

# 09: Real-time Linux

## Microcontrollers

Stefan Huber

Dept. for Information Technologies and Digitalisation  
FH Salzburg

Summer 2025

# Real-Time Linux

# Two types of operating systems

GPOS (general-purpose operating system):

- ▶ Attempt to cover a broad range of application fields, e.g., desktop, server.
- ▶ Optimized for the “average” case, the average throughput, ...
- ▶ Linux traditionally is strong in scaling: embedded systems, smart phones, entertainment systems, TV boxes, desktop, servers, high-performance clusters, ...
- ▶ Linux also is strong in running on a large variety of architectures, e.g., Alpha, Arm, HPPA, i386, amd64, ia64, m68k, mips, PowerPC, RISC-V, S390, x32, ...

RTOS (real-time operating systems):

- ▶ Specialized for real-time operation
- ▶ Typically runs on embedded systems

# A shift towards COTS and real-time Linux

Proprietary embedded systems<sup>1</sup> traditionally run typically either of two software stacks:


- ▶ **Bare metal**, i.e., without operating system, often on specialized hardware.
- ▶ **Special-purpose RTOS**, e.g., VxWorks or FreeRTOS.

## Issues

- ▶ **Porting software environments** (e.g., Python, Qt, or TensorFlow) can be between complex and unachievable.
- ▶ **Maintaining security** updates is troublesome. Having state-of-the-art security mechanisms is virtually unachievable.
- ▶ Leveraging **complex, contemporary computing platforms** is non-trivial. Operating a modern ARM processor bare-metal is unorthodox.<sup>2</sup>
- ▶ **Product strategy**: Costly, longer time-to-market, risky and costly transitions of hardware.

---

<sup>1</sup> Typically the hardware design also also proprietary. This requires hardware-competent software engineers and software-competent hardware engineers to a certain level.

<sup>2</sup> Booting a modern ARM processor with proprietary boot loaders or setting up the DRAM timing is delicate, especially without vendor support. 

# A shift towards COTS and real-time Linux

There is a shift towards COTS (commercial off-the-shelf) computing platforms:

- ▶ Cheaper and reduced time-to-market
- ▶ Increased computational power by complex multi-core processors
- ▶ Huge software ecosystem, well established tools in industry and academia
- ▶ Contemporary and elaborate security mechanisms

## Issue

They are traditionally **made for a GPOS** (general purpose OS), like Linux. But a GPOS focuses on high average performance but **not real time**.

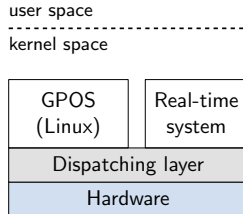
Two general approaches:

- ▶ **Single kernel** approach: Making Linux per se real-time capable.
- ▶ **Co-kernel** approach: Combining a (non-realtime) GPOS (e.g., Linux) with an RTOS.

# Co-kernel approach

Combine real-time system with a GPOS is via a **co-kernel** approach:<sup>3</sup>

- ▶ Real-time tasks are executed by an **additional RTOS**, called the co-kernel.
- ▶ An **intermediate layer** dispatches interrupts from hardware. Sometimes hypervisor techniques are employed here.

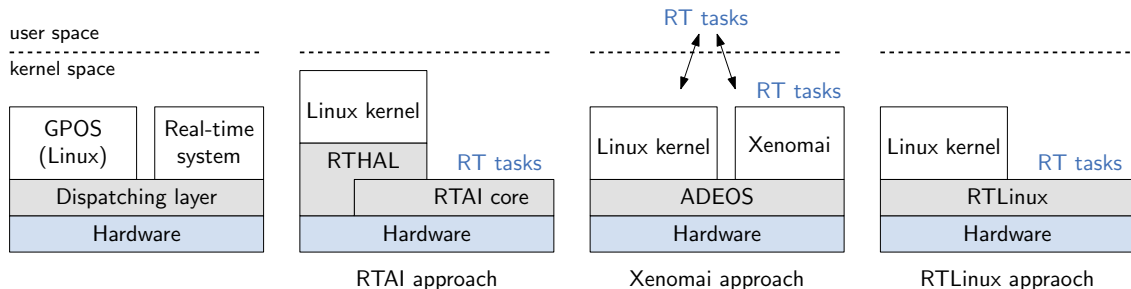


<sup>3</sup> Also called pico-kernel, nano-kernel or dual-kernel approach.

# Co-kernel approach

Combine real-time system with a GPOS is via a **co-kernel** approach:<sup>3</sup>

- ▶ Real-time tasks are executed by an **additional RTOS**, called the co-kernel.
- ▶ An **intermediate layer** dispatches interrupts from hardware. Sometimes hypervisor techniques are employed here.
- ▶ This approach is **popular in industry** and used by a couple of real-time Linux projects [RMF19].



