

Leveraging the Linux CPU Scheduler To Write Real-Time MT Apps

Kaiwan N Billimoria

#B'lore Kernel Meetup, April 2025

Leveraging the Linux CPU Scheduler

\$ whoami

I've been in the software industry from late 1991

I worked on Unix and then 'discovered' Linux around '96-'97 (2.0/2.1 kernel's)

Been glued to it ever since!

Self-taught: Linux scripting, apps, kernel OS/drivers, embedded Linux, debug
all along the journey

Of course, the learning continues to this day!

Contributed to open source as well as closed-source projects; Linux kernel, a v
small bit...

My GitHub repos: <https://github.com/kaiwan>

Leveraging the Linux CPU Scheduler

\$ whoami

Author of five books on Linux (all published by Packt Publishing, Birmingham, England)

Linux Kernel Programming, 2nd Ed, Feb 2024

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](#)

[My LFX \(Linux Foundation\) profile page](#)

The screenshot shows the Amazon author page for Kaiwan N Billimoria. At the top, there's a profile picture and the name 'Kaiwan N Billimoria' with a 'Following' button. Navigation tabs for 'HOME', 'ABOUT', and 'ALL BOOKS' are visible. The main header displays the author's name. Below this, the 'About the author' section provides a bio: 'Kaiwan Billimoria taught himself BASIC programming on his Dad's office IBM PC when he was in the 9th grade (back in 1983). The urge to learn, hack and master at the level of...' with a 'Read full bio' link. The 'Most popular' section features a book cover for 'Linux Kernel Programming' with a 34-star rating and a price of \$31.19. The 'Top Kaiwan N Billimoria titles for you' section displays five book covers with their respective ratings: 'Linux Kernel Programming' (34 stars), 'Linux Kernel Programming: A comprehensive a...' (34 stars), 'Linux Kernel Debugging: Leverage proven...' (19 stars), 'Linux Kernel Programming Part 2 - Char Device D...' (116 stars), and 'Hands-On System Programming with Linux: Explore Li...' (19 stars).

Kaiwan N Billimoria

✓ Following HOME ABOUT ALL BOOKS

Kaiwan N Billimoria

About the author

Kaiwan Billimoria taught himself BASIC programming on his Dad's office IBM PC when he was in the 9th grade (back in 1983). The urge to learn, hack and master at the level of...

[Read full bio](#)

Most popular

Linux Kernel Programming
Linux Kernel Programming: A comprehensive and practical guide...
★★★★☆ 34
Kindle Edition
\$31¹⁹

Top Kaiwan N Billimoria titles for you

- Linux Kernel Programming**
Linux Kernel Programming: A comprehensive a...
★★★★☆ 34
- Linux Kernel Programming**
Linux Kernel Programming: A comprehensive a...
★★★★☆ 34
- Linux Kernel Debugging**
Linux Kernel Debugging: Leverage proven...
★★★★☆ 19
- Linux Kernel Programming Part 2 - Char Device D...**
Linux Kernel Programming Part 2 - Char Device D...
★★★★☆ 116
- Hands-On System Programming with Linux**
Hands-On System Programming with Linux: Explore Li...
★★★★☆ 19

Leveraging the Linux CPU Scheduler

\$ whoami

GitHub repo for this presentation

kaiwanTECH LinkTree

LinkedIn public profile

My Tech Blog [please do follow!]

Corporate training

My GitHub repos [a request: please do star the repos you like]

Leveraging the Linux CPU Scheduler

Agenda

What does real-time actually mean

The Linux process state machine

How Linux schedules processes and threads – an overview

(a bit on kernel architecture, as required)

The POSIX Scheduling Policies and what they mean

Setting CPU scheduling policy and priority on an application thread

Demo soft RT MT app

Converting vanilla Linux to an RTOS

Leveraging the Linux CPU Scheduler

This presentation: the materials

All materials used here - the PDF slides + the source code – can be found here:

https://github.com/kaiwan/cpu_sched_demo

Do git clone it!

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 – the 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

