Real-time Linux Scheduling Latency

after S.Rostedt, D.B. Oliveira, D. Casini, R.Oliveira, T.Cucinotta

March 2021

1) Real-Time

Real-Time (RT) :

- is about timing behavior not performance
- deterministic/predictable scheduling

∼ opposite of “batch work” on a server (think of a build server where performance is about long-run rather than reactivity)

⇒ timing response guarantees/bounding = safety bound

⇒ predict the worst case

With Linux ? :

- understanding timing behavior of linux
- priority-base scheduling : high priority needs to be able to preempt low priority

⇒ faster in worst case scenarios but slower in the average scenarios (otherwise this would be the default kernel)

Applications :

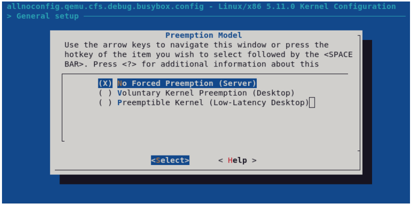
- robotics
- stock exchange
- music studio recording (no “glitches” with jack low-latency audio server)
- death or life devices ?
  - ⇒ NO ! Linux kernel too complex for formal methods ?

Two approaches with the Linux kernel :

- dual-kernel approach : RTAI, RTLinux, **Xenomai Cobalt**, Xenomai Mercure
  - ⇒ Linux becomes a task alongside high-priority RT tasks
  - ⇒ lots of work : support new architecture, implement specific tools/libraries (libc)
  - ⇒ bad scaling, (wo)man power problem
- in-kernel approach
  - ⇒ maximize preemptible code sections = allow scheduling almost everywhere
  - ⇒ take advantage of all the tools/optimizations/industrial support. . .
  - ⇒ Linux too big for $<$ 1$ chip ?
  - ⇒ try to keep up-to-date with mainline kernel
  - ⇒ 2021 : try to merge the patches in mainline kernel

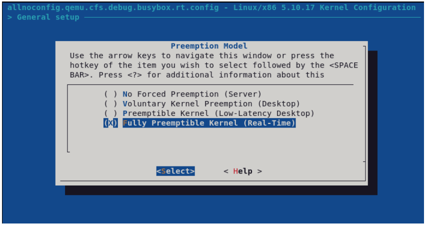
⇒ both Linux-approaches are ∼15/20 years old

Standard Linux preemption model configuration : **PREEMPT_NONE**, default = **PREEMPT_VOLUNTARY**, **PREEMPT** :



```
allnoconfig.qemu.cfs.debug.busybox.config - Linux/x86 5.11.0 Kernel Configuration
> General setup ──────────────────────────────────────────────────────────────────

                     Preemption Model
  Use the arrow keys to navigate this window or press the
  hotkey of the item you wish to select followed by the <SPACE
  BAR>. Press <?> for additional information about this

          (X) No Forced Preemption (Server)
          ( ) Voluntary Kernel Preemption (Desktop)
          ( ) Preemptible Kernel (Low-Latency Desktop)

                  <Select>    < Help >
```

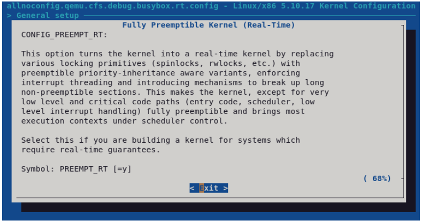
With patches (new **PREEMPT_RT**) :



```
allnoconfig.qemu.cfs.debug.busybox.rt.config - Linux/x86 5.10.17 Kernel Configuration
> General setup ──────────────────────────────────────────────────────────────────────

                     Preemption Model
  Use the arrow keys to navigate this window or press the
  hotkey of the item you wish to select followed by the <SPACE
  BAR>. Press <?> for additional information about this

          ( ) No Forced Preemption (Server)
          ( ) Voluntary Kernel Preemption (Desktop)
          ( ) Preemptible Kernel (Low-Latency Desktop)
          (X) Fully Preemptible Kernel (Real-Time)

                  <Select>    < Help >
```

