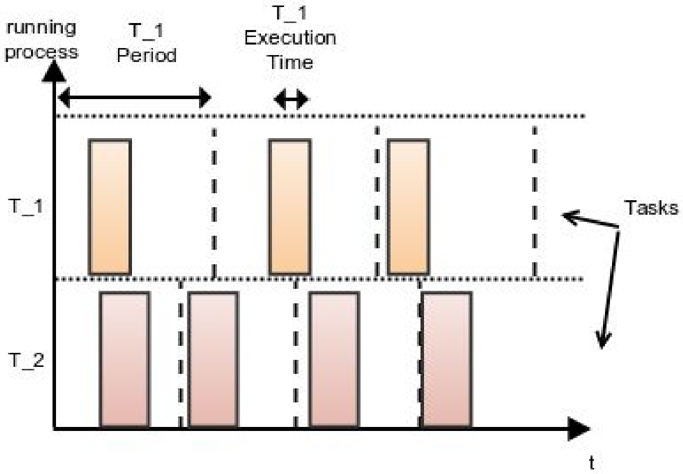


Fig. (1a) - Example of working periodic real-time tasks with deadlines equal to periods. Not deadline misses.

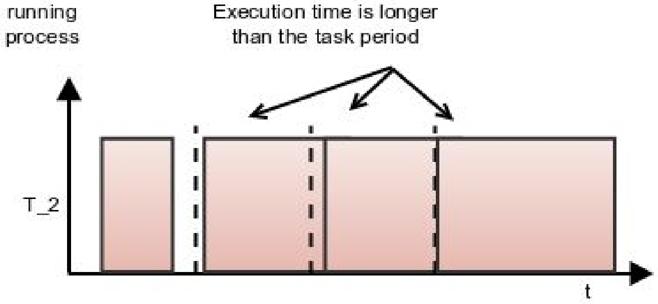


Fig. (1b) - Example of problematic periodic real-time task with deadline misses due to too long execution time.

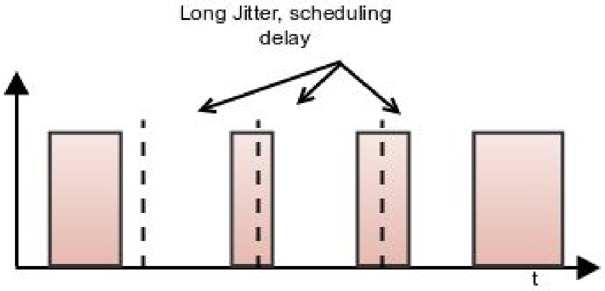


Fig. (1c) - Example of problematic periodic real-time task with deadline misses due to too jitter (scheduling latency).

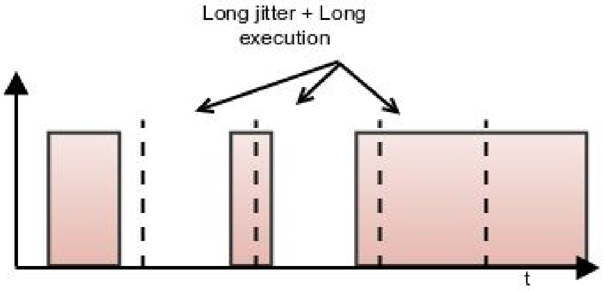


Fig. (1d) - Example of problematic periodic real-time task with deadline misses due to too jitter combined with too long execution time.