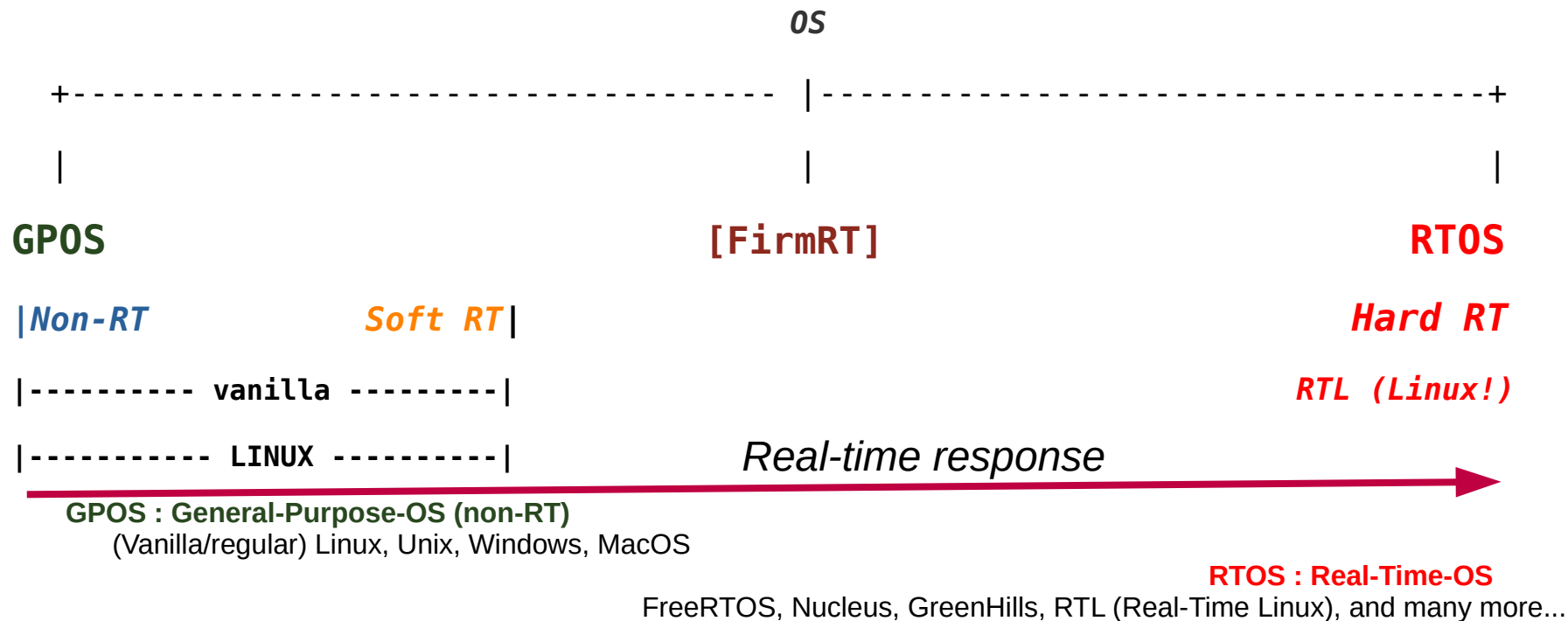
In a real-time system, while max system latencies can dramatically reduce, overall throughput can suffer

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



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

Examples include most personal / business / enterprise software, 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 many embedded systems, consumer electronics devices (smartphones, media players, etc)

[Firm RT : between Soft and Hard RT]

RTOS / 'Hard' RT

Deterministic, virtually no (or very low) jitter

Deadline(s) **MUST** be met a 100% of the time, else results in catastrophic failure, even financial and/or loss to human life

Examples include many kinds of transport, ATC systems, stock exchanges, medical electronics (pacemakers), factory floor robots, etc

Leveraging the Linux CPU Scheduler

REMEMBER ! A Key Point

The vanilla / mainline Linux OS is always a GPOS. Though a GPOS, it is (only) **SOFT** Real-Time capable, *not* 'hard RT'

I do mention how to 'convert' regular (vanilla) Linux to a hard real-time RTOS later in this presentation

