# Real-time Linux at LPC2021 : stalld vs NOHZ_FULL

September 2021

1. Real-Time, preemption and latencies in Linux
2. Linux scheduler
3. RT task + non-RT tasks

## 1 ) Real-Time, preemption and latencies in Linux

Real-Time (RT) :

$\Rightarrow$ timing response guarantees, bound on worst case, ...

Levels of preemption in the Linux kernel :

- latency-throughput tradeoff : maximize throughput != minimize scheduling latencies

$\Rightarrow$ faster in worst case scenarios but slower in the average scenarios

- PREEMPT_RT > PREEMPT > PREEMPT_VOLUNTARY (debian default) > PREEMPT_NONE

Linux PREEMPT_RT patches :

- spinlocks $\rightsquigarrow$ raw_spinlock + sleeping spinlocks (= rt mutex)
- threaded interrupt handler
- priority inheritance to avoid priority inversion
- ...

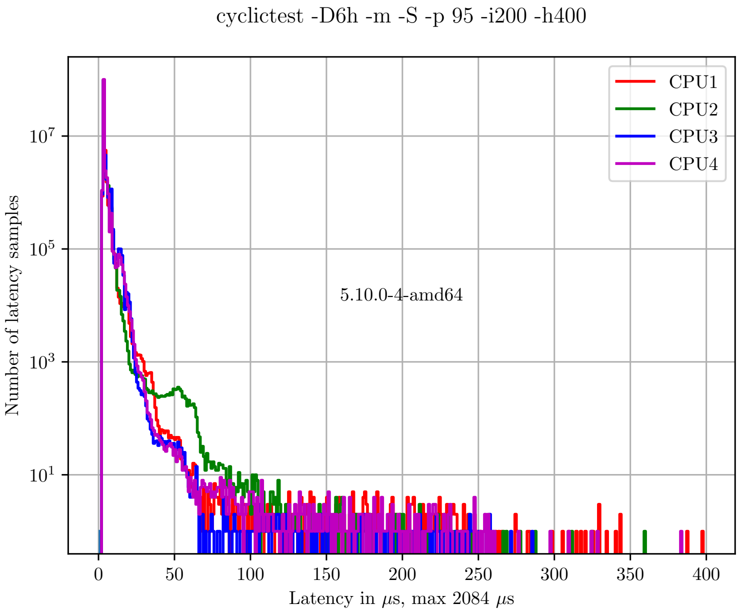
$\Rightarrow$ more work for the scheduler

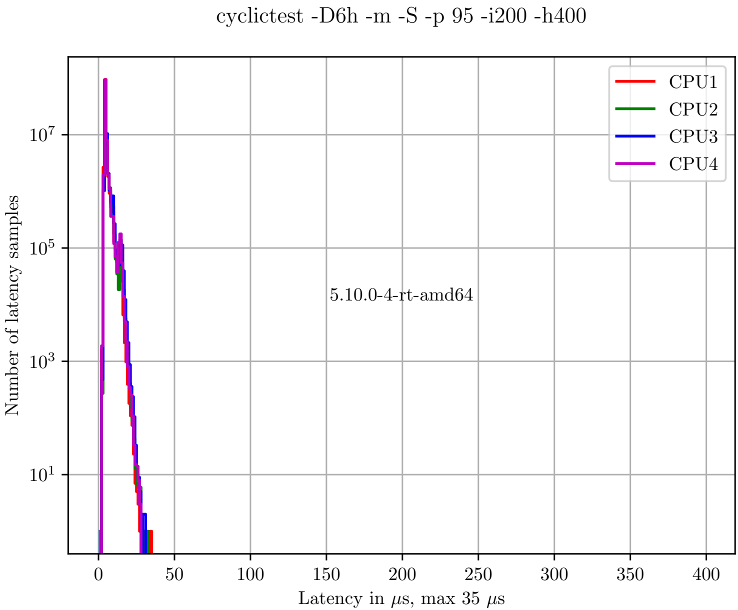
$\Rightarrow$ almost merged ! after > 15 years of work

Latency :

- $:= T_{\text{actual start after context switch}} - T_{\text{expected activation}}$ of highest priority task
- = function of
  - $\Rightarrow$ IRQs (Interrupt ReQuests)
  - $\Rightarrow$ NMIs (Non Maskable Interrupts)
  - $\Rightarrow$ thread scheduling and locking mechanisms

So if more sections of code (as IRQ handlers) have preemption enabled and get scheduled, priority task can start earlier and latencies decrease...