<sup>3</sup> Also called pico-kernel, nano-kernel or dual-kernel approach.

# Co-kernel approach

## Advantage:

- ▶ Real-time is *easier* to achieve through special-purpose real-time subsystems.

## Disadvantage:

- ▶ RT tasks are *outside of Linux*, so the Linux advantage is partially lost.
- ▶ RT tasks often *in kernel space* rather than user space.
- ▶ *IPC* between kernels needs special care.

## Summary

We would wish for a single-kernel approach: a real-time capable Linux kernel. But this is *much* harder to achieve.



# PREEMPT\_RT patches

PREEMPT\_RT makes the Linux kernel real-time capable.

- ▶ High-resolution timers, IRQ threads, ...
- ▶ Preemptive kernel data structures

A set of patches first released<sup>4</sup> 2006, see [RMF19]

- ▶ After many years of finalizing, in mainline kernel since 2024-09-20 released with kernel 6.12.<sup>5</sup>

RT tasks  
-----  
user space  
kernel space

Linux + PREEMPT\_RT

Hardware

## Fully preemptive kernel

- ▶ If a low-priority task ends up in a non-preemptive kernel code then it can delay the execution of real-time tasks, which may then miss its deadline.
- ▶ Hence, we would like to minimize all non-preemptive codes in the kernel.
- ▶ Also improves responsiveness for desktop applications beyond real-time.

<sup>4</sup> On LWN in 2004, but work started in 1999. [Cor23]

<sup>5</sup> <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=baeb9a7d8b60b021d907127509c44507539c15e5>

# Linux scheduling classes

The Linux kernel provides different scheduler classes:

**SCHED\_NORMAL** By default a process is scheduled in this non-real-time class. Designed for mixed load, like batch processing, number crunching and desktop interaction.

**SCHED\_RR and SCHED\_FIFO** These are POSIX-compliant real-time classes with static priorities that take precedence over SCHED\_NORMAL. A running thread is preempted only by a thread of higher priority or by yielding the CPU voluntarily.<sup>6</sup>

**SCHED\_DEADLINE** A scheduler for the sporadic task model with deadlines. It implements a **global earliest deadline first** (GEDF) algorithm (with Constant Bandwidth Server (CBS)).

The SCHED\_RR and SCHED\_FIFO classes **do not consider deadlines** for real-time scheduling, but we have to choose the static priorities and design the tasks in a way to achieve a real-time system.

See details in `man sched`, [Oli18a] and [Oli18b].

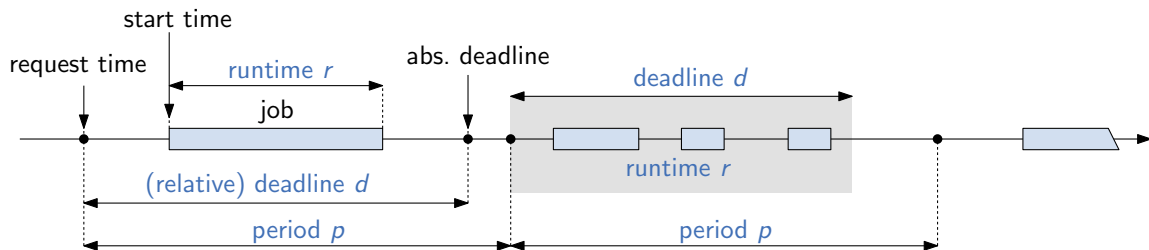
---

<sup>6</sup> SCHED\_RR also implements time-slicing for tasks of equal priority.

# The task model of SCHED\_DEADLINE

By means of `sched_setattr()` we set **runtime**  $r$ , **deadline**  $d$  and **period**  $p$ .

- ▶ Every  $p$  ns we spend  $r$  ns within the first  $d$  ns.
- ▶ It models a sequence of **jobs** with WCET at most  $r$  ns.



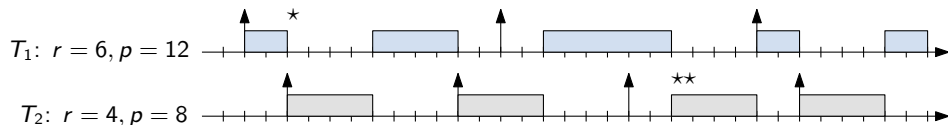
- ▶ The kernel performs a **schedulability test**<sup>7</sup> and requires  $0 < r \leq d \leq p$ .
- ▶ A thread calling `sched_yield()` marks the end of the current job and waits for the next period.

<sup>7</sup> On a single-core system it is necessary and sufficient. On a multi-core system it is only sufficient.