Leveraging the Linux CPU Scheduler

The Linux process state machine

CPU Scheduling on Linux

Chapter 17

From the 'Hands-On
System
Programming with
Linux' book

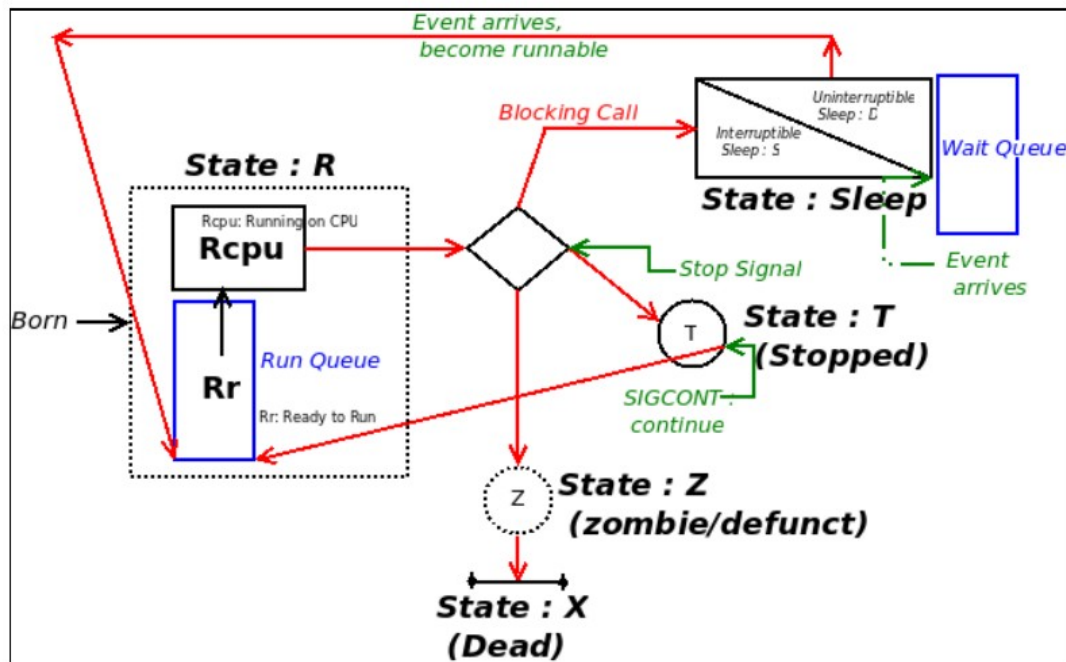


Figure 1: Linux state machine

How Linux schedules processes / threads

Key point

In CS, there is the notion of the **Kernel Schedulable Entity (KSE)**

What entity exactly, does the OS schedule?

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

A pillar of the classic UNIX / Linux design 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

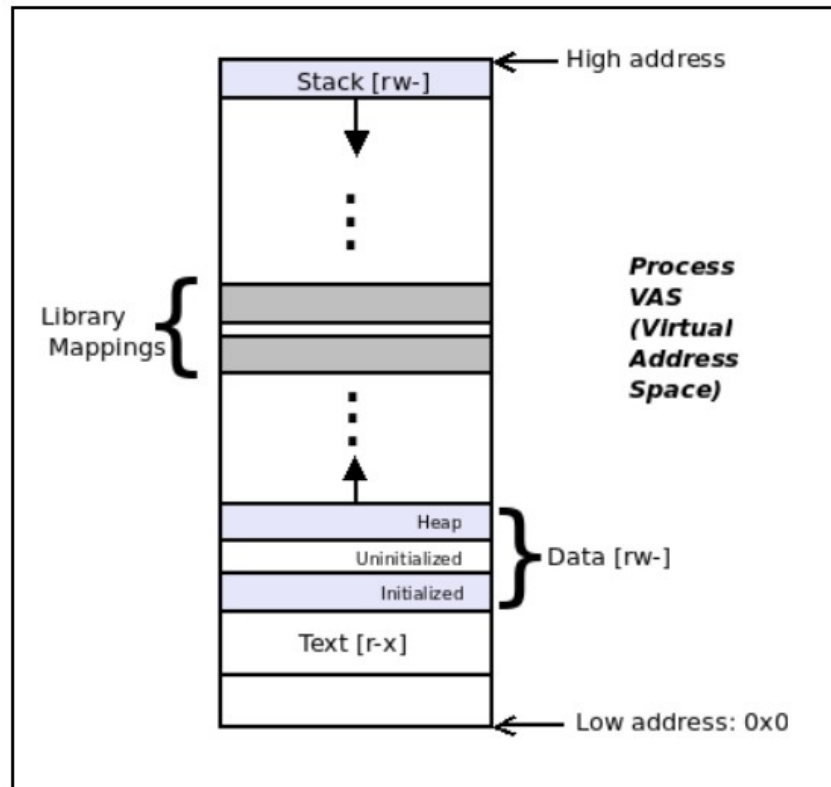
- a process must have at least one thread – main()! (aka T0)

- It can have >1 thread; if so, it’s termed a **multithreaded (MT)** process

Leveraging the Linux CPU Scheduler

How Linux schedules processes / threads:

*The process
Virtual Address Space
(VAS) – Userspace
(simplified)*



Leveraging the Linux CPU Scheduler

How Linux schedules processes / threads

Key points

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

 - Exception: kthreads only 'see' kernel-space and thus have only a kernel-mode stack

Leveraging the Linux CPU Scheduler

How Linux schedules processes / threads

Example scenario – 3 processes, two of which are multithreaded, plus a few kthreads (kernel threads)

The conceptual diagram shows each thread's task structure within the kernel (struct task_struct)

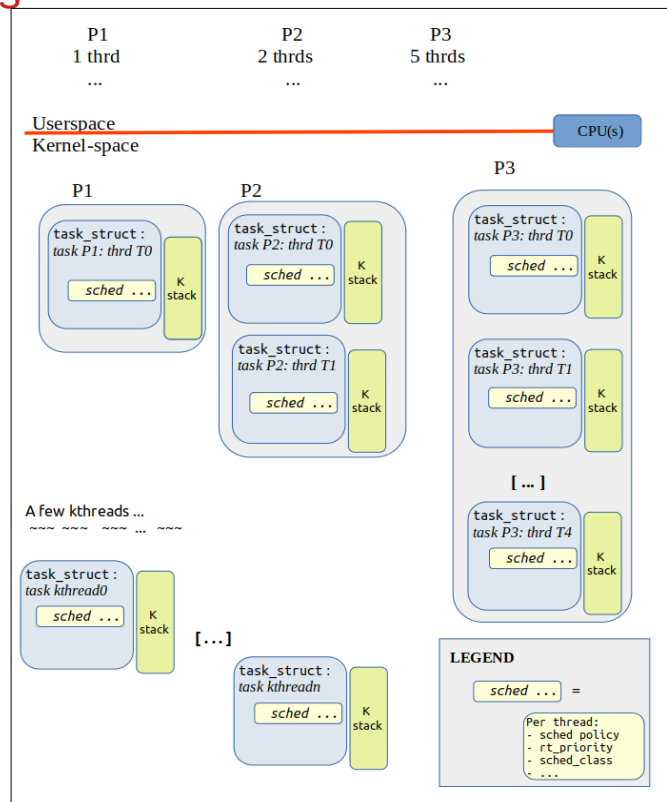
Furthermore, each (usermode) thread has 2 stacks