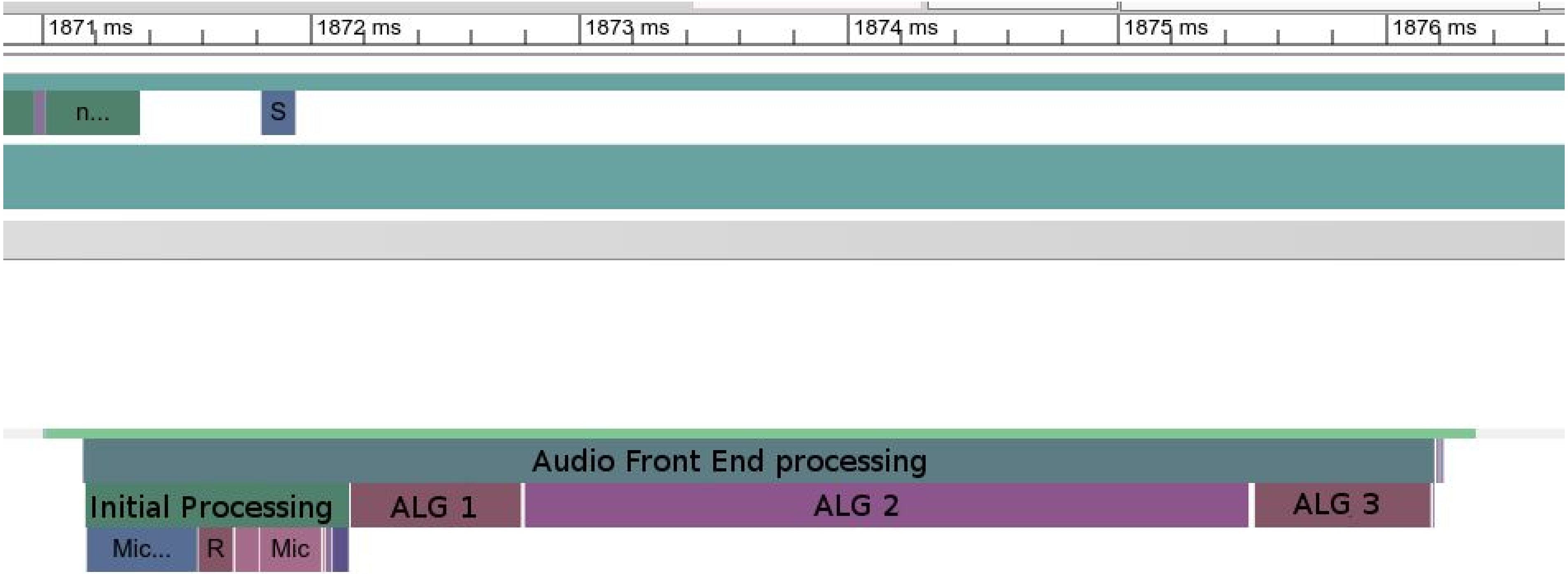
Image Source: Eduardo Valentin <edubezval@gmail.com>