Debian binary : linux-image-amd64 ⇝ linux-image-rt-amd64

Make preemption enabled almost everywhere :



```
allnoconfig.qemu.cfs.debug.busybox.rt.config - Linux/x86 5.10.17 Kernel Configuration
> General setup ─────────────────────────────────────────────────────────────────────
                      Fully Preemptible Kernel (Real-Time)
 CONFIG_PREEMPT_RT:

 This option turns the kernel into a real-time kernel by replacing
 various locking primitives (spinlocks, rwlocks, etc.) with
 preemptible priority-inheritance aware variants, enforcing
 interrupt threading and introducing mechanisms to break up long
 non-preemptible sections. This makes the kernel, except for very
 low level and critical code paths (entry code, scheduler, low
 level interrupt handling) fully preemptible and brings most
 execution contexts under scheduler control.

 Select this if you are building a kernel for systems which
 require real-time guarantees.

 Symbol: PREEMPT_RT [=y]
                                                                  ( 68%)
                               < Exit >
```

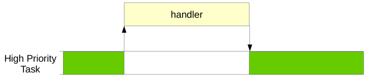
- spinlocks ⇝ raw_spinlock + sleeping spinlocks (= rt mutex)
- threaded interrupt handler
- priority inheritance to avoid priority inversion = when a high priority is blocked because of some task of lower priority
- others hacks already merged in mainline kernel

⇒ HUGE collection of patches

⇒ 80% already mainlined (timers, interrupt handlers,tracing infrastructure. . .)

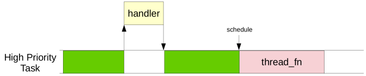
Normal interrupt preempts the task and executes the handler function :

```
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
        const char *name, void *dev)
```



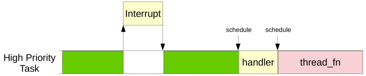
Threaded interrupt schedules the thread_fn function (*top/bottom half approach*) :

```
request_threaded_irq(unsigned int irq, irq_handler_t handler,
        irq_handler_t thread_fn,
        unsigned long flags, const char *name, void *dev)
```



Forced threaded interrupts just acknowledges the device ($< 1\ \mu$s) :

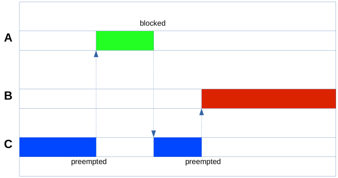
```
request_threaded_irq(unsigned int irq, irq_handler_t handler,
        irq_handler_t thread_fn,
        unsigned long flags, const char *name, void *dev)
```



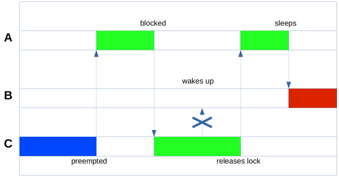
⇒ merged in mainline since 2009

⇒ all interrupts as threads `threadirqs` in mainline since 2011

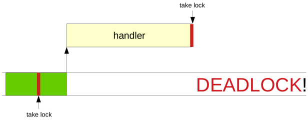
Priority inversion :



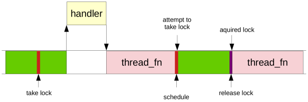
Priority inheritance :



Most of the spinlock are transformed in rt-mutexes. The others are now called raw_spinlocks.



There are in fact needed now with the threaded IRQs !



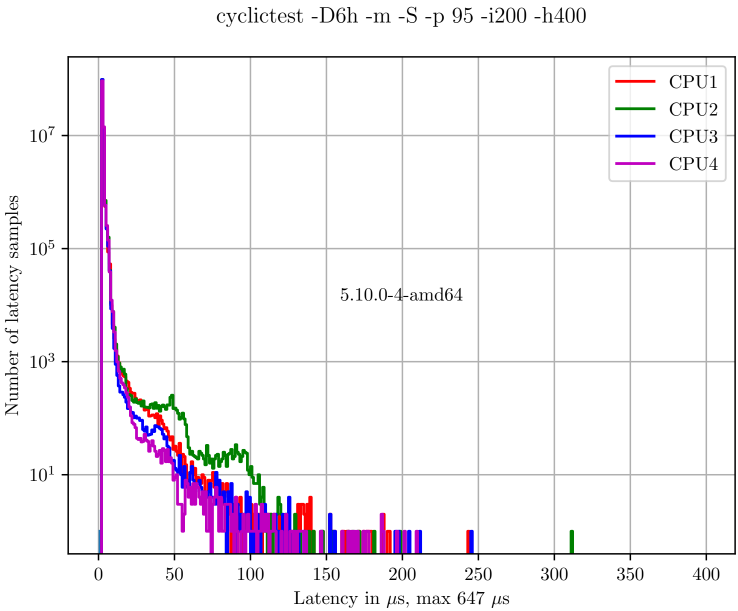
Usual way to tests real-time kernel behavior is to use the `rt-tests` suite :

- timer latency
- signal latency
- functionning of priority-inheritance mutexes
- ...