- a usermode stack (not seen here)

- a kernel-mode stack (seen)

As each task structure, and thus each thread, contains sched-related fields - sched policy, priority, class, and so on - this implies we can query / set these fields on a per thread basis!

How? Via various 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 (class) code within the kernel

POSIX standard – an 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 Completely Fair Scheduling (CFS) (seen later in the <i>4 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">- 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: <ul style="list-style-type: none">- 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)

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 <i>swapper</i> ; 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 Completely Fair Scheduling (CFS) 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">- 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: <ul style="list-style-type: none">- 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:

(Soft) Real-Time

SCHED_FIFO

SCHED_RR

Non Real-Time

SCHED_OTHER [default]

SCHED_BATCH

SCHED_IDLE

(RT) prio



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 <i>swapper</i> ; 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)

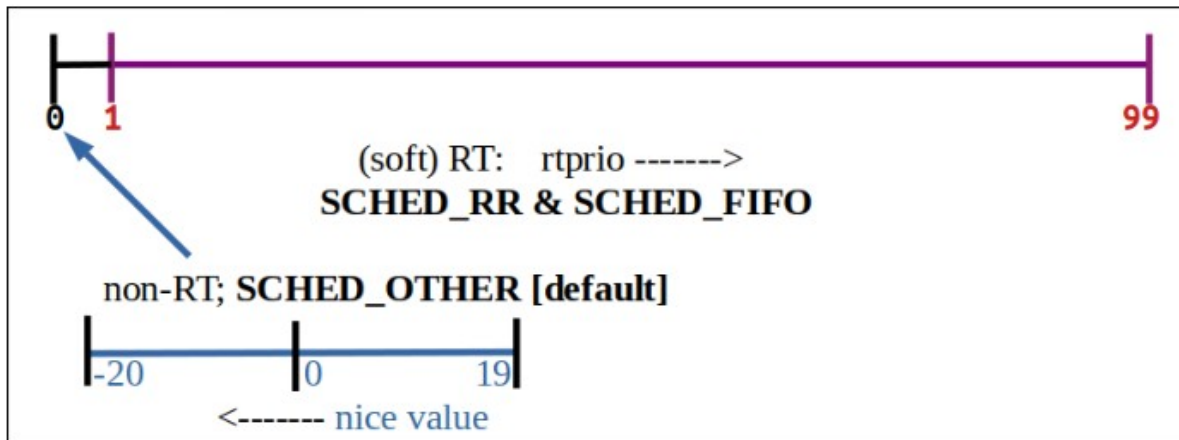
Leveraging the Linux CPU Scheduler

The POSIX Scheduling Policies and what they mean

So how do the **priorities** – for both RT and non-RT threads - work?

The diagram below clearly shows us that, in the presence of a (soft) RT thread, normal (non-RT) threads – SCHED_OTHER/BATCH/IDLE – will always 'lose'

(Of course. visualize SCHED_BATCH and SCHED_IDLE as being non-realtime)



Leveraging the Linux CPU Scheduler

Utilities – to ease the work effort!

`nice(1)`

`renice(1)`

`chrt(1)`

`taskset(1)`

`query_task_sched.sh`: a small wrapper script I wrote (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

See a
recording of
this script
[here!](#) (via
asciinema !)

Leveraging the Linux CPU Scheduler

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

So: how *does* one query or set a given thread's CPU scheduling policy and/or priority?

Via 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 required kernel code path; the kernel (/ driver) code then executes with kernel privilege)

Leveraging the Linux CPU Scheduler

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

Via Pthreads APIs - for a given thread:

`pthread_getschedparam(3)` : **query** scheduling policy and priority

`pthread_setschedparam(3)` : **set** scheduling policy and priority

Note that 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

Via system calls

`sched_setattr(2)`, `sched_setparam(2)`, `sched_setscheduler(2)`

`sched_getattr(2)`, `sched_getparam(2)`, `sched_getscheduler(2)`

Note that the 'set' APIs require either root access or the CAP_SYS_NICE capability bit

Tip: on 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_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 the caller to have 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

