

# Leveraging the Linux CPU Scheduler

## Leveraging the Linux CPU Scheduler to write real-time MT apps

Kaiwan N Billimoria  
***kaiwan*TECH**

LinkedIn public profile

My Tech Blog [please do follow!]

Corporate Training: <https://bit.ly/ktdcorp>

My GitHub page [please do star the repos you like!]

GitHub repo for this presentation

# Leveraging the Linux CPU Scheduler

**\$ whoami**

- I've been in the software industry from late 1991
- 'Discovered' Linux around '96-'97
- Been glued to it ever since!
- Self-taught: Linux kernel OS/drivers/embedded/debugging, along the journey
- Contributed to opensource as well as closed-source projects; Linux kernel, a very small bit...
- My GitHub repos: <https://github.com/kaiwan>



# Leveraging the Linux CPU Scheduler

## \$ whoami

- **Author of four books on Linux** (all published by Packt Publishing, Birmingham, England)
  - **Linux Kernel Debugging**, Aug 2022
  - **Linux Kernel Programming**, Mar 2021
  - **Linux Kernel Programming, Part 2 (Char Drivers)**, Mar 2021
  - **Hands-On System Programming with Linux**, Oct 2018
- ***My Amazon author page***

# Leveraging the Linux CPU Scheduler

## *Agenda*

- What does real-time actually mean
- The Linux process state machine
- How Linux schedules processes and threads
- The POSIX Scheduling Policies and what they mean
- Setting CPU scheduling policy and priority on an application thread
  - Demo MT app
- Converting vanilla Linux to an RTOS



# Leveraging the Linux CPU Scheduler

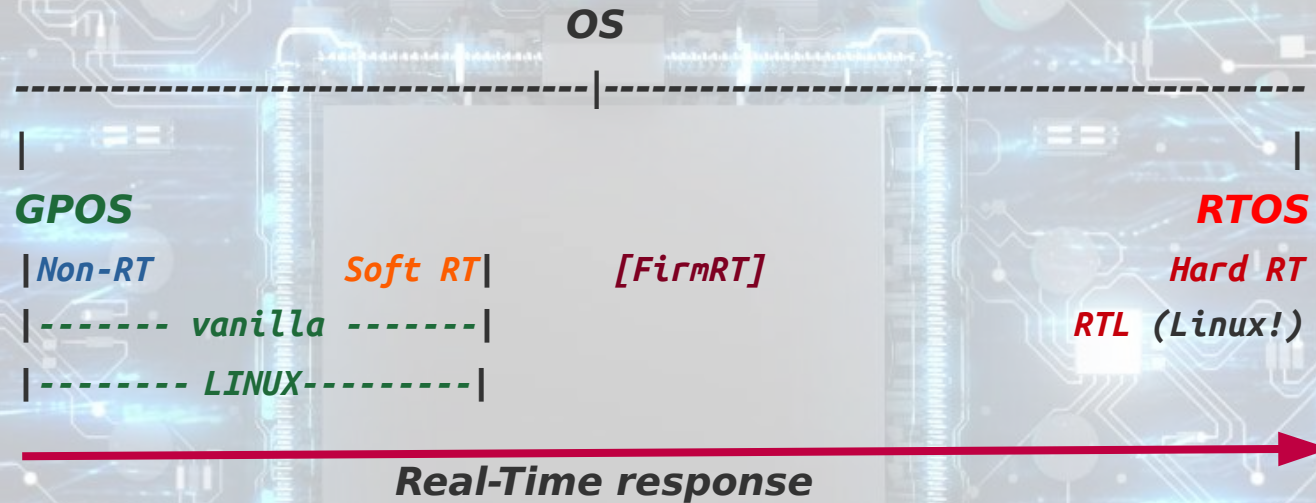
## What does real-time actually mean

- From the Wikipedia page on Real-time computing: ***“Real-time computing (RTC) is the computer science term for hardware and software systems subject to a “real-time constraint”, for example from event to system response. Real-time programs must **guarantee response within specified time constraints**, often referred to as “deadlines”.**”*
- **Deterministic response**
  - *Predictable with a guaranteed worst-case response time no matter the load on the system*
  - **Jitter** - *variance in response time; should be small and consistent; should be very low in a well designed RTOS + RT apps*
- *Real-time does not necessarily mean real fast*
- *While max system latencies can dramatically reduce, overall throughput can suffer*
- *RT systems and RTOS's (Real-Time Operating Systems) tend to be characterized by  $O(1)$  algorithms*

# Leveraging the Linux CPU Scheduler

What does real-time actually mean

- **Soft, firm and hard real-time**



- **GPOS : General-Purpose-OS (non-RT)**
  - (Vanilla/regular) Linux, Unix, Windows, MacOS
- **RTOS : Real-Time-OS**
  - FreeRTOS, Nucleus, RTL (Real-Time Linux), and many more...



# Leveraging the Linux CPU Scheduler

What does real-time actually mean

- **Non-RT, Soft, firm and hard real-time**
- **GPOS Non-RT :**
  - Zero determinism, any amount of jitter (it doesn't really matter!)
  - Absolutely no guarantee on how long it will take to complete a given task; no concept of a sacrosanct deadline at all
  - Most personal / business / enterprise software domains, Web apps
- **GPOS 'Soft' RT :**
  - best-effort; strives to achieve the given deadline BUT no guarantees!
  - Some determinism, some jitter
  - Small impact when deadline not met
  - Good examples are consumer electronics devices (smartphones, media players, etc)
- **[Firm RT : between Soft and Hard RT]**
- **RTOS / 'Hard' RT:**
  - deterministic, virtually no jitter (or very low)
  - Deadline MUST be set 100% of the time, else results in catastrophic failure, even financial and/or loss to human life
  - Examples - many kinds of transport, ATC systems, stock exchanges, pacemakers, factory floor robots, etc

# Leveraging the Linux CPU Scheduler

What does real-time actually mean

- *Non-RT, Soft, firm and hard real-time*

- *GPOS Non-RT :*

- Zero determinism, no concept of jitter (it doesn't really matter!)
- Absolutely no guarantee on how long it will take to complete a given task; no concept of a sacrosanct deadline at all

**REMEMBER / Key Point**

- The vanilla / mainline Linux OS is always only

- **SOFT** Real-Time capable, not 'hard RT'

- Some determinism, some jitter

- Small impact when deadline not met

- Good examples are consumer electronics devices (smartphones, media player, etc)

- *[Firm RT : between Soft and Hard RT]*

- We discuss how to 'convert' regular (vanilla) Linux to a hard real-time RTOS later in this presentation

- deterministic, virtually no jitter (or very low)

- Deadline is sacrosanct, if not met, the time, else results in catastrophic failure, even financial and/or loss to human life

- Examples - many kinds of transport, ATC systems, stock exchanges, pacemakers, factory floor robots, etc