Amazon Echo

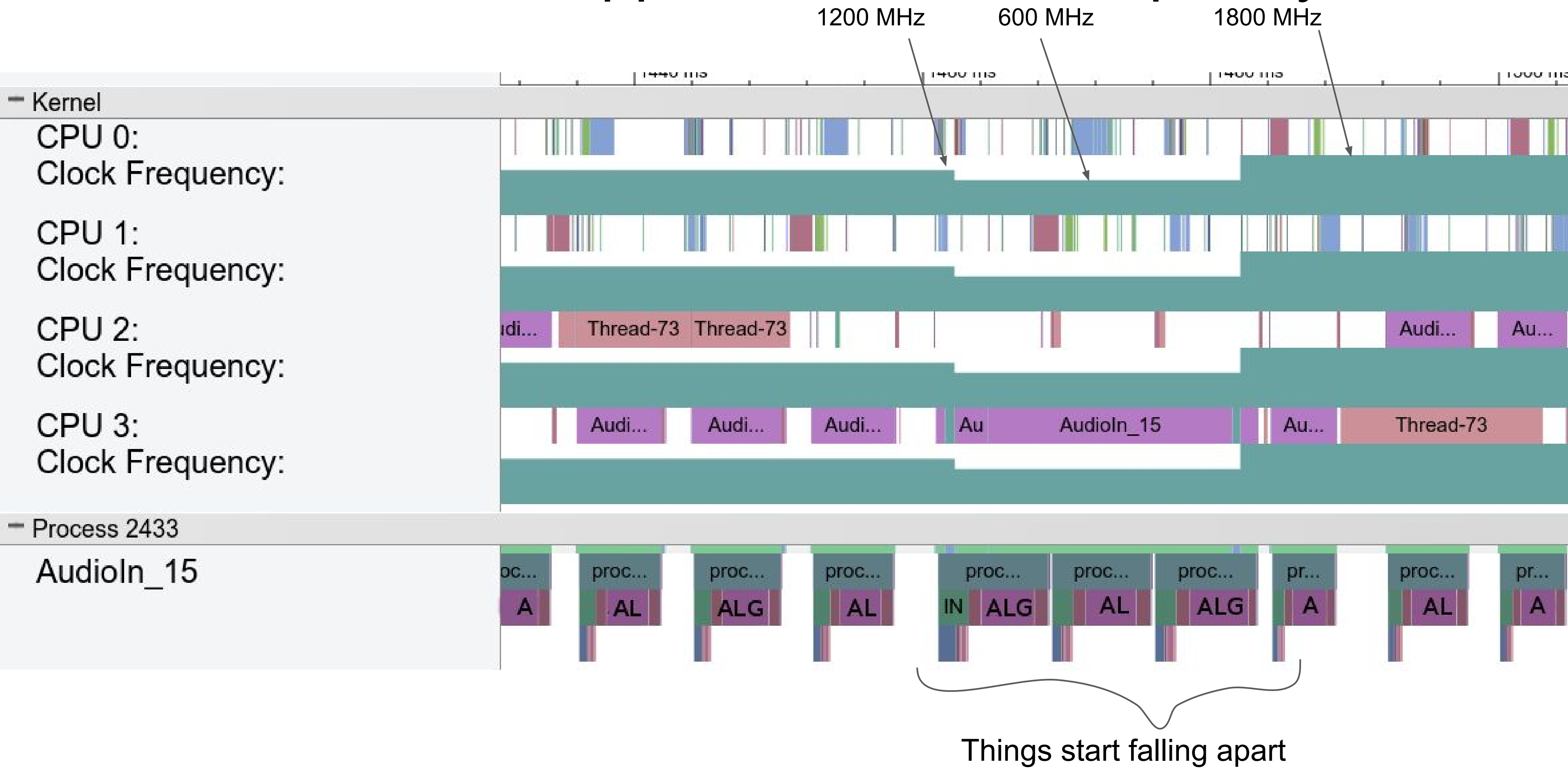
- A product that has stringent requirements on response time and customer experience
- Always listening for “Alexa”
- Examples of audio algorithms always running.
    - From amazon.com/echo product page

> “Tucked under the light ring is an array of seven microphones that use beam-forming technology and enhanced noise cancellation. With far-field voice recognition, Echo can hear you ask a question from any direction—even while playing music.

Analysis of Audio pipeline with Android systrace



Real time issues: application - CPU frequency



Real time issues: application - Cache misses

Analysis of Audio bottlenecks with perf utility

```
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 4K of event 'cache-misses'
# Event count (approx.): 11166978
#
# Overhead      Command          Shared Object
# ........  ...........  ....................  ..........................................
#
    30.21%  mediaserver  libc.so               [.] memmove
            |
            --- memmove
               |
               |--80.10%--

                         Audio Algorithm Hotpath

                         _ZN7android6Thread11_threadLoopEPv
                         _ZN13thread_data_t10trampolineEPKS_
                         _ZL15__pthread_startPv
                         __start_thread
                         __bionic_clone
                         0x0
                         _ZN7android6Parcel14releaseObjectsEv
                         _ZN7android6Parcel14freeDataNoInitEv
                         _ZN7android6ParcelD1Ev
                         _ZN7android20BpAudioFlingerClient15ioConfigChangedEiiPKv
                         0x0
                         0x648d0006
```

Generic Android Audio Pipeline

Real time issues: application - Other issues?

- Lack of parallelization ?
- Compiler issues ?

Real time issues: system - Scheduling

- Get your priorities right
- Use the right policy

Real time issues: Scheduling: Find issues..