The demo soft RT
MT app code's
available here:
link

See a recording of
this program [here](https://asciinema.org/a/716622)! [
<https://asciinema.org/a/716622>
]
(via asciinema)

```
demo_app $ ./runit.sh -h
Usage: runit.sh [option]
 0 : do NOT set the runtime value - the 'max cpu bandwidth' - to the period value,
    so that the (soft) RT threads do 'leak' any CPU to non-RT threads
    (this is the default behaviour)
 1 : DO set the runtime value - the 'max cpu bandwidth' - to the period value,
    so that the (soft) RT threads do NOT 'leak' any CPU to non-RT threads
-h : show this help screen
demo_app $
demo_app $ ./runit.sh
FYI: lets lookup a couple of kernel sched-related tunables:
sched_rt_period_us = 1000000
sched_rt_runtime_us = 950000

sched_pthrdrtprio_dbg already has the capability CAP_SYS_NICE enabled
[+] taskset -c 01 ./sched_pthrdrtprio_dbg 20
sched_pthrdrtprio.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 (1985210): now creating realtime pthread p2..
main() thread (1985210): now creating realtime pthread p3..
m RT Thread p3 (LWP 1985210) here in function thrdrtprio_p3()
  setting sched policy to SCHED_FIFO and RT priority HIGHER to 30 in 4 seconds..
mm RT Thread p2 (LWP 1985210) here in function thrdrtprio_p2()
  setting sched policy to SCHED_FIFO and RT priority to 20 in 2 seconds..
p2: working
p3: working
p3: exiting..
p2: exiting..
demo_app $
```

Leveraging the Linux CPU Scheduler

Demo - MultiThreaded (MT) app

Here, we're running on an x86_64 with Ubuntu 22.04 LTS and the 5.19.0-35-generic kernel (for system tunables, the arch and kernel ver does matter)

System Tunables related to CPU sched does affect the output! Here, the relevant kernel tunables are:

`/proc/sys/kernel/sched_rt_period_us` : time (microseconds) of the total 'period'

`/proc/sys/kernel/sched_rt_runtime_us` : time (microseconds) that a soft RT (SCHED_FIFO / SCHED_RR) task will be allowed to consume of the total period

DEFAULTS

```
# cat /proc/sys/kernel/sched_rt_period_us /proc/sys/kernel/sched_rt_runtime_us
1000000
950000
```

[TIP: quickly lookup kernel / sysctl tunables via <https://sysctl-explorer.net/>]

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, the kernel 'leaks' a little CPU so that non-RT threads don't completely starve!

(This is why, in the default case, the main() thread was able to – perhaps – print some characters in between!)

To Try! If you pass the parameter 1 to our runit.sh script, it sets the runtime value to the period, thus NOT allowing any CPU bandwidth 'leakage'

Demo - MultiThreaded (MT) app

Just as can be done with our `runit.sh` script (by passing the parameter 1), below, we show how you can 'manually' setup to allow no CPU bandwidth 'leakage': We can of course, as root, change the defaults; let's try setting up **such that our (soft) RT tasks (pthreads) get 100% CPU when they want it:**

```
# echo 1000000 > /proc/sys/kernel/sched_rt_runtime_us
# cat /proc/sys/kernel/sched_rt_period_us /proc/sys/kernel/sched_rt_runtime_us
1000000
1000000
```

Now run the app:

```
$ sudo taskset 02 ./sched pthread rtprio dbg 12
```

```
[...] setting sched policy to SCHED_FIFO and RT priority to 12 in 2 seconds..
```

[illegible][illegible][illegible][illegible][illegible]

Works! (no CPU 'leakage' to the main() non-RT thread...)

Leveraging the Linux CPU Scheduler

Demo - MultiThreaded (MT) app

systemd and CPU scheduling

systemd, the modern init framework on Linux, is powerful; allows one to set various configs and tunables - non programatically, by editing the *service unit* file - when starting an app at boot

See the man page on `systemd.exec(5)` for details

Among them, are the following:

CPU Scheduling related

`Nice=` ; set the nice level of a non-RT thread or process

`CPUSchedulingPolicy=` ; sets the CPU scheduling policy for executed processes. Takes one of `other`, `batch`, `idle`, `fifo` or `rr`

f.e. the `e2scrub_reap.service` unit sets the `CPUSchedulingPolicy=idle` (implying, run it when the system's idle) [[link](#)]

`CPUSchedulingPriority=` ; sets the CPU scheduling priority for executed processes

`CPUAffinity=` ; Controls the CPU affinity of the executed processes

Can set **resource limits** on the process(es) being executed

`LimitCPU=`, `LimitFSIZE=`, `LimitDATA=`, `LimitSTACK=`, `LimitCORE=`, `LimitRSS=`, `LimitRTPRIO=`, `LimitRTTIME=`, ...

Capabilities you wish the process to have when running (with the `CapabilityBoundingSet=XXX ...`)

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)

The typical Linux OS's (SCHED_FIFO and SCHED_RR) CPU scheduling schemas can be essentially summarized as a

Fixed Priority Preemptive Scheduler

We saw that priorities are present: non-RT 'nice' values (-20 to +19) and a (soft) RT priority scale from 1 to 99

Fixed priority: the app developer *fixes* the thread priority; the OS honors it

Preemptive: from Wikipedia: "In computing, **preemption** is the act of temporarily interrupting an executing task, with the intention of resuming it at a later time. This interrupt is done by an external scheduler with no assistance or cooperation from the task..."