# Leveraging the Linux CPU Scheduler

## The Linux process state machine

CPU Scheduling on Linux

Chapter 17

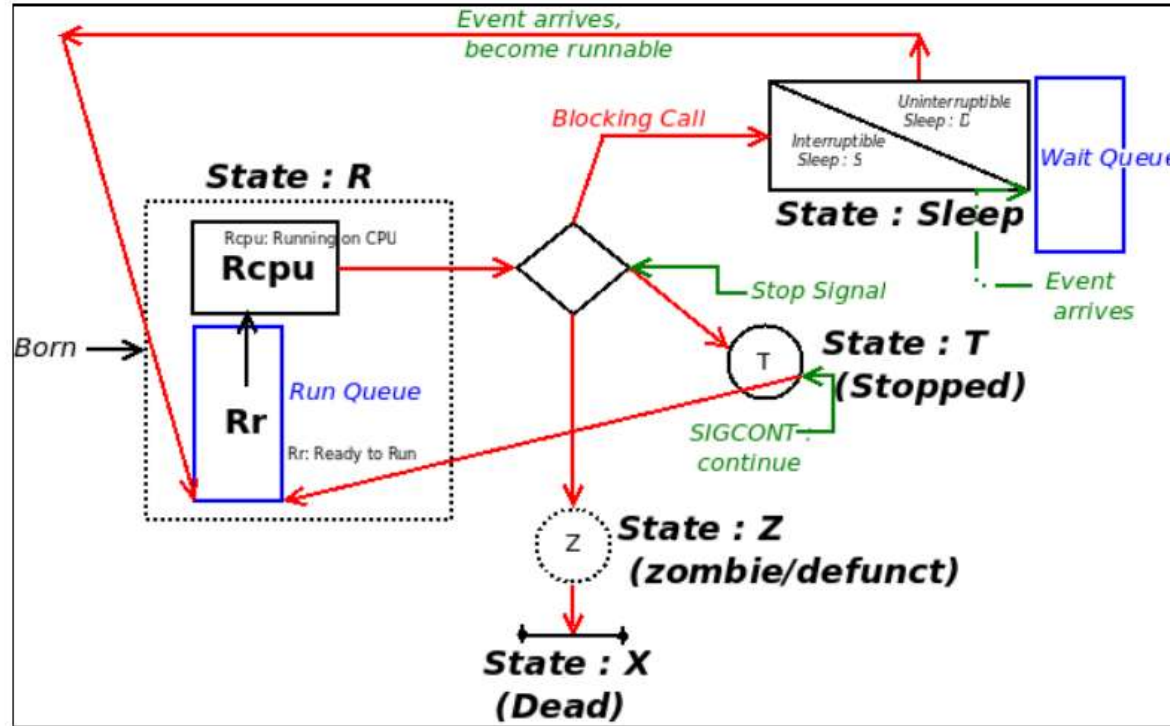


Figure 1: Linux state machine

From my *Hands-On System Programming with Linux* book

# Leveraging the Linux CPU Scheduler

## How Linux schedules processes / threads

- Key point
  - In CS, there is the notion of the **Kernel Schedulable Entity (KSE)**
  - The KSE on Linux is a *thread* not a process



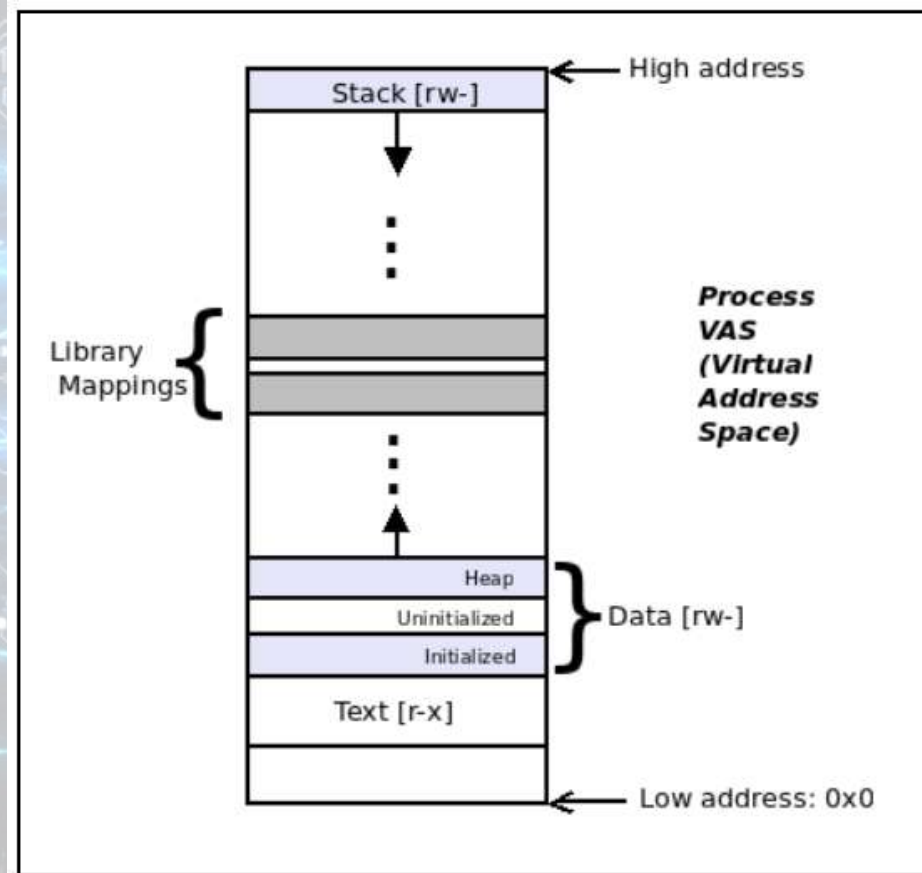
# Leveraging the Linux CPU Scheduler

## How Linux schedules processes / threads

- Key points
  - A **process** is an instance of a program in execution
  - UNIX / Linux philosophy- *Everything is a process; if not, it's a file*
  - The Linux OS is architected such that
    - every process has it's own sandbox – the **process VAS** (Virtual Address Space; diagram on following slide)
    - a **thread** is an execution path within a process
    - a process must have at least one thread – `main()`! (aka T0)
    - It can have >1 thread; if so, it's termed a **multithreaded (MT)** process