Using scheduler delay statistics (CONFIG_SCHEDSTATS)

`cat /proc/<pid>/sched` for AudioIn thread in Android (CFS policy)

```
se.exec_start                    :     115703.404109
se.vruntime                      :    1761777.444578
se.sum_exec_runtime              :       1619.691347
se.statistics.wait_start         :          0.000000
se.statistics.sleep_start        :     118028.360499
se.statistics.block_start        :          0.000000
se.statistics.sleep_max          :      23528.490473
se.statistics.block_max          :          0.009298
se.statistics.exec_max           :          6.342921
se.statistics.slice_max          :          0.430483  <- only for high load tasks
se.statistics.wait_max           :         12.998756
se.statistics.wait_sum           :        279.959758
```

Real time issues: Scheduling: Find issues..

`cat /proc/<pid>/sched` for AudioIn thread in Android (RT policy)

```
pid 4732's current scheduling policy: SCHED_FIFO
pid 4732's current scheduling priority: 2

AudioIn_1A (4732, #threads: 34)
-------------------------------------------------------------------
se.exec_start                                :        287659.027531
se.vruntime                                  :            -4.755003
se.sum_exec_runtime                          :         11894.018812
se.statistics.wait_start                     :             0.000000
se.statistics.sleep_start                    :             0.000000
se.statistics.block_start                    :             0.000000
se.statistics.sleep_max                      :             0.000000
se.statistics.block_max                      :             0.000000
se.statistics.exec_max                       :             6.541440
se.statistics.slice_max                      :             0.000000
se.statistics.wait_max                       :             0.000000
se.statistics.wait_sum                       :             0.000000
```

Not that useful for RT!

Real time issues: Scheduling: Find issues..

Getting scheduling delays for any scheduler policy

/proc/pid/schedstat already calculates total run queue delays per task

Why not also find the maximum run delay?

```
@@ -68,6 +68,8 @@ static inline void sched_info_dequeued(struct rq *rq, struct task_struct *t)
                        delta = now - t->sched_info.last_queued;
        sched_info_reset_dequeued(t);
        t->sched_info.run_delay += delta;
+       schedstat_set(t->se.statistics.run_delay_max,
+                           max(delta, t->se.statistics.run_delay_max));
```

Real time issues: Scheduling: Find issues..

```
maximum runqueue delay in /proc/<pid>/schedstat for RT task

-------------------------------------------------------------------
se.exec_start                                    :      495117.327836
se.vruntime                                      :          -5.000000
se.sum_exec_runtime                              :       22153.249640
se.statistics.wait_start                         :           0.000000
se.statistics.sleep_start                        :           0.000000
se.statistics.block_start                        :           0.000000
se.statistics.sleep_max                          :           0.000000
se.statistics.block_max                          :           0.000000
se.statistics.exec_max                           :           0.881424
se.statistics.exec_hist[HIST_0_10US]             :               5864
se.statistics.exec_hist[HIST_10US_100US]         :               4834
se.statistics.exec_hist[HIST_100US_1MS]          :              22478
se.statistics.exec_hist[HIST_1MS_10MS]           :                  0
se.statistics.exec_hist[HIST_10MS_100MS]         :                  0
se.statistics.exec_hist[HIST_MORE_100MS]         :                  0
se.statistics.slice_max                          :           0.000000
se.statistics.wait_max                           :           0.000000
se.statistics.run_delay_max                      :           0.377203
```

Real time issues: Scheduling: Find issues..

Room for improvement even with this:

- We handle cases where a task is queued -> CPU
- What about cases where a task is migrated after queuing?
  (queued -> dequeued -> queued on another rq -> CPU)
- This works for run_delay because its cumulative, but not so much for run_delay_max:

Here's a modification of Steven's rt lock test (https://lwn.net/Articles/425583/)

Task	run_delay_max	rdm_naive	run_migr_max	nr_running_migr	old_rq_delay_max
0:	8.955969	8.955969	0.000947	449	5.039939
1:	8.494575	5.974765	0.000779	677	4.049992
2:	8.462174	5.958943	0.010491	572	3.048980
3:	5.514705	5.514705	0.000396	3	0.000579
4:	5.557162	5.548496	0.000747	6	5.548496
5:	5.953767	5.953767	0.000410	2	0.008089
6:	0.031044	0.027696	0.000281	1	0.003067
7:	0.003993	0.003993	0.000000	0	0.000000

Real time issues: Scheduling: Find issues..