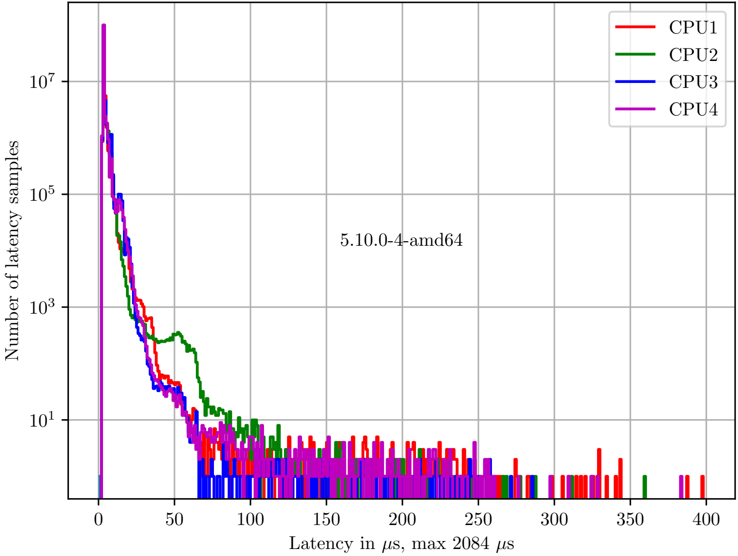
The main program is `cyclictest` :

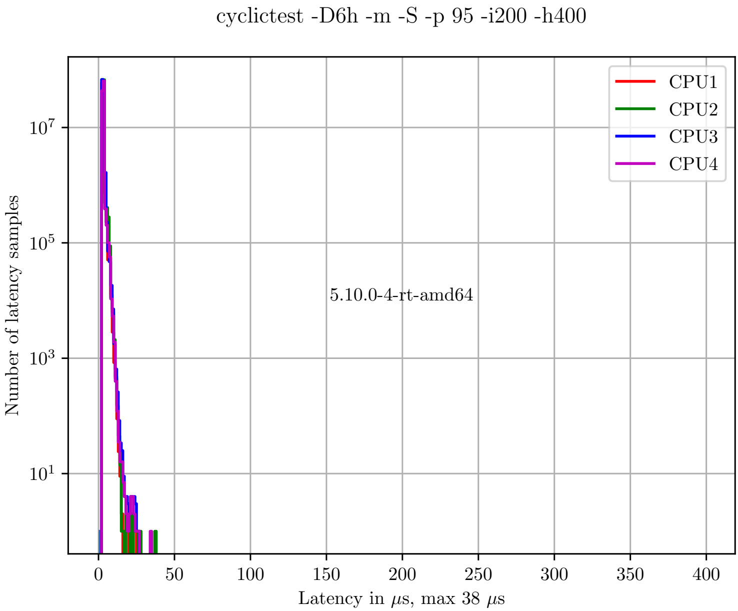
- `-p,--prio=PRIO` of first thread (default 80)
- `-i,--interval=INTV` of first thread (default 1000μs)
- `-l,--loops=LOOPS` of first thread (default 0=endless)
- `-D,--duration=TIME`
- `-t,--threads=NUM` (default 1,empty means #CPU)
- `-m,--mlockall` lock current and future memory alloc
- `-a,--affinity=PROC-SET`
- `-S,--smp` = `-t -a`
- `-h,--histogram=US` max latency tracked