# Leveraging the Linux CPU Scheduler

How Linux schedules processes / threads



From my *Hands-On System Programming with Linux* book



# Leveraging the Linux CPU Scheduler

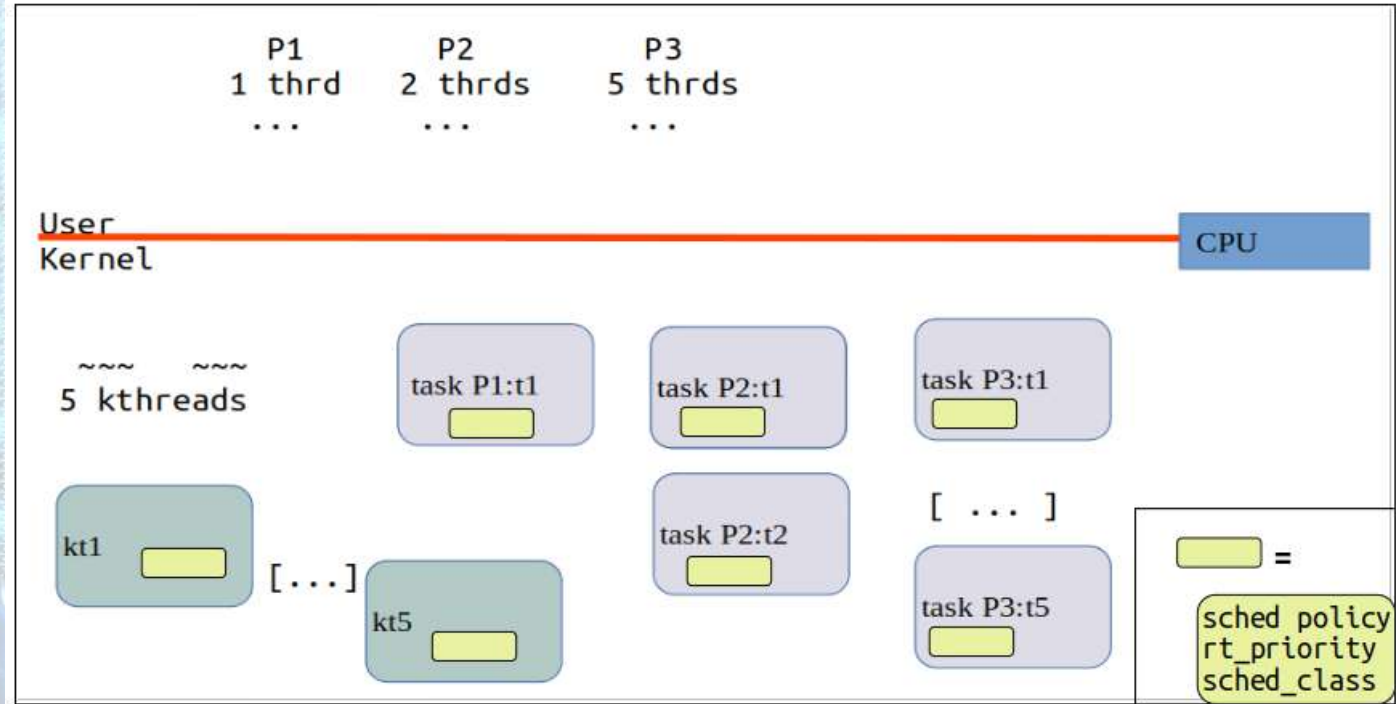
## How Linux schedules processes / threads

- Key point
  - The Linux OS is architected such that
    - every process has it's own sandbox – the **process VAS** (Virtual Address Space; diagram on following slide)
    - a **thread** is an execution path within a process
    - a process must have at least one thread – main()! (aka T0)
    - It can have >1 thread; if so, it's termed as a **multithreaded (MT)** process
    - Every thread alive has it's own (kernel) task structure, user and kernel-mode stacks
    - (see diagram on next slide)

# Leveraging the Linux CPU Scheduler

## How Linux schedules processes / threads

- Example scenario – 3 processes, two of which are multithreaded, plus some kthreads
- The conceptual diagram shows each thread's task structure within the kernel (*struct task\_struct*); being concise, the diagram doesn't show the user/kernel-mode stacks that each thread also has...
- As each task structure contains fields for sched policy, priority, class, and so on, this implies we **can query / set these fields on a per thread basis!**
- How? Via programming interfaces – utils, library APIs, system calls! (will be covered...)





# Leveraging the Linux CPU Scheduler

## The POSIX Scheduling Policies and what they mean

- Scheduling policy – essentially an algorithm that specifies how exactly threads (KSE's) are scheduled
  - Implemented by the OS as scheduling code within the kernel
- POSIX standard – OS must support three scheduling policies:
  - SCHED\_FIFO
  - SCHED\_RR
  - SCHED\_OTHER (or SCHED\_NORMAL)

# Leveraging the Linux CPU Scheduler

## The POSIX Scheduling Policies and what they mean

- The Linux OS supports these three scheduling policies, plus two more:
  - SCHED\_FIFO
  - SCHED\_RR
  - SCHED\_OTHER (or SCHED\_NORMAL)
  - SCHED\_BATCH
  - SCHED\_IDLE



# Leveraging the Linux CPU Scheduler

## The POSIX Scheduling Policies and what they mean

Scheduling policy	Key points	Priority scale
SCHED_OTHER or SCHED_NORMAL	Always the default; threads with this policy are non-real-time; internally implemented as a <b>Completely Fair Scheduling (CFS)</b> class (seen later in the <i>A word on CFS and the runtime value</i> section). The motivation behind this schedule policy is fairness and overall throughput.	Real-time priority is 0; the non-real-time priority is called the nice value: it ranges from -20 to +19 (a lower number implies superior priority) with a base of 0
SCHED_RR	The motivation behind this schedule policy is a (soft) real-time policy that's moderately aggressive. Has a finite timeslice (typically defaulting to 100 ms). A SCHED_RR thread will yield the processor IFF (if and only if): <ul style="list-style-type: none"><li>- It blocks on I/O (goes to sleep).</li><li>- It stops or dies.</li><li>- A higher-priority real-time thread becomes runnable (which will preempt this one).</li><li>- Its timeslice expires.</li></ul>	(Soft) real-time: 1 to 99 (a higher number implies superior priority)
SCHED_FIFO	The motivation behind this schedule policy is a (soft) real-time policy that's (by comparison, very) aggressive. A SCHED_FIFO thread will yield the processor IFF: <ul style="list-style-type: none"><li>- It blocks on I/O (goes to sleep).</li><li>- It stops or dies.</li><li>- A higher-priority real-time thread becomes runnable (which will preempt this one).</li></ul> It has, in effect, infinite timeslice.	(same as SCHED_RR)

What exactly do these scheduling policies mean?

Study the tables...

(The tables are from my *Linux Kernel Programming* book)

SCHED_BATCH	The motivation behind this schedule policy is a scheduling policy that's suitable for non-interactive batch jobs, less preemption.	Nice value range (-20 to +19)
SCHED_IDLE	Special case: typically the PID 0 kernel thread (traditionally called the swapper; in reality, it's the per CPU idle thread) uses this policy. It's always guaranteed to be the lowest-priority thread on the system and only runs when no other thread wants the CPU.	The lowest priority of all (think of it as being below the nice value +19)

# Leveraging the Linux CPU Scheduler

## The POSIX Scheduling Policies and what they mean

Scheduling policy	Key points	Priority scale
SCHED_OTHER or SCHED_NORMAL	Always the default; threads with this policy are non-real-time; internally implemented as a <b>Completely Fair Scheduling (CFS)</b> class (seen later in the <i>A word on CFS and the vruntime value</i> section). The motivation behind this schedule policy is fairness and overall throughput.	Real-time priority is 0; the non-real-time priority is called the nice value: it ranges from -20 to +19 (a lower number implies superior priority) with a base of 0
SCHED_RR	The motivation behind this schedule policy is a (soft) real-time policy that's moderately aggressive. Has a finite timeslice (typically defaulting to 100 ms). A SCHED_RR thread will yield the processor IFF (if and only if): - It blocks on I/O (goes to sleep). - It stops or dies. - A higher-priority real-time thread becomes runnable (which will preempt this one). - Its timeslice expires.	(Soft) real-time: 1 to 99 (a higher number implies superior priority)
SCHED_FIFO	The motivation behind this schedule policy is a (soft) real-time policy that's (by comparison, very) aggressive. A SCHED_FIFO thread will yield the processor IFF: - It blocks on I/O (goes to sleep). - It stops or dies. - A higher-priority real-time thread becomes runnable (which will preempt this one). It has, in effect, infinite timeslice.	(same as SCHED_RR)