Cyclictest

“cyclictest measures the delta from when it's scheduled to wake up from when it actually does wake up”

Clark Williams, An Overview of Realtime Linux, Redhat Summit, June 18-20th, 2008.

https://rt.wiki.kernel.org/index.php/Cyclictest

Real time issues: Tools: latency_hist (demo)

Latency Hists (available in RT Patchset)
CONFIG_INTERRUPT_OFF_LATENCY
CONFIG_PREEMPT_OFF_LATENCY
CONFIG_WAKEUP_LATENCY

- Possible latencies:
  - Preemption Off historgram
  - IRQs Off historgram

- Effective latencies:
  - Histogram of wake up latency per CPU
  - Details of Task experiencing the latency

Real time issues: Tools: latency_hist

- Demo:
    - Example RT task: Cyclic test
    - Example kernel module: introduce Preempt Off latency
    - Results:
        - Cyclic test shows latency
        - Latency hists shows hists & maximum effective latency

Real time issues: Tools: latency_hist

Code in ‘trouble maker’ kernel module:

```
int x;

static int __init test_module_init(void)
{
        unsigned long j, i, loop1 = 100, loop2 = 10000;

        /* Introduces a preempt delay about about 50ms */

        preempt_disable();

        for (i = 0; i < loop1 * loop2; i++)
                ACCESS_ONCE(x) += 1;

        preempt_enable();

        return -1;
}
```

Real time issues: Tools: latency_hist

Running trouble maker module in a loop:

```
while [ 1 ]; do insmod ./preemptd.ko; done
```

Run cyclictest with priority 80:

```
./cyclictest -t1 -p 80 -n -i 10000 -l 10000
```

Real time issues: Tools: latency_hist

cyclictest gets victimized sooner or later:

```
root@raspberrypi:/home/pi# ./cyclictest -t1 -p 80 -n -i 10000 -l 10000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.18 0.12 0.06 1/125 1382

T: 0 ( 1075) P:80 I:10000 C:   5252 Min:     14 Act:   36 Avg:   36 Max:    2024
```

Real time issues: Tools: latency_hist

latency_hist histograms can show per-CPU latency histograms:

```
# cat /sys/kernel/debug/latency_hist/wakeup/CPU3
...
  583                  1
  718                  1
  772                  1
  807                  1
  850                  1
  853                  1
  861                  1
 1120                  1
 1470                  1
 1895                  1
 1994                  1
```

Real time issues: Tools: latency_hist

latency_hist shows details of max wakeup latency per-CPU:

```
# cat /sys/kernel/debug/latency_hist/wakeup/max_latency-CPU3
```

```
CPU 3 max latency info:
1075 80 1994 (0) cyclictest <- 1329 -21 insmod 1107.418695
```

Real time task migration considerations

Some experiments with migrate.c from Steven Rostedt's kernel shark article.

What we have:

- Set of RT tasks (2 * num of CPUs) in increasing priority (task 0 lowest)
- Set of locks (num of CPUs)

All tasks do the following in a loop:

- Acquire a lock
- Busy loop for duration inversely proportional to priority (ex, task X, busy loops for (8+1) - Xms for 8 tasks). Task 0, busy for 9 ms, Task 1 for 8ms …. Task 7 for 2ms)
- Release lock
- Sleep for duration directly proportional to priority (task 0 for 1ms, task 7 for 7ms)

Basically, higher the priority of a task, the less busy they are and the more they sleep

Real time task migration considerations

Regular tasks:

Task	vol	nonvol	migrated	iterations	loops
0:	183	394	188	29	2177107
1:	384	323	193	29	2120412
2:	377	344	260	29	1874924
3:	358	339	269	29	1803971
4:	354	317	249	29	1697549
5:	362	315	254	29	1534982
6:	348	303	277	29	1400623
7:	349	224	248	28	1231380
8:	370	191	220	28	1115534
9:	372	181	227	29	1010928
10:	380	203	210	29	920205
11:	402	108	184	29	803405
12:	385	117	196	28	654852
13:	410	75	178	28	509504
14:	423	29	160	29	405603
15:	431	20	173	29	271828
total:	5888	3483	3486	460	19532807

Real time task migration considerations

RT tasks:

Task	vol	nonvol	migrated	iterations	loops
0:	79	4840	3720	31	2118518
1:	416	1832	1716	31	1992225
2:	448	816	1079	32	1958756
3:	449	626	873	32	1813756
4:	453	607	806	32	1687723
5:	447	401	595	32	1542516
6:	454	291	429	32	1431262
7:	447	172	249	32	1294005
8:	456	69	110	32	1177279
9:	466	36	72	32	1028266
10:	458	29	52	32	905890
11:	459	11	31	32	781165
12:	468	5	36	32	650250
13:	459	2	14	32	528495
14:	466	3	19	32	391158
15:	476	2	25	32	263475
total:	6901	9742	9826	510	19564739

Total migrations go up with RT policy for high number of RT tasks contending for CPU

Real time task migration considerations

RT migration:

- Unlike the fair scheduler, the RT scheduler has an active migration strategy

- High prio wakeup on a lower-prio task CPU can push the lower prio to other CPU

- Scheduler also pulls RT tasks from other CPUs.

- In pursuit of responsiveness, the active strategy gives available CPU resource ASAP

- This can lower throughput due to cache misses or NUMA considerations but makes sure that task gets CPU as quickly as possible.

- When there’s a lot of contention, higher prio tasks migrate less at cost of lower prio tasks

Real time task migration considerations

Pinning lower priority RT tasks during contention can actually CPU-starve them

Experiment on RPI, base line: (mbusy is the number of memory operations done during busy loop)

Task	vol	nonvol	migrated	iterations	loops(K)	mbusy(K/s)	lock-wait(ms)	busy(ms)
0:	473	1631	1473	118	168	40.97	0	4120
1:	557	1541	1411	139	173	40.46	0	4281
2:	557	1125	1183	139	166	44.38	0	3749
3:	557	58	106	139	221	68.82	0	3219
4:	557	2	4	139	184	68.64	0	2684
5:	557	0	2	139	148	68.94	0	2147
6:	557	0	2	139	110	68.48	0	1611
7:	557	0	1	139	74	68.99	0	1076
total:	4372	4357	4182	1091	1			

Real time task migration considerations

Pinning lower priority RT tasks during contention can actual CPU-starve them

Experiment on RPI, with pinning of task 1 and 2 to unique CPUs:

Task	vol	nonvol	migrated	iterations	loops(K)	mbusy(K/s)	lock-wait(ms)	busy(ms)
0:	472	964	1028	117	149	36.71	0	4069
1:	556	559	2	138	134	31.49	0	4279
2:	561	563	1	139	138	36.88	0	3768
3:	563	1098	1097	139	170	52.91	0	3217
4:	564	493	1015	139	185	69.21	0	2684
5:	558	0	2	139	149	69.61	0	2147
6:	566	75	170	139	111	69.12	0	1611
7:	561	0	3	139	74	69.47	0	1075
total:	4401	3752	3318	1089	1			

Notice that the throughput (mbusy) of task 1 and 2 drop because of pinning

Real time task migration considerations

Conclusions:

- Assign priorities to RT tasks carefully, higher prio RT tasks migrate much lesser
- Consider performance issues introduced by continuous migrations of lower prio RT tasks
- Always measure and choose the best affinities and priorities for your particular system
- Remember that interrupt threads are also RT tasks and can also be pinned.

Real time issues: Other notes (temporary placeholder)

Kprobes for IRQ threads
Find all IRQ threads that didn’t sleep for a duration

Systrace data for rt migrations

OSQ locks for old kernels and !RT_PATCHSET

Vmalloc purge issue - done