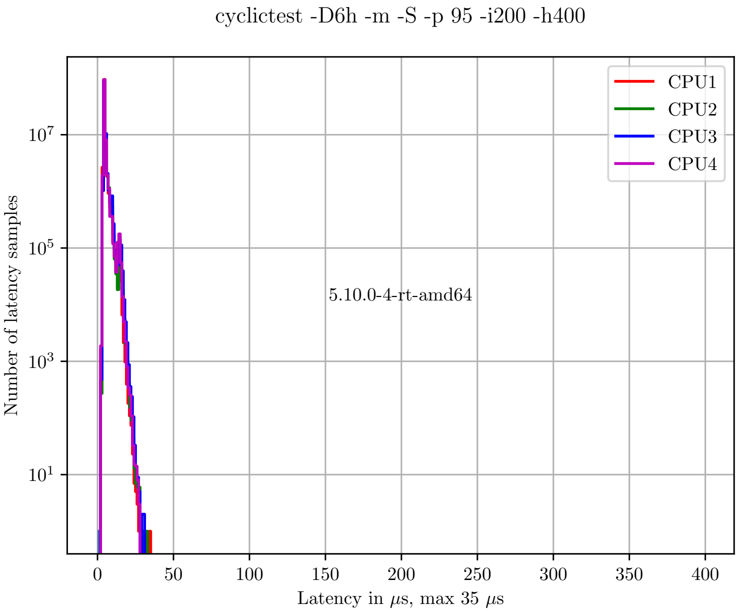
⇒ `cyclictest -D6h -m -S -p95 -i200 -h400`

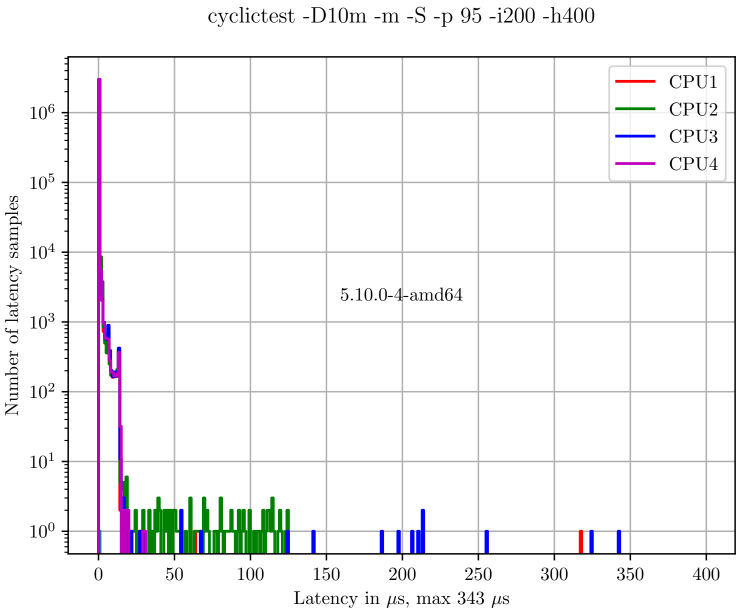

cyclictest -D6h -m -S -p 95 -i200 -h400
Number of latency samples
10^7
10^5
10^3
10^1
0
50
100
150
200
250
300
350
400
Latency in μs, max 647 μs
CPU1
CPU2
CPU3
CPU4
5.10.0-4-amd64

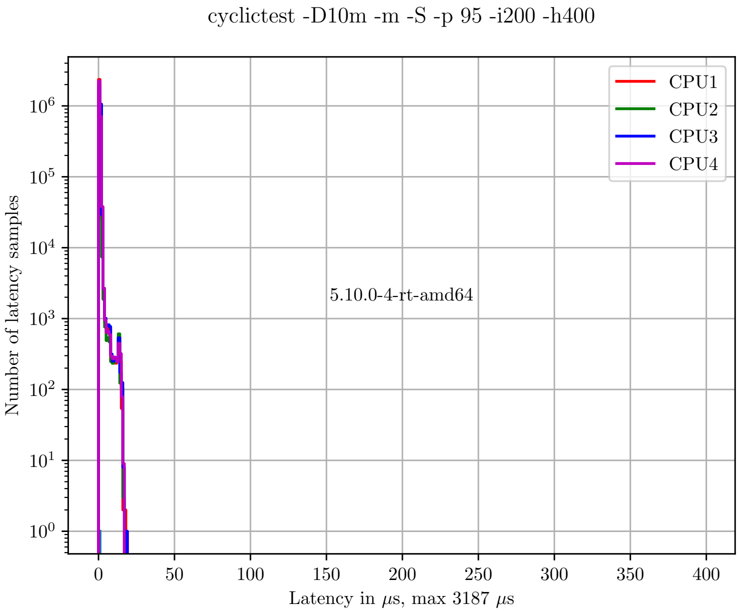
cyclictest -D6h -m -S -p 95 -i200 -h400




cyclictest -D6h -m -S -p 95 -i200 -h400
CPU1
CPU2
CPU3
CPU4
5.10.0-4-rt-amd64
10^7
10^5
10^3
10^1
Number of latency samples
0
50
100
150
200
250
300
350
400
Latency in μs, max 38 μs