cyclictest -D6h -m -S -p 95 -i200 -h400
CPU1
CPU2
CPU3
CPU4
5.10.0-4-amd64
Number of latency samples
$10^7$
$10^5$
$10^3$
$10^1$
0
50
100
150
200
250
300
350
400
Latency in $\mu$s, max 2084 $\mu$s


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

# 2 ) Linux scheduler

Linux multi-core scheduler =
distributed mono-core + load-balancing :

- scheduling policies within scheduling classes
- scheduling with higher priority first
- tasks can migrate between CPUs, classes, policies
- runqueue per core :

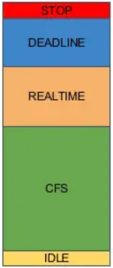
⇒ Stop : no policy
stop_machine, migration, RCU, ftrace ...

⇒ Deadline :
SCHED_DEADLINE

⇒ Realtime : prio $\in$ [0,99]
SCHED_FIFO SCHED_RR

⇒ Cfs : prio = 120 + nice $\in$ [100,139] $\rightsquigarrow$
weight $\rightsquigarrow$ vruntime
SCHED_NORMAL SCHED_BATCH
SCHED_IDLE ( >139)

⇒ Idle : no policy
swapper, low-power state



3 ) RT task + non-RT tasks

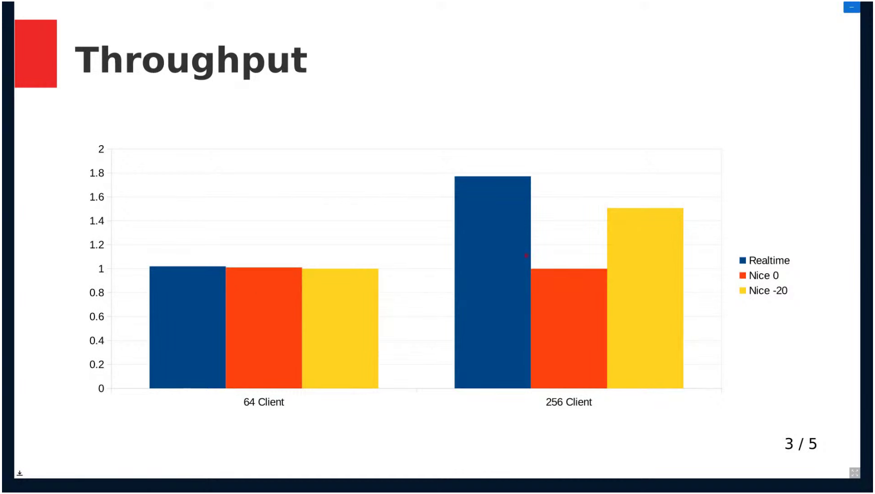
Old-fashion static deployment :

1. Linux parameters in Grub config (reboot)
   `BOOT_IMAGE=/boot/vmlinuz-5.15.0-rc2+ root=UUID=..`
   `ro isolcpus=3 quiet`
2. `sched_setaffinity` system call
   - `taskset -c 3 chrt -f 78 ./my-critical-RT-app`
   - `numactl ...`

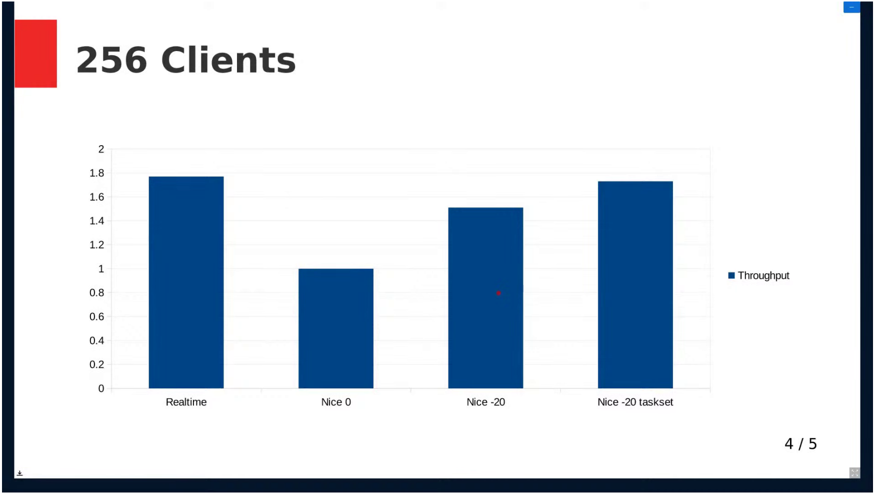
Modern days dynamic deployment : Cgroups (Docker, Kubernetes, ) :