So, listing them in 'priority order', on vanilla / mainline Linux, we have:

(RT) prio

- (Soft) Real-Time
  - SCHED\_FIFO
  - SCHED\_RR
- Non Real-Time
  - SCHED\_OTHER [default]
  - SCHED\_BATCH
  - SCHED\_IDLE

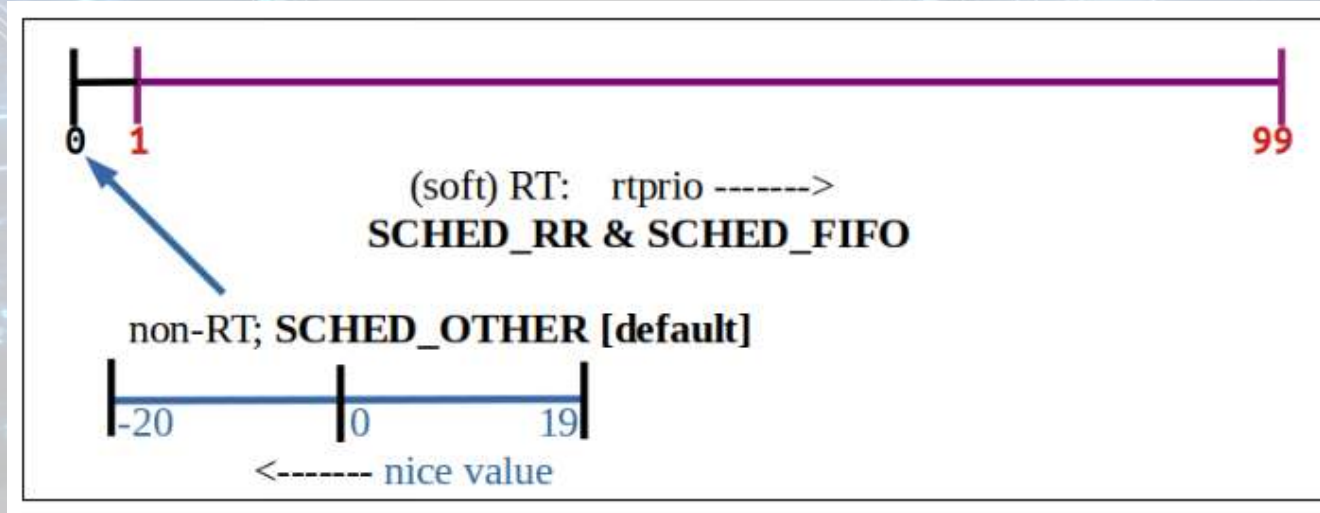
SCHED_BATCH	The motivation behind this schedule policy is a scheduling policy that's suitable for non-interactive batch jobs, less preemption.	Nice value range (-20 to +19)
SCHED_IDLE	Special case: typically the PID 0 kernel thread (traditionally called the swapper; in reality, it's the per CPU idle thread) uses this policy. It's always guaranteed to be the lowest-priority thread on the system and only runs when no other thread wants the CPU.	The lowest priority of all (think of it as being below the nice value +19)



# Leveraging the Linux CPU Scheduler

## The POSIX Scheduling Policies and what they mean

- So how do the (RT/non-RT) priorities work?
- This tells us: in the presence of a (soft) RT thread, normal (non-RT) threads – SCHED\_OTHER/BATCH/IDLE – will always ‘lose’
- (Visualize SCHED\_BATCH and SCHED\_IDLE as being non-RT as well)



# Leveraging the Linux CPU Scheduler Utilities – to ease the work effort!

- nice(1)
- renice(1)
- chrt(1)
- taskset(1)
  - A script (that uses chrt and taskset) to display the CPU sched policy, RT priority and CPU affinity mask of every process: [link](#)



# Leveraging the Linux CPU Scheduler

## The POSIX Scheduling Policies and what they mean

- A script that shows all threads, their scheduling policy, RT priority and CPU affinity mask;

[link.](#)

- *Sample run:*

```
scripts $ ./query_task_sched.sh
```

PID	TID	Name	Sched Policy	Prio	*RT	CPU-affinity-mask
1	1	systemd	SCHED_OTHER	0		fff
2	2	kthreadd	SCHED_OTHER	0		fff
3	3	rcu_gp	SCHED_OTHER	0		2
4	4	rcu_par_gp	SCHED_OTHER	0		fff
5	5	netns	SCHED_OTHER	0		fff
7	7	kworker/0:0H-events_highpri	SCHED_OTHER	0		1
9	9	mm_percpu_wq	SCHED_OTHER	0		2
10	10	rcu_tasks_rude_	SCHED_OTHER	0		fff
11	11	rcu_tasks_trace	SCHED_OTHER	0		fff
12	12	ksoftirqd/0	SCHED_OTHER	0		1
13	13	rcu_sched	SCHED_OTHER	0		fff
14	14	migration/0	SCHED_FIFO	99	***	1
15	15	idle_inject/0	SCHED_FIFO	50	*	1
17	17	cpuhp/0	SCHED_OTHER	0		1
18	18	cpuhp/1	SCHED_OTHER	0		2
19	19	idle_inject/1	SCHED_FIFO	50	*	2
20	20	migration/1	SCHED_FIFO	99	***	2
21	21	ksoftirqd/1	SCHED_OTHER	0		2
23	23	kworker/1:0H-events_highpri	SCHED_OTHER	0		2
24	24	cpuhp/2	SCHED_OTHER	0		4
25	25	idle_inject/2	SCHED_FIFO	50	*	4
26	26	migration/2	SCHED_FIFO	99	***	4
27	27	ksoftirqd/2	SCHED_OTHER	0		4

# Leveraging the Linux CPU Scheduler

## Setting CPU scheduling policy and priority on an application (process) thread

- How does one query / set a thread's CPU scheduling policy and/or priority?
  - Both Pthreads API as well as system calls
  - The Pthreads APIs are in fact (and obviously) wrappers over the system calls
  - (as, if anything needs to be done within the kernel, we require a syscall – it switches the calling process/thread to the kernel code path and the code executes with kernel privilege)



# Leveraging the Linux CPU Scheduler

Setting CPU scheduling policy and priority on an application (process) thread

- Pthreads APIs - for a given thread:
  - `pthread_getschedparam(3)` : query scheduling policy and priority
  - `pthread_setschedparam(3)` : set scheduling policy and priority
    - the 'set' API requires either root access or the CAP\_SYS\_NICE capability bit

# Leveraging the Linux CPU Scheduler

Setting CPU scheduling policy and priority on an application (process) thread

- System calls
  - `sched_setattr(2)`, `sched_setparam(2)`
  - `sched_getattr(2)`, `sched_setscheduler(2)`
  - the 'set' APIs require either root access or the `CAP_SYS_NICE` capability bit
- Tip- Ubuntu: try doing  
`man -k sched` *to see all sched-related man pages!*
- Learn more: the man page `sched(7)` is useful!