cyclictest -D6h -m -S -p 95 -i200 -h400
Number of latency samples
10^7
10^5
10^3
10^1
0
50
100
150
200
250
300
350
400
Latency in μs, max 35 μs
CPU1
CPU2
CPU3
CPU4
5.10.0-4-rt-amd64


cyclictest -D10m -m -S -p 95 -i200 -h400
CPU1
CPU2
CPU3
CPU4
5.10.0-4-amd64
Number of latency samples
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0
50
100
150
200
250
300
350
400
Latency in μs, max 343 μs


cyclictest -D10m -m -S -p 95 -i200 -h400
CPU1
CPU2
CPU3
CPU4
5.10.0-4-rt-amd64
Number of latency samples
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0
50
100
150
200
250
300
350
400
Latency in μs, max 3187 μs

Oliveira, Casini, Oliveira & Cucinotta papers :

- model the IRQs, NMIs, thread synchronization mechanisms into an automata
- model stable between Linux versions (contrary to a model entirely generated from traces)
- able to find some linux code errors
- apply in to get a model of the Linux+PREEMPT_RT patches latency

⇒ not just the latency but root causes of the latency

⇒ $\sim$ Oliveira 2020 PhD thesis

- automata compiled as a kernel module

⇒ new tracing infrastructure to understand the root causes of this latency

⇒ alternative to `cyclictest` and kernel `ftrace`, user-space `trace-cmd` and graphic interface `kernelshark`

2) Model of IRQs, NMIs and thread synchronization mechanisms

Journal of Systems Architecture Volume 107, August 2020, 101729

A Thread Synchronization Model for the PREEMPT_RT Linux Kernel

Daniel B. de Oliveira[a,b,c], Rômulo S. de Oliveira[b], Tommaso Cucinotta[c]

[a] RHEL Platform/Real-time Team, Red Hat, Inc., Pisa, Italy.

[b] Department of Systems Automation, UFSC, Florianópolis, Brazil.

[c] RETIS Lab, Scuola Superiore Sant'Anna, Pisa, Italy.

Synchronization mechanisms between :

- IRQs (Interrupt ReQuests)
- NMIs (Non Maskable Interrupts) = cannot ignore interrupts = hw errors, parity/ECC errors, system
- thread scheduling, context switch, . . .
- locking : mutex, rwlocks, semaphores

Model with an automata $G = (X, E, F, \Gamma, x_0, X_m)$:

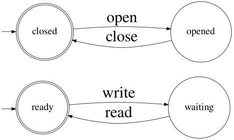
- X finite set of states
- E finite set of events
- $F : X \times E \to X$ transition function
- $\Gamma(x)$ set of events e such that $F(x, e)$ is defined in state x
- x_0 initial state
- X_m set of final states

On all the interesting Linux mechanisms ?

Modular approach relying on automata theory :

- generators, specifications, parallel composition
- computed automatically thanks to a dedicated software `Supremica IDE`

Two generators or sub-automatas :

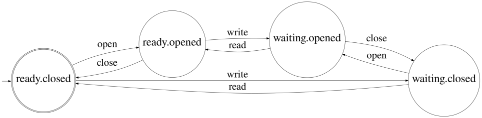


For a model of Linux synchronization mechanisms, this will be the minimal operations in kernel synchronization.

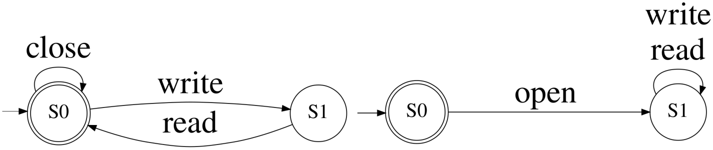
Parallel composition or synchronous composition of two generators $G1 = (X1, E1, f1, \Gamma 1, x01, Xm1)$ $G2 = (X2, E2, f2, \Gamma 2, x02, Xm2)$ introduces the notion of *private* events and *common* events

$$G_1 \parallel G_2 := Ac(X_1 \times X_2, E_1 \cup E_2, f_{1\|2}, \Gamma_{1\|2}, (x_{01}, x_{02}), X_{m1} \times X_{m2})$$

$$\begin{aligned}
f_{1\|2}((x_1, x_2), e) \quad &:= \quad \begin{cases} (f_1(x_1, e), f_2(x_2, e)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, e), x_2) & \text{if } e \in \Gamma_1(x_1) \backslash E_2 \\ (x_1, f_2(x_2, e) & \text{if } e \in \Gamma_2(x_2) \backslash E_1 \\ \text{undefined} & \text{otherwise} \end{cases} \\
\Gamma_{1\|2}((x_1, x_2)) \quad &:= \quad \Gamma_1(x_1) \cup \Gamma_2(x_2)
\end{aligned}$$



Specifications between generators are also implemented as an automata.