Two kinds of preemption wrt CPU scheduling:

- usermode preemption (the 'while(1) and analog clock processes' thought experiment!)
- kernel-mode preemption (Linux 2.6 onward is capable of it; it's a kernel config)

Leveraging the Linux CPU Scheduler

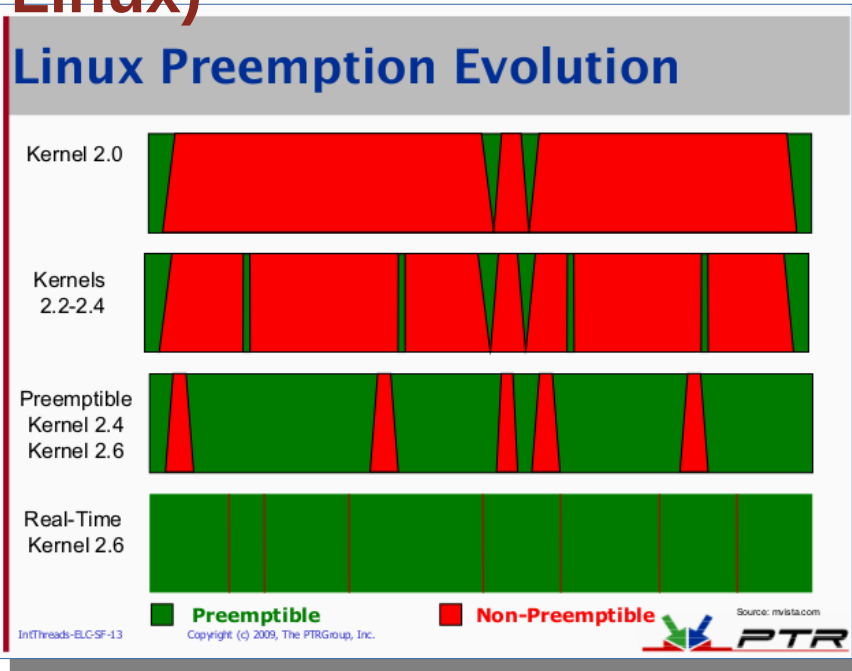
Converting vanilla (GPOS) Linux to an RTOS with RTL (Real-Time Linux)

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 interfaces

"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

RTL effort: Founder and project lead – **Thomas Gleixner**

Thomas – and collaborators - have been working to port regular (vanilla) Linux to an RTOS for many years, right from 2004!

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

Also, kernel preemption certainly isn't the only 'feature' of RTL, it has many more; the RTL patch is quite invasive and affects kernel code within the scheduling, timers (HRT), locking, interrupt handling (threaded interrupts!), PI, tracing, and more...

Leveraging the Linux CPU Scheduler

A new dawn – vanilla Linux can now be built to run as an RTOS !

“Something pretty historic (at least for geeks like us), happened this week of 16 September 2024; the Linux realtime effort, christened PREEMPT_RT and then Real-Time Linux (RTL), has finally – finally! – been fully merged into the kernel. ...”

- From my tech blog article ***‘The Linux kernel now with RTL fully merged’***, Sept 2024.

It took just 20 years ! :-)

A pic of Thomas Gleixner presenting the last printk-related PR – the one that completes the full inclusion of RTL – to Linus Torvalds at the Open Source Summit Europe in Vienna on 19 Sept 2024. The pull request was presented to Linus in hard-copy gold paper, tied with a ribbon! (That’s Thomas on the left, Linus on the right.)