# Leveraging the Linux CPU Scheduler

## Setting CPU scheduling policy and priority on an application (process) thread

More syscalls for scheduling...

From my *Hands-On System Programming with Linux* book

Here's a list—a sampling, really—of some of the more important of these APIs:

- `sched_setscheduler(2)`: Sets the scheduling policy and parameters of a specified thread.
- `sched_getscheduler(2)`: Returns the scheduling policy of a specified thread.
- `sched_setparam(2)`: Sets the scheduling parameters of a specified thread.
- `sched_getparam(2)`: Fetches the scheduling parameters of a specified thread.
- `sched_get_priority_max(2)`: Returns the maximum priority available in a specified scheduling policy.
- `sched_get_priority_min(2)`: Returns the minimum priority available in a specified scheduling policy.
- `sched_rr_get_interval(2)`: Fetches the quantum used for threads that are scheduled under the round-robin scheduling policy.
- `sched_setattr(2)`: Sets the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of `sched_setscheduler(2)` and `sched_setparam(2)`.
- `sched_getattr(2)`: Fetches the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of `sched_getscheduler(2)` and `sched_getparam(2)`.

# Leveraging the Linux CPU Scheduler

Setting CPU scheduling policy and priority on an application (process) thread

- Here, we focus on the Pthreads APIs only...
- Their signature:

```
#include <pthread.h>
```

```
int pthread_setschedparam(pthread_t thread, int policy,  
                           const struct sched_param *param);  
  
int pthread_getschedparam(pthread_t thread, int *policy,  
                           struct sched_param *param);
```

## • Parameters:

- pthread\_t thread : the thread to query / set
- int policy : one of (the supported CPU scheduling policies): SCHED\_FIFO, SCHED\_RR, SCHED\_RR, SCHED\_OTHER (or SCHED\_NORMAL), SCHED\_BATCH, SCHED\_IDLE
- struct sched\_param \*param : contains only 1 member, the CPU scheduling priority:

```
struct sched_param {  
    int sched_priority;    /* Scheduling priority */  
};
```

- Recall, the 'set' API(s) requires either root access or the CAP\_SYS\_NICE capability bit
- As the man pages show us, several related APIs exist:
- SEE ALSO

getrlimit(2), sched\_get\_priority\_min(2), pthread\_attr\_init(3), pthread\_attr\_setinheritsched(3), pthread\_attr\_setschedparam(3), pthread\_attr\_setschedpolicy(3), pthread\_create(3), pthread\_self(3), pthread\_setschedprio(3), pthreads(7), sched(7)



# Leveraging the Linux CPU Scheduler

## Demo and Code walkthrough - MultiThreaded (MT) app running as (soft) Real-Time

- The demo MT app code's available here: [link](#)

```
demo_app $ ./runit.sh
FYI: sched_rt_period_us and sched_rt_runtime_us values:
1000000
1000000

sched_pthrd_rtprio_dbg already has the capability CAP_SYS_NICE enabled
[+] taskset -c 01 ./sched_pthrd_rtprio_dbg 20
sched_pthrd_rtprio.c:main : SCHED_FIFO priority range is 1 to 99

Note: to create true (soft) RT threads, and have it run as expected, you need to:
    1. Run this program as superuser -OR- have the capability CAP_SYS_NICE (better!)
    2. Ensure it runs on a single CPU core (use, f.e., taskset -c02 <prg-name>)
main() thread (352862): now creating realtime pthread p2..
main() thread (352862): now creating realtime pthread p3..
m RT Thread p3 (LWP 352862) here in function thrd_p3()
  setting sched policy to SCHED_FIFO and RT priority HIGHER to 30 in 4 seconds..
m RT Thread p2 (LWP 352862) here in function thrd_p2()
  setting sched policy to SCHED_FIFO and RT priority to 20 in 2 seconds..
m p2: working
m p3: working
m p3: exiting..
m p2: exiting..
demo_app $
```

# Leveraging the Linux CPU Scheduler

## Demo - MultiThreaded (MT) app

- Here, we're running on an x86\_64 with Ubuntu 20.04 LTS and the 5.15.0-52-generic kernel (for system tunables, the arch and kernel ver does matter)
- System Tunables related to CPU sched does affect the output!
  - `/proc/sys/kernel/sched_rt_period_us` : time (microseconds) of the total 'period'
  - `/proc/sys/kernel/sched_rt_runtime_us` : time (microseconds) that an RT (SCHED\_FIFO/SCHED\_RR) task will be allowed to consume of the total period
- ```
# cat /proc/sys/kernel/sched_rt_period_us /proc/sys/kernel/sched_rt_runtime_us
```

```
1000000
950000
```
- So: an RT task/thread is by default allowed to consume 950,000 us of 1,000,000 us, i.e., 950 ms of 1 s, i.e., 95% of the CPU bandwidth
- This implies that the kernel by default allows non-RT tasks 5% CPU bandwidth; IOW, we 'leak' a little CPU so that non-RT threads don't completely starve!
- (This is why the main() thread was able to – perhaps – print some characters in between!)





# Leveraging the Linux CPU Scheduler

Now that you know how to write a RT app (with C on Linux), here are some exercises for you to try:

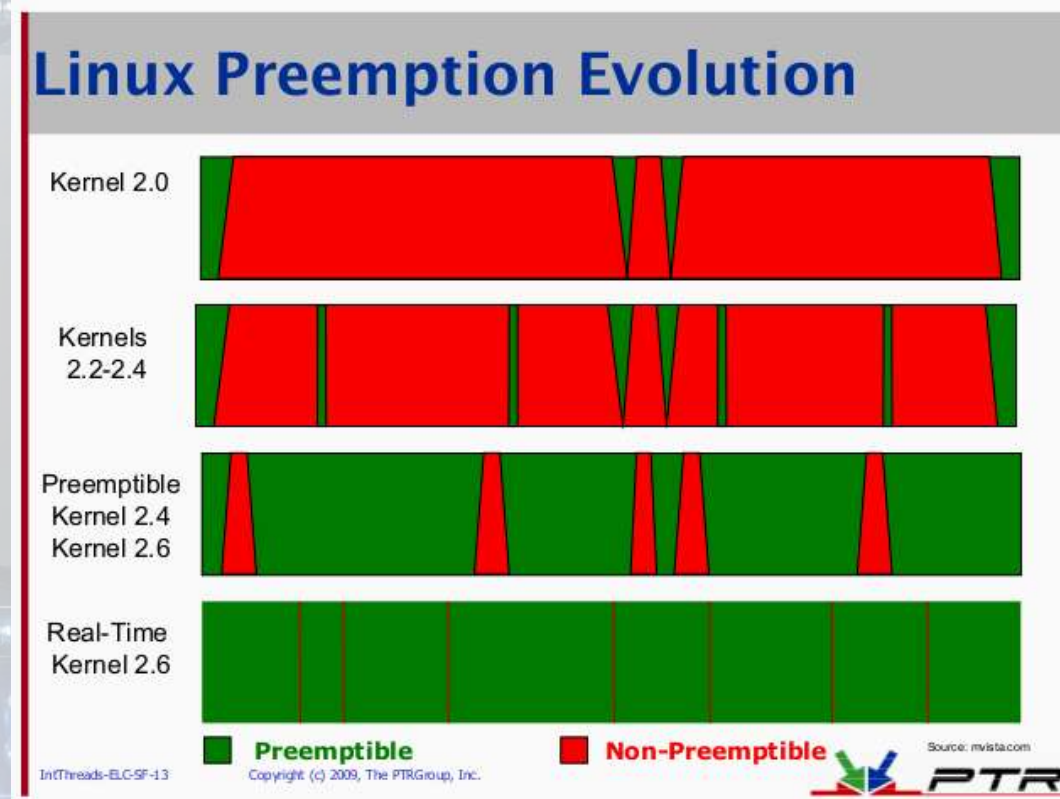
[Link](#) (to a file on the GitHub repo of *Hands-On System Programming with Linux*).



# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL (Real-Time Linux)

- Evolving the Linux kernel to becoming (almost a 100%) preemptible!
- Has ultimately led to the creation of the hard RT Linux project – RTL!
- IOW, we can now run Linux as a true RTOS ! While still having the really useful POSIX programming interface
- *"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT\_RT."* -- Linus Torvalds



# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

- Founder and project lead – Thomas Gleixner
- Thomas – and collaborators - have been working to port regular (vanilla) Linux to an RTOS for many years
- Ever since the 2.6.18 kernel (back in Sept 2006!), there are patches available to convert Linux into an RTOS!
- This original – and tremendous – effort is named the **PREEMPT\_RT** patch
- More recently, from Oct 2015, the Linux Foundation has adopted the project; it's now called *The RTL Collaborative Project*; **RTL = Real-Time Linux !**
  - We shouldn't confuse RTL with co-kernel approaches such as Xenomai or RTAI, or the older and now-defunct attempt called RTLinux



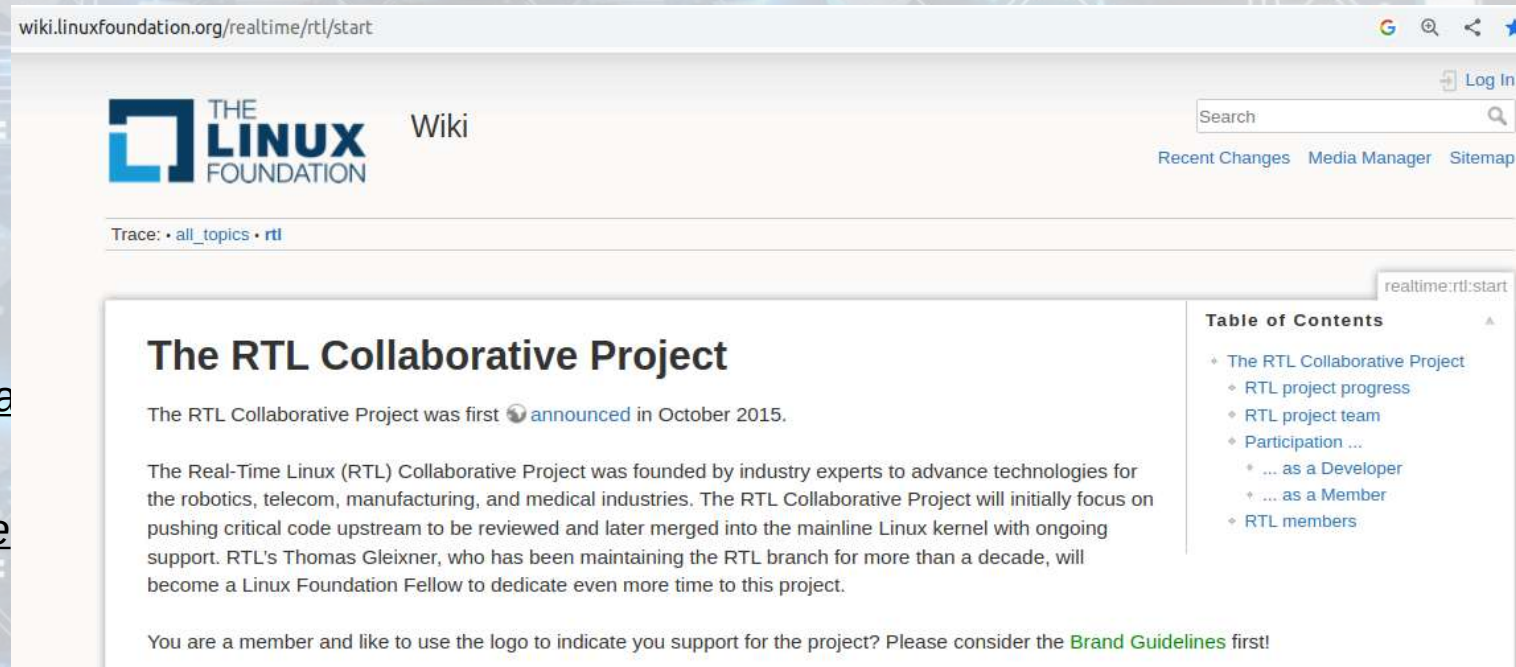


# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

### FAQs regarding RTL

- Q. Is there documentation, a Wiki?
- A. Yes indeed:  
<https://wiki.linuxfoundation.org/realtime/rtl/start>
- Also, the 'old' Wiki site is still very useful!

A screenshot of the 'The RTL Collaborative Project' page on the Linux Foundation Wiki. The page title is 'The RTL Collaborative Project'. The text states that the project was first announced in October 2015. It describes the project's goal to advance technologies for robotics, telecom, manufacturing, and medical industries by pushing critical code upstream to the mainline Linux kernel. A 'Table of Contents' on the right lists sections like 'The RTL Collaborative Project', 'RTL project progress', 'RTL project team', 'Participation ...', and 'RTL members'. The page also includes a 'Log In' button, a search bar, and links for 'Recent Changes', 'Media Manager', and 'Sitemap'. The URL in the browser address bar is 'wiki.linuxfoundation.org/realtime/rtl/start'.

wiki.linuxfoundation.org/realtime/rtl/start

THE LINUX FOUNDATION Wiki

Search

Recent Changes Media Manager Sitemap

Trace: • all\_topics • rtl

### The RTL Collaborative Project

The RTL Collaborative Project was first announced in October 2015.

The Real-Time Linux (RTL) Collaborative Project was founded by industry experts to advance technologies for the robotics, telecom, manufacturing, and medical industries. The RTL Collaborative Project will initially focus on pushing critical code upstream to be reviewed and later merged into the mainline Linux kernel with ongoing support. RTL's Thomas Gleixner, who has been maintaining the RTL branch for more than a decade, will become a Linux Foundation Fellow to dedicate even more time to this project.

You are a member and like to use the logo to indicate you support for the project? Please consider the [Brand Guidelines](#) first!