For a model of Linux synchronization mechanisms, this will be for example the necessary conditions to call the scheduler.

And `Supremica IDE` computes the final automaton

- verifies there is no dead-lock
- final automata deterministic (only 1 transition to next step)

⇒ and Linux PREEMPT_RT too if the model is correct

Is the model really modeling the Linux kernel ?

Kernel event	Automaton event	Description
hw_local_irq_disable	preemptirq:irq_disable	Begin IRQ handler
hw_local_irq_enable	preemptirq:irq_enable	Return IRQ handler
local_irq_disable	preemptirq:irq_disable	Mask IRQs
local_irq_enable	preemptirq:local_irq_enable	Unmask IRQs
nmi_entry	**irq_vectors:nmi**	Begin NMI handler
nmi_exit	**irq_vectors:nmi**	Return NMI Handler

Kernel event	Automaton event	Description
preempt_disable	preemptirq:preempt_disable	Disable preemption
preempt_enable	preemptirq:preempt_enable	Enable preemption
preempt_disable_sched	preemptirq:preempt_disable	Disable preemption to call the scheduler
preempt_enable_sched	preemptirq:preempt_enable	Enables preemption returning from the scheduler
schedule_entry	**sched:sched_entry**	Begin of the scheduler
schedule_exit	**sched:sched_exit**	Return of the scheduler
sched_need_resched	**sched:set_need_resched**	Set need resched
sched_waking	sched:sched_waking	Activation of a thread
sched_set_state_runnable	**sched:sched_set_state**	Thread is runnable
sched_set_state_sleepable	**sched:sched_set_state**	Thread can go to sleepable
sched_switch_in	sched:sched_switch	Switch in of the thread under analysis
sched_switch_suspend	sched:sched_switch	Switch out due to a suspension of the thread under analysis
sched_switch_preempt	sched:sched_switch	Switch out due to a preemption of the thread under analysis
sched_switch_blocking	sched:sched_switch	Switch out due to a blocking of the thread under analysis
sched_switch_in_o	sched:sched_switch	Switch in of another thread
sched_switch_out_o	sched:sched_switch	Switch out of another thread

Kernel event	Automaton event	Description
mutex_lock	**lock:rt_mutex_lock**	Requested a RT Mutex
mutex_blocked	**lock:rt_mutex_block**	Blocked in a RT Mutex
mutex_acquired	**lock:rt_mutex_acquired**	Acquired a RT Mutex
mutex_abandon	**lock:rt_mutex_abandon**	Abandoned the request of a RT Mutex
write_lock	**lock:rwlock_lock**	Requested a R/W Lock or Sem as writer
write_blocked	**lock:rwlock_block**	Blocked in a R/W Lock or Sem as writer
write_acquired	**lock:rwlock_acquired**	Acquired a R/W Lock or Sem as writer
write_abandon	**lock:rwlock_abandon**	Abandoned a R/W Lock or Sem as writer
read_lock	**lock:rwlock_lock**	Requested a R/W Lock or Sem as reader
read_blocked	**lock:rwlock_block**	Blocked in a R/W Lock or Sem as reader
read_acquired	**lock:rwlock_acquired**	Acquired a R/W Lock or Sem as reader
read_abandon	**lock:rwlock_abandon**	Abandon a R/W Lock or Sem as reader

