Corporate Technology

SIEMENS

# System and Application Analysis with LTTng

## Project examples

- Serial input latency
- Sporadic delay in high prio application thread

*How LTTng was used in Siemens projects to solve problems*
*Gernot Hillier, CT SE 2*

SIEMENS

# Serial input latency

## Situation

- target hardware connected to serial port of „fast machine“ (Multicore, fast 64bit CPUs, ...)
- high prio application has to react on signal from serial port within 10 ms
- expected time for reading character from serial port: << 1ms

## Problem

- Unclear, undeterministic latency (several ms) from arrival of character to wakeup of application
- Possible reasons: locking, priority inversion, system load, ...?



 

# Serial input latency

## Method:

- **Trace simple testcase: bash shell prompt on serial port (does read(), write() in an endless loop to echo characters)**
- **Filter out events around IRQ arrival, then filter out right CPU**

## Trace output, part 1/2:

**kernel_arch_syscall_entry: 308.506794 (cpu_3), 3051, bash, SYSCALL { syscall_id = 0 [sys_read+0x0/0xaa], ip = 0x7f29c1040f40 }** => *application blocks on read()*

[...]

**kernel_irq_entry: 310.147709 (cpu_3), 0, swapper, IRQ { irq_id = 4, kernel_mode = 1, ip = 18446744071564210684 }** => *char arriving on serial line, IRQ 4 asserted*

**kernel_timer_set: 310.147722 (cpu_3), 0, swapper, IRQ { expires = 4294944731, function = 0xffffffff80245288** [...] **}** => *timer set, calling function* `delayed_work_timer_fn()`

**kernel_irq_exit: 310.147728 (cpu_3), 0, swapper, SYSCALL { handled = 1 }**

=> *now CPU does nothing (no event) for nearly 3 ms*

SIEMENS

# Serial input latency

## Trace output, part 2/2:

**kernel_irq_entry: 310.150507 (cpu_3), 0, swapper, IRQ { irq_id = 239, kernel_mode = 1, ip = 18446744071564210684 }** => *timer interrupt occurs (HZ = 250)*

[...]

**kernel_sched_schedule: 310.150549 (cpu_3), 14, events/3, SYSCALL { prev_pid = 0, next_pid = 14, prev_state = 0 }** => *workqueue thread for CPU 3 gets scheduled*

**kernel_sched_try_wakeup: 310.150562 (cpu_3), 14, events/3, SYSCALL { pid = 3051, state = 1 }** => *workqueue thread wakes up ...*

**kernel_sched_schedule: 310.150570 (cpu_3), 3051, bash, SYSCALL { prev_pid = 14, next_pid = 3051, prev_state = 1 }** => *... our target process*

**kernel_arch_syscall_exit: 310.150585 (cpu_3), 3051, bash, USER_MODE { ret = 1 }**

*=> read() returns, target application ...*

**kernel_arch_syscall_entry: 310.150600 (cpu_3), 3051, bash, SYSCALL { syscall_id = 1 [sys_write+0x0/0xaa], ip = 0x7f29c1040fc0 }** => *... can finally echo character*

# Serial input latency

## Further steps

- Code review of Linux serial driver, search for usage of work queues (serial8250_interrupt → serial8250_handle_port → receive_chars → tty_flip_buffer_push → schedule_delayed_work → queue_delayed_work → queue_delayed_work_on)
- Result: Linux serial code uses delayed work queues (delay: 1 jiffy) to handle incoming characters
- Reason: don't wake up userspace on each character
  => reduce overhead, increase throughput for „normal applications“

## Conclusion

- throughput optimization in Linux serial code conflicts with our use case
- Own, simple serial receive routine was implemented for critical path

SIEMENS

# Unclear delay in high prio application thread

## Situation

- multi-threaded application with a number of low-prio workers and high-prio thread with „soft realtime“ requirements
- large C++ application using rich middleware
- full code review (down to system call level) very time-consuming

## Problem

- unclear, sporadic delays at some code points
- Possible reasons: system load, locking, application problems, ...?

SIEMENS

# Unclear delay in high prio application thread

## Method:

- roughly identify problematic code points with application time stamping
- mark problematic code passages with invalid syscalls (syscall 600 at start of code point, 601 after code point if delay too high); LTTng userspace markers not ready
- Filter out events between syscalls, then filter out right CPU

## Trace output, first try (excerpt):

**kernel_arch_syscall_entry: 565.743 (cpu_5), 3309, PrioThread, { syscall_id = 600 }**

**kernel_arch_syscall_exit: 565.743 (cpu_5), 3309, PrioThread, { ret = -38 }**

**kernel_sched_schedule: 565.743 (cpu_5), 0, swapper, { prev_pid = 3309, next_pid = 0, prev_state = 2 }** => *kernel switches to idle task for unknown reason*

**kernel_arch_trap_entry: 565.774 (cpu_5), 3309, PrioThread, { trap_id = 14, ... }** => *process returned from page fault after 30 ms!*

**kernel_arch_syscall_entry: 565.774 (cpu_5), 3309, PrioThread, { syscall_id = 601 }**

SIEMENS

# Unclear delay in high prio application thread

## Method:

- **Review kernel code => trace marker in page fault is too late**
- **Add additional trace markers to arch/x86/mm/fault.c:do_page_fault**

## Trace output, second try (excerpt):

**kernel_arch_syscall_entry: 575.226 (cpu_5), 3309, PrioThread, { syscall_id = 600 }**

**kernel_arch_syscall_exit: 575.226 (cpu_5), 3309, PrioThread, { ret = -38 }**

**kernel_arch_page_fault_entry: 575.226 (cpu_5), 3309, PrioThread { ip = 0x7f1a3c073d9c }**

*=> theory confirmed, delay is caused by page fault*

**kernel_arch_page_fault_addr: 575.226 (cpu_5), 3309, PrioThread, { addr = 139735431535128, ip = 0x7f1a3c0 }** *=> additional trace point to get faulting address*

**kernel_sched_schedule: 575.226 (cpu_5), 3336, WorkThread, { prev_pid = 3309, next_pid = 3336 }** *=> This time, another low prio thread is runnable and scheduled*

**kernel_arch_trap_entry: 575.258 (cpu_5), 3309, PrioThread, { trap_id = 14, ... }**

**kernel_arch_syscall_entry: 575.258 (cpu_5), 3309, PrioThread, { syscall_id = 601 }**

## Unclear delay in high prio application thread

### Further steps

- decoding of page faulting address (with the help of /proc/<pid>/maps) => mmaped area on disk
- code review for access to mmaped disk area

### Conclusion

- commonly used application function used by high prio thread caused access to mmaped area on disk
- application code restructuring to get rid of this access