Leveraging the Linux CPU Scheduler

Converting vanilla (GPOS) Linux to an RTOS with RTL

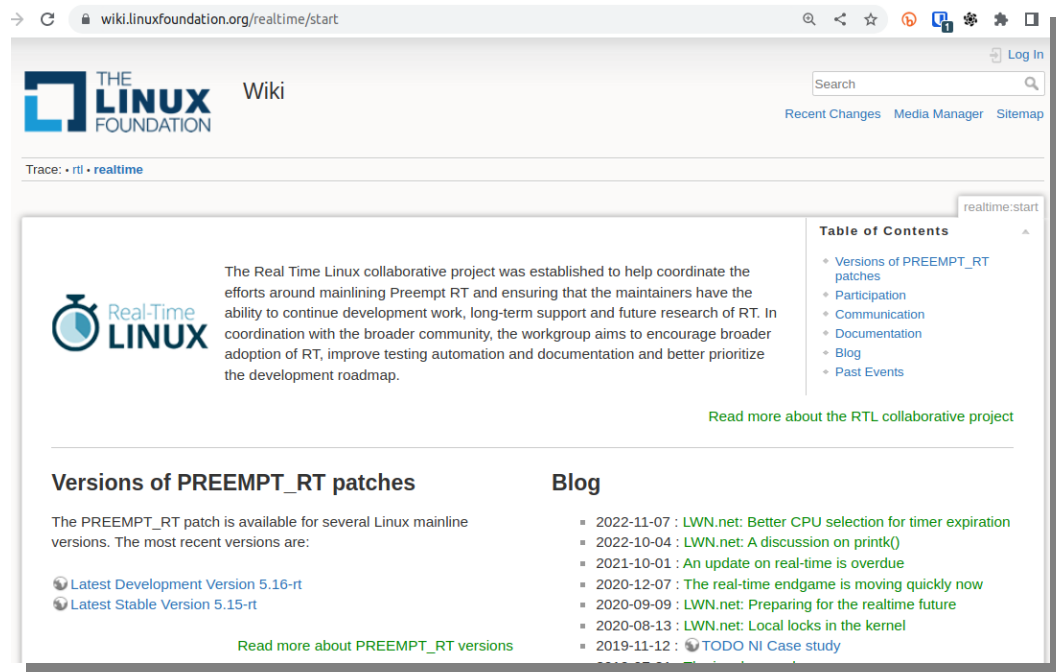
FAQs regarding RTL

Q. Is there documentation, a Wiki?

A. Yes indeed:

<https://wiki.linuxfoundation.org/realtime/start>

Also, the 'old' Wiki site is still very useful!



The screenshot shows a web browser displaying the Real Time Linux Wiki page at wiki.linuxfoundation.org/realtime/start. The page features the Linux Foundation logo and a search bar. The main content area includes a "Table of Contents" with links to "Versions of PREEMPT_RT patches", "Participation", "Communication", "Documentation", "Blog", and "Past Events". Below this, there is a section titled "Versions of PREEMPT_RT patches" which states that the patch is available for several Linux mainline versions and lists the "Latest Development Version 5.16-rt" and "Latest Stable Version 5.15-rt". A "Blog" section on the right lists recent articles from LWN.net and TODO NI, including topics like "Better CPU selection for timer expiration", "A discussion on printk()", "An update on real-time is overdue", "The real-time endgame is moving quickly now", "Preparing for the realtime future", "Local locks in the kernel", and "TODO NI Case study".

Leveraging the Linux CPU Scheduler

Converting vanilla (GPOS) Linux to an RTOS with RTL

FAQs regarding RTL

OUTDATED NOW !

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

A. No (*at least, not yet!*).

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/>

← → ↺ 🔒 mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/6.1/

Index of /pub/linux/kernel/projects/rt/6.1/

../	20-Feb-2023 16:49	-
incr/	16-Mar-2023 20:41	-
older/	16-Mar-2023 20:41	-
patch-6.1.19-rt8.patch.gz	16-Mar-2023 20:41	54K
patch-6.1.19-rt8.patch.sign	16-Mar-2023 20:41	833
patch-6.1.19-rt8.patch.xz	16-Mar-2023 20:41	48K
patches-6.1.19-rt8.tar.gz	16-Mar-2023 20:41	84K
patches-6.1.19-rt8.tar.sign	16-Mar-2023 20:41	833
patches-6.1.19-rt8.tar.xz	16-Mar-2023 20:41	67K
sha256sums.asc	16-Mar-2023 20:42	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 LTS kernel release, 6.1, is here:

<https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/6.1/patch-6.1.19-rt8.patch.xz>

Carefully note the precise version number of the RTL patch; it can *only* be applied upon that particular mainline kernel (in this example, 6.1.19).

Converting vanilla (GPOS) Linux to an RTOS with RTL

FAQs regarding RTL

Q. How can I check if RTL is enabled, within kernel/driver code paths?

A. `if (IS_ENABLED(CONFIG_PREEMPT_RT))`
 `[...]`

For detailed explanations on applying the RTL patch, reconfiguring and rebuilding the kernel, etc, using the popular Raspberry Pi, please 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 actually help?