Name	**States**	**Events**	**Transitions**
G01 Sleepable or runnable	2	3	3
G02 Context switch	2	4	4
G03 Context switch other thread	2	2	2
G04 Scheduling context	2	2	2
G05 Need resched	1	1	1
G06 Preempt disable	3	4	4
G07 IRQ Masking	2	2	2
G08 IRQ handling	2	2	2
G09 NMI	2	2	2
G10 Mutex	3	4	6
G11 Write lock	3	4	6
G12 Read lock	3	4	6
S01 Sched in after wakeup	2	5	6
S02 Resched and wakeup sufficency	3	10	18
S03 Scheduler with preempt disable	2	4	4
S04 Scheduler doesn't enable preemption	2	6	6
S05 Scheduler with interrupt enabled	2	4	4
S06 Switch out then in	2	20	20
S07 Switch with preempt/irq disabled	3	10	14
S08 Switch while scheduling	2	8	8
S09 Schedule always switch	3	6	6
S10 Preempt disable to sched	2	3	4
S11 No wakeup right before switch	3	5	8
S12 IRQ context disable events	2	27	27
S13 NMI blocks all events	2	34	34
S14 Set sleepable while running	2	6	6
S15 Don't set runnable when scheduling	2	4	4
S16 Scheduling context operations	2	3	3
S17 IRQ disabled	3	4	4
S18 Schedule necessary and sufficient	8	9	27
S19 Need resched forces scheduling	7	25	53
S20 Lock while running	2	16	16
S21 Lock while preemptive	2	16	16
S22 Lock while interruptible	2	16	16
S23 No suspension in lock algorithms	3	10	19
S24 Sched blocking if blocks	3	10	20
S25 Need resched blocks lock ops	2	15	17
S26 Lock either read or write	3	6	6
S27 Mutex doesn't use rw lock	2	11	11
S28 RW lock does not sched unless block	4	11	22
S29 Mutex does not sched unless block	4	7	16
S30 Disable IRQ in sched implies switch	5	6	10
S31 Need resched preempts unless sched	3	5	12
S32 Does not suspend in mutex	3	5	11
S33 Does not suspend in rw lock	3	8	16
Model	9017	34	20103

The PREEMPT RT task model has:

- 12 generators, 33 specifications, 9017 states, 23103 transitions
- deterministic automata

3) Model the latency bound

Proceedings 32nd Euromicro Conference on Real-time Systems (ECRTS 2020)

Demystifying the Real-Time Linux Scheduling Latency

Daniel Bristot de Oliveira
Red Hat, Italy
bristot@redhat.com

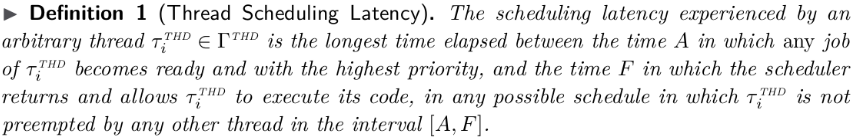
Daniel Casini
Scuola Superiore Sant'Anna, Italy
daniel.casini@santannapisa.it

Rômulo Silva de Oliveira
Universidade Federal de Santa Catarina, Brazil
romulo.deoliveira@ufsc.br

Tommaso Cucinotta
Scuola Superiore Sant'Anna, Italy
tommaso.cucinotta@santannapisa.it

Idea : taking a subset of the model (9/12 generators, 14/33 specifications) and perform a case analysis to model the latency

Thread scheduling latency is actual start F (after context switch) - expected activation of highest priority A :



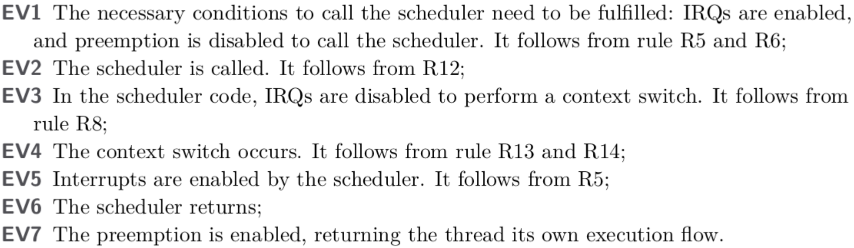
▶ **Definition 1** (Thread Scheduling Latency). *The scheduling latency experienced by an arbitrary thread* $\tau_i^{THD} \in \Gamma^{THD}$ *is the longest time elapsed between the time* A *in which* any *job of* τ_i^{THD} *becomes ready and with the highest priority, and the time* F *in which the scheduler returns and allows* τ_i^{THD} *to execute its code, in any possible schedule in which* τ_i^{THD} *is not preempted by any other thread in the interval* $[A, F]$.