- `echo 0 > cpuset.sched_load_balance`
- `cpuset.cpus/cpu_exclusive`

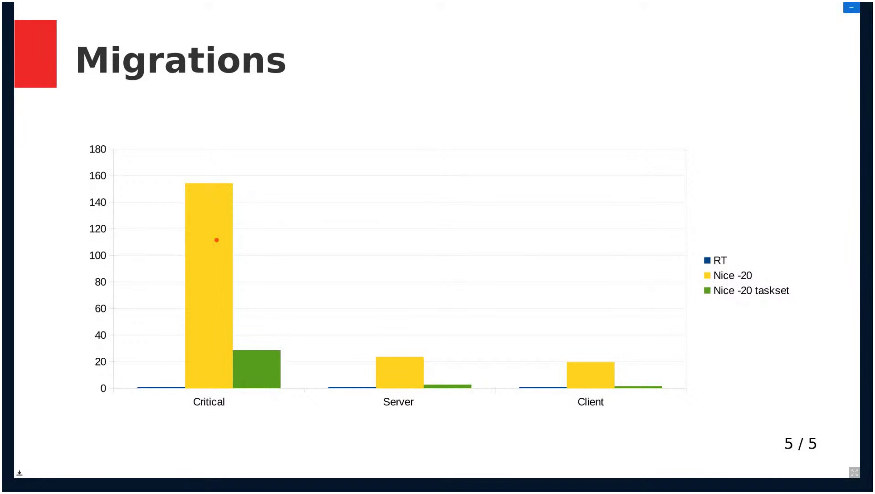
Performance when RT + non-RT ?




How to get performance back ?




How to get performance back ?




taskset ⇒ almost Realtime policy performance

OS noise that can cause a RT-task to miss its timing deadline :

- IRQs, SMIs
- drivers
- resource contention

⇒ SCHED_FIFO/RR RT-task . . . but

⇒ starvation of per-cpu kworker on the isolated cpus contending with the RT-task

⇒ RT-throttling not sufficient for other RT-tasks of lower priority

```
echo 2000000 > /proc/sys/kernel/sched_rt_period_us
echo 1000000 > /proc/sys/kernel/sched_rt_runtime_us
```

stalld : `https://github.com/bristot/stalld`

1. detecting starving threads
2. starting on housekeeping cpu a pthread for each isolated cpus
3. boosting temporarly thanks to SCHED_DEADLINE (or SCHED_FIFO)
4. 10 $\mu$s every s to give time to starving thread

3GPP specification for 5G : Radio Tower $\leftrightarrow$ Data-center maximum delay

$\Rightarrow$ Tx + processing + ack $< 3\mu s \Rightarrow$ `cyclictest` $< 10\ \mu s$

Telco people using user-space DPDK polling-mode NIC drivers

$\Rightarrow$ SCHED_FIFO prio=90 on isolated cpus

and basic services

$\Rightarrow$ SCHED_NORMAL on "housekeeping" cpus (sshd, dockerd, ...)

Linux kernel starts both non-RT and RT kthreads on every CPUs :

- SCHED_FIFO prio = 1 on isolated CPUs get starved $\Rightarrow$ cascading lockups

Limitations of stalld :

- scalability : pthread for every isolated cpu potentially starved
- running on housekeeping CPUs competing with the housekeeping tasks can get starved or can cause starving

$\Rightarrow$ hard to use

Sharan Turlapati (VMware) Srivatsa Bhat (MIT) come with an in-kernel solution :

- per-cpu starvation monitor list
- hrtimer for `boost_duration_time` and `starvation_duration_time` (user configuration)
- boost or deboost `sched_setattr` in hardirq context

Questions and reactions 2020-2021 :

- is the user-space stalld a debugging tool ?
- is the kernel stall monitor a ugly hack ?
- is adding priority to a process, a user-space decision ?
- single user-space thread is less overhead than kernel per-cpu solution anyway ?

Long-term solution instead of a workaround :

⇒ fix NOHZ_FULL isolation mode in the subsystems

*'How many man-years have been spent on developping stalld and stalld-ng instead of looking at the underlying problems and fix that ? I mean it's not rocket science. Most of the pain points are knowed. There are patches actually floating around, they are shitty patches but they could be polished up. So instead we waist time on things that are completely bonkers.'* Thomas Gleixner