latencytest – Benno
Sennoner; audio
sampling under load

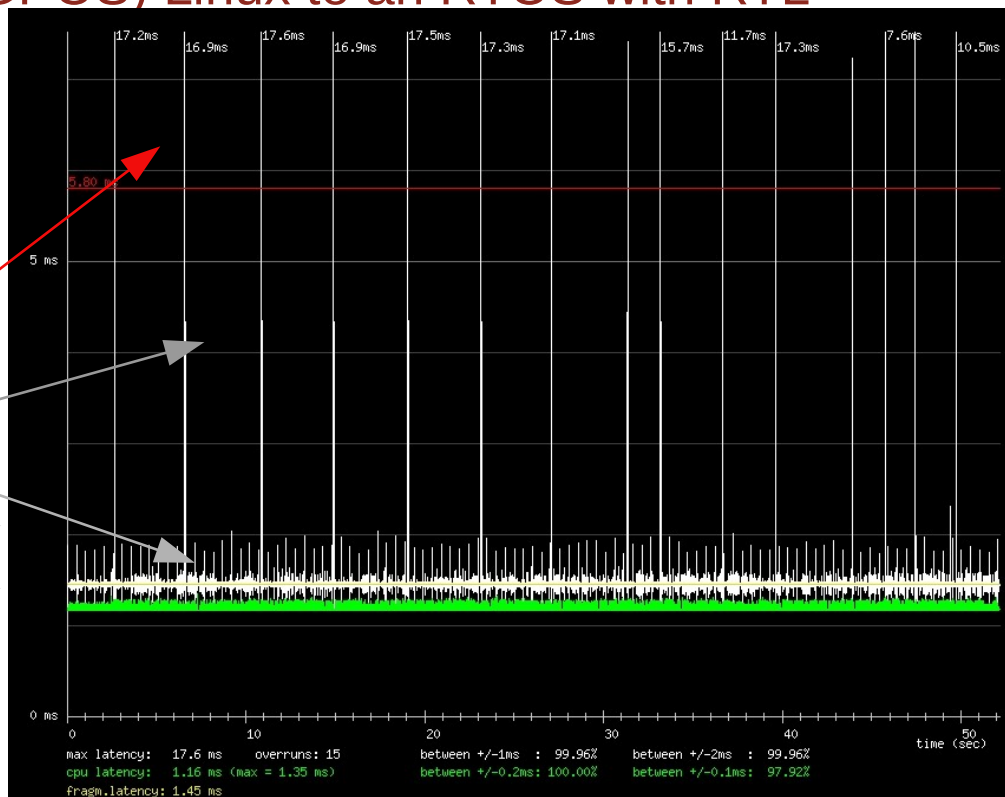
Graph - result of a
LatencyTest benchmark
on a Standard
(vanilla 2.6) kernel !

X-axis: time

Y-axis: the white vertical bars
represent the latency

This 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 the same
LatencyTest
Benchmark on a
**2.6 Preemptible Kernel
(CONFIG_PREEMPT)**

(I have to say it: WOW)

Notice that it's
excellent overall! Plus the
jitter (the variance) is now
indeed 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!)

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

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 the mlock[all](2) syscall to keep memory regions 'locked' – marked as non-swappable
- Can use the madvise(2) syscall to provide hints to the kernel regarding memory - sequential reads, read-ahead, etc
- From the older RTL Wiki site (Do note- several of these links are now marked as *Obsolete Content*; so, YMMV!):
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:

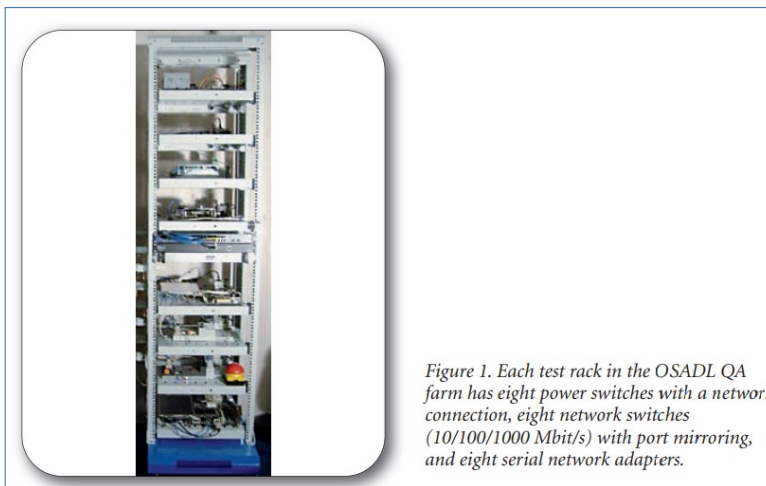
See the 'old' Wiki site (again, this is now marked as *Obsolete Content*):

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

OSADL test rack

Drones...

Factory-floor



Leveraging the Linux CPU Scheduler

All right, we're about done... BUT ...

There's more to learn (*a/ways!*), on Linux CPU Scheduling !

- CPU Sched tunables (/proc/sys/kernel/sched_*); see man page on proc(5)
- Visualizing process/thread execution (with LTTng and the TraceCompass GUI, trace-cmd, KernelShark GUI, ...)
- Kernel internals wrt Linux OS CPU scheduling
 - Modular scheduler classes, their algos (esp the implementation of their pick_next_task(), per-cpu runqueues, how & when is schedule() invoked, etc)
- Control Groups (Cgroups)
- Latency and it's measurement

(Watch out! shameless plug comin' up!) 😊

For all these topics – and much more! - do read/refer *Linux Kernel Programming, 2nd Ed.*

Leveraging the Linux CPU Scheduler

Thank you !

Q&A

Leveraging the Linux CPU Scheduler

Contact Info

Kaiwan N Billimoria

kaiwan@kaiwantech.com

kaiwan.billimoria@gmail.com

<https://amazon.com/author/kaiwanbillimoria>

[LinkedIn public profile](#)

My Tech Blog [please do follow!]

Corporate & Public Domain Training Courses: <https://kaiwantech.com>

<https://bit.ly/ktcorp>

[GitHub page](#)