Model divided in two parts :

- blocking/scheduling
- interference from IRQ/NMI

$$L = L^{\mathrm{IF}} + I^{\mathrm{NMI}}(L) + I^{\mathrm{IRQ}}(L).$$

$I^{NMI}(L)$ and $I^{IRQ}(L)$ can be estimated from experiments and tests
L^{IF} is the context switch and priority inversion blocking $\Rightarrow$ try to find a bound



EV1 The necessary conditions to call the scheduler need to be fulfilled: IRQs are enabled, and preemption is disabled to call the scheduler. It follows from rule R5 and R6;

EV2 The scheduler is called. It follows from R12;

EV3 In the scheduler code, IRQs are disabled to perform a context switch. It follows from rule R8;

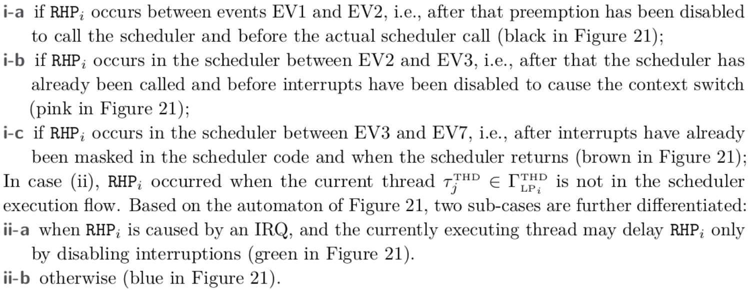
EV4 The context switch occurs. It follows from rule R13 and R14;

EV5 Interrupts are enabled by the scheduler. It follows from R5;

EV6 The scheduler returns;

EV7 The preemption is enabled, returning the thread its own execution flow.

Mutually case exclusive analysis :



i-a if $\mathtt{RHP}_i$ occurs between events EV1 and EV2, i.e., after that preemption has been disabled to call the scheduler and before the actual scheduler call (black in Figure 21);

i-b if $\mathtt{RHP}_i$ occurs in the scheduler between EV2 and EV3, i.e., after that the scheduler has already been called and before interrupts have been disabled to cause the context switch (pink in Figure 21);

i-c if $\mathtt{RHP}_i$ occurs in the scheduler between EV3 and EV7, i.e., after interrupts have already been masked in the scheduler code and when the scheduler returns (brown in Figure 21);

In case (ii), $\mathtt{RHP}_i$ occurred when the current thread $\tau_j^{\mathrm{THD}} \in \Gamma_{\mathrm{LP}_i}^{\mathrm{THD}}$ is not in the scheduler execution flow. Based on the automaton of Figure 21, two sub-cases are further differentiated:

ii-a when $\mathtt{RHP}_i$ is caused by an IRQ, and the currently executing thread may delay $\mathtt{RHP}_i$ only by disabling interruptions (green in Figure 21).

ii-b otherwise (blue in Figure 21).

Param.	Length of the longest interval
D_{PSD}	in which preemptions are disabled to schedule.
D_{PAIE}	in which the system is in state `pe_ie` of Figure 21.
D_{POID}	in which the preemption is disabled to postpone the scheduler or IRQs are disabled.
D_{ST}	between two consecutive occurrences of EV3 and EV7.

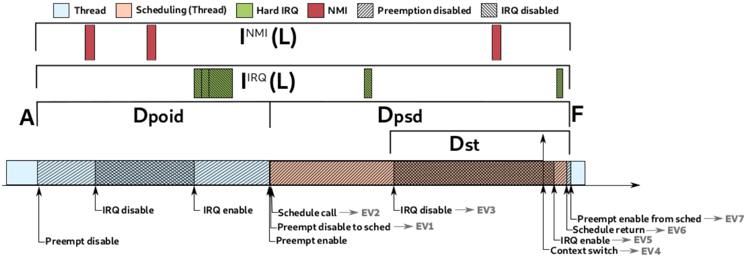


Figure 22 Reference timeline.

$$L^{IF} \leq max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD},$$

$$L = max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD} + I^{NMI}(L) + I^{IRQ}(L)$$

4) Conclusion

PREEMPT_RT patches :

- several advantages on other Linux real-time approches
- soon in main git repository (2021?)
- some optimizations already merged

Model with the automata approach :

- 12 generators, 33 specifications

⇒ model tractable

⇒ model stable between versions

⇒ can help to find bugs

- claims that Linux+patches is deterministic and the scheduling latency is bounded

Details not covered in this presentation that are worth to look at :

- extension of the `perf` tool and automata compiled insided a kernel module
- comparison with `cyclictest` and `ftrace`