#### Table of Contents

- The RTL Collaborative Project
  - RTL project progress
  - RTL project team
  - Participation ...
    - ... as a Developer
    - ... as a Member
  - RTL members

# Leveraging the Linux CPU Scheduler

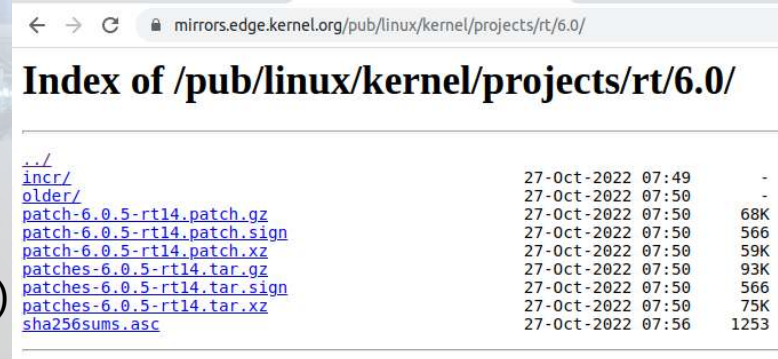
## Converting vanilla (GPOS) Linux to an RTOS with RTL

### FAQs regarding RTL

- Q. Is the code a part of the Linux kernel?

A. No; it's available as a patch series (or a single patch) from the kernel.org site here:

<https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/>



The screenshot shows a web browser window with the URL <https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/6.0/>. The page title is "Index of /pub/linux/kernel/projects/rt/6.0/". It lists various files and directories with their last modification dates and sizes.

| File/Directory              | Last Modified     | Size |
|-----------------------------|-------------------|------|
| ./                          | 27-Oct-2022 07:49 | -    |
| incr/                       | 27-Oct-2022 07:50 | -    |
| older/                      | 27-Oct-2022 07:50 | -    |
| patch-6.0.5-rt14.patch.gz   | 27-Oct-2022 07:50 | 68K  |
| patch-6.0.5-rt14.patch.sign | 27-Oct-2022 07:50 | 566  |
| patch-6.0.5-rt14.patch.xz   | 27-Oct-2022 07:50 | 59K  |
| patches-6.0.5-rt14.tar.gz   | 27-Oct-2022 07:50 | 93K  |
| patches-6.0.5-rt14.tar.sign | 27-Oct-2022 07:50 | 566  |
| patches-6.0.5-rt14.tar.xz   | 27-Oct-2022 07:50 | 75K  |
| sha256sums.asc              | 27-Oct-2022 07:56 | 1253 |

- The RTL patch does *not* exist for every single kernel release (too many of 'em)
- You'll find directories containing the RTL patch(es) for various selected kernel releases
- For example, at the time of this writing, the (single) RTL patch for the latest stable kernel release, 6.0, is here:  
<https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/6.0/patch-6.0.5-rt14.patch.xz>
- Note the version #; it can *only* be applied upon the 6.0.5 kernel.



# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

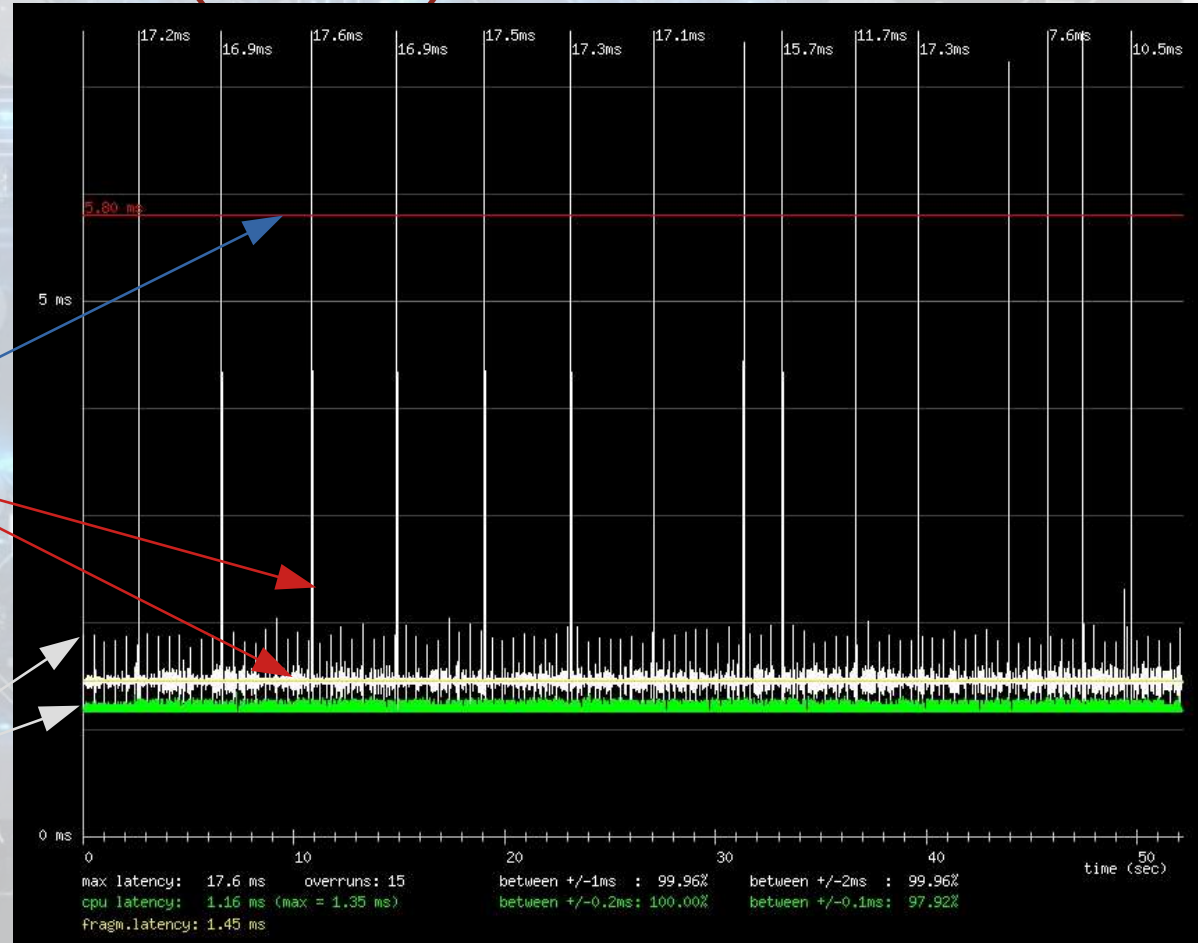
### FAQs regarding RTL

- Q. Why isn't RTL a part of the mainline kernel?  
A. (From my *Linux Kernel Programming* book):  
"- Much of the RTL work has indeed been merged into the mainline kernel; this includes important areas such as the scheduling subsystem, mutexes, lockdep, threaded interrupts, PI, tracing, and so on. In fact, **an ongoing primary goal of RTL is to get it merged** as much as is feasible (we show a table summarizing this in the *Mainline and RTL – technical differences summarized* section).  
- Linus Torvalds deems that Linux, being primarily designed and architected as a GPOS, should not have highly invasive features that only an RTOS really requires; so, though patches do get merged in, it's a slow deliberated process. ..."
- Q. How can I check if RTL is enabled, within code  
A. `if (IS_ENABLED(CONFIG_PREEMPT_RT))`  
[...]
- For detailed explanations on applying the RTL patch, reconfiguring and rebuilding the kernel, etc on a Raspberry Pi: pl see *Linux Kernel Programming, Ch 11 – The CPU Scheduler, Part 2* section *Converting mainline Linux into an RTOS* (based on the 5.4 LTS kernel).

# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

- Results? Does RTL help?
- **latencytest** – Benno Sennoner; audio sampling under load
- Graph - result of a *LatencyTest benchmark on a Standard (vanilla 2.6) kernel*
- White vertical bars represent the latency
- The red line represents the latency where the human ear can detect dropouts
- Notice that though it's overall very good, there is considerable jitter





# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

- **Results?**
- *latencytest* – Benno Sennoner; audio sampling under load
- Result of a LatencyTest Benchmark on a 2.6 Preemptible Kernel (CONFIG\_PREEMPT)
- Notice that it's overall excellent! Plus the jitter is now very low...



# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

- **Results?**
  - using the ***cyclicttest*** utility
- From: *Linux Kernel Programming, Ch 11 – The CPU Scheduler, Part 2* section *Measuring scheduling latency with cyclicttest*
  - Basic procedure:
    - Place the DUT (Device Under Test), a Raspberry Pi, under load:  
`stress --cpu 6 --io 2 --hdd 4 \`  
`--hdd-bytes 1MB --vm 2 \`  
`--vm-bytes 128M --timeout 1h`
    - In parallel, run cyclicttest:  
`sudo cyclicttest -duration=1h \`  
`-m -Sp90 -i200 -h400 \`  
`-q >output`
    - Compute/tabulate results

### The CPU Scheduler - Part 2

Chapter 11

## Viewing the results

We carry out a similar procedure for the remaining two test cases and summarize the results of all three in Figure 11.14:

| DUT (Device Under Test)                                | System Latency (us) |         |           |
|--------------------------------------------------------|---------------------|---------|-----------|
|                                                        | Min                 | Avg     | Max       |
| Raspberry Pi 3B+ ; 5.4.70-rt40 RTL kernel              | 7 us                | 26 us   | 256 us    |
| Raspberry Pi 3B+ ; 5.4.51-v7+ standard kernel          | 3 us                | 16.3 us | 14,595 us |
| x86_64 ; Ubuntu 20.04 5.4.0-48-generic standard kernel | 1 us                | 3.8 us  | 21,027 us |

Figure 11.14 – Results of the (simplistic) test cases we ran showing the min/avg/max latencies for different kernels and systems while under some stress



# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

- Results?
  - using the *cyclictest* utility
- “Interesting; though the maximum latency of the RTL kernel is much below the other standard kernels, both the minimum and, more importantly, average latencies are superior for the standard kernels. This ultimately results in superior overall throughput for the standard kernels”
- “... the results quite clearly show that it's deterministic (a very small amount of jitter) with an RTOS and highly non-deterministic with a GPOS! (As a rule of thumb, standard Linux will result in approximately +/- 10 microseconds of jitter for interrupt processing, whereas on a microcontroller running an RTOS, the jitter will be far less, around +/- 10 nanoseconds!)

# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

- For a hard-RT project, we require
  - an RTOS (like RTL)
  - apps carefully written with RT guidelines in mind! (esp those in the time-critical path)
- **App Guidelines**
  - don't perform non-deterministic / possibly blocking operations in time-critical code paths (eg. malloc(), ...)
  - use mlock[all](2) to keep memory regions 'locked' – marked as non-swappable
  - can use madvise(2) to provide hints to the kernel regarding memory - sequential reads, read-ahead, etc
  - From the older RTL Wiki site (unsure if links are current):  
*Tips and Techniques*
    - [HOWTO: Build an RT-application](#)
    - [CPU shielding using /proc and /dev/cpuset](#)
    - [Cpuset management utility/tutorial](#)
    - [How to tune your IRQ priorities for low-latency audio](#)

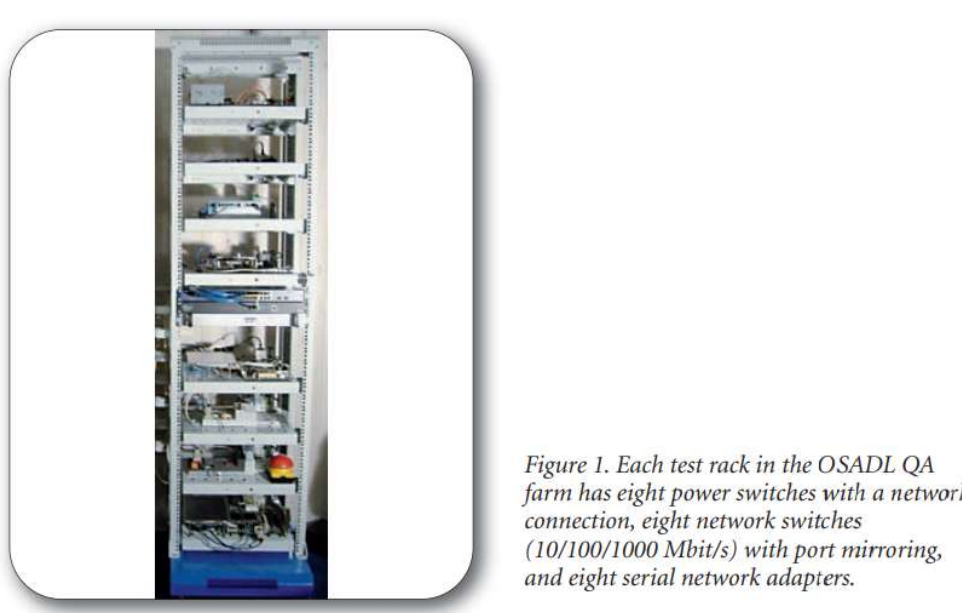


# Leveraging the Linux CPU Scheduler

## Converting vanilla (GPOS) Linux to an RTOS with RTL

- Examples of using RTL in the real world:
  - See the 'old' Wiki site:  
[https://rt.wiki.kernel.org/index.php/Systems\\_based\\_on\\_Real\\_time\\_preempt\\_Linux](https://rt.wiki.kernel.org/index.php/Systems_based_on_Real_time_preempt_Linux)

- OSADL test rack



# Leveraging the Linux CPU Scheduler

- **More to learn (*always!*), on Linux CPU Scheduling !**
  - CPU Sched tunables (/proc/sys/kernel/sched\_\*); see man page on proc(5)
  - Visualizing process/thread execution (with LTTng, trace-cmd, KernelShark, ...)
  - Kernel internals wrt Linux OS CPU scheduling
    - modular scheduler classes, their per-cpu runqueues, how & when is schedule() invoked, etc
  - Control Groups (Cgroups)
  - Latency and it's measurement
  - For all these topics – and much more! - do read/refer *Linux Kernel Programming*.



# Leveraging the Linux CPU Scheduler

*Thank you !*

Q&A