# EDF and bad jobs

Two tasks  $T_1$  and  $T_2$  with deadline equals period.

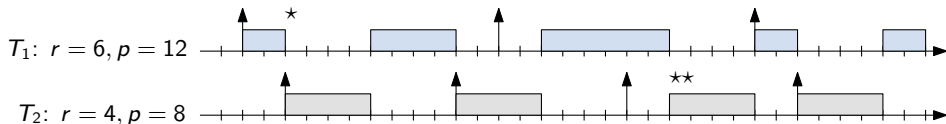
- ▶ At  $\star$  we preempt  $T_1$  and schedule  $T_2$ , whose deadline is earlier. At  $\star\star$  we keep running  $T_1$ , whose deadline is earlier.



# EDF and bad jobs

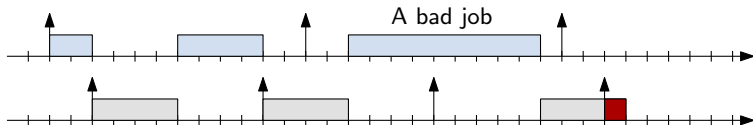
Two tasks  $T_1$  and  $T_2$  with deadline equals period.

- ▶ At  $\star$  we preempt  $T_1$  and schedule  $T_2$ , whose deadline is earlier. At  $\star\star$  we keep running  $T_1$ , whose deadline is earlier.



A bad job of  $T_1$  can force  $T_2$  to miss its deadline.

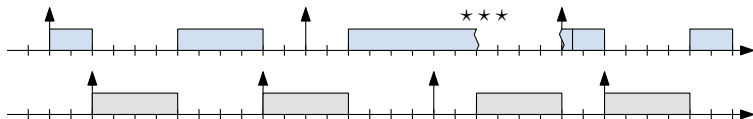
- ▶ A plain EDF is lacking **temporal isolation** between threads!



# Constant Bandwidth Server

SCHED\_DEADLINE implements EDF with [Constant Bandwidth Server](#) (CBS) algorithm.

- ▶ A job gets a “time budget” equal to its runtime parameter.
- ▶ If a job exhausted its budget, it is [throttled](#). That means, it is not scheduled until next period.
- ▶ This provides [temporal isolation](#) between threads.



# SCHED\_DEADLINE: Three application models

```
1 int main (int argc, char **argv) {
2
3     /* A task that runs 20ms within 100ms periods. */
4     struct sched_attr attr;
5     memset(&attr, 0, sizeof(attr));
6     attr.size = sizeof(attr);
7     attr.sched_policy = SCHED_DEADLINE;
8     attr.sched_runtime = 20000000;
9     attr.sched_deadline = attr.sched_period = 100000000;
10
11     int flags = 0;
12     if (sched_setattr(0, &attr, flags) < 0) {
13         perror("sched_setattr() failed");
14         return EXIT_FAILURE;
15     }
16
17     /* The actual work... */
18     work();
19     return EXIT_SUCCESS;
20 }
```

# SCHED\_DEADLINE: Three application models

```
1 /* Model: Periodic tasks */
2 void work() {
3     while(1) {
4         /* Cylic work. WCET of 20ms. */
5         periodic_job();
6         /* Notify scheduler about end of
7         * job. Only for real-time
8         * scheduling! */
9         sched_yield();
10    }
11 }
```



# SCHED\_DEADLINE: Three application models

```
1 /* Model: Periodic tasks */
2 void work() {
3     while(1) {
4         /* Cylic work. WCET of 20ms. */
5         periodic_job();
6         /* Notify scheduler about end of
7         * job. Only for real-time
8         * scheduling! */
9         sched_yield();
10    }
11 }
```

```
1 /* Model: Sporadic event-handling task. */
2 void work() {
3     while(1) {
4         /* Sleep until next event. */
5         blocking_wait_for_event();
6         /* After wakeup: React on event,
7         * e.g., process data, produce
8         * result. */
9         sporadic_job();
10    }
11 }
```

# SCHED\_DEADLINE: Three application models

```
1 /* Model: Periodic tasks */
2 void work() {
3     while(1) {
4         /* Cyclic work. WCET of 20ms. */
5         periodic_job();
6         /* Notify scheduler about end of
7          * job. Only for real-time
8          * scheduling! */
9         sched_yield();
10    }
11 }
```

```
1 /* Model: Sporadic event-handling task. */
2 void work() {
3     while(1) {
4         /* Sleep until next event. */
5         blocking_wait_for_event();
6         /* After wakeup: React on event,
7          * e.g., process data, produce
8          * result. */
9         sporadic_job();
10    }
11 }
```

```
1 /* Model: Aperiodic, bandwidth-guaranteed computing. */
2 void work() {
3     /* Just compute, without blocking or calls or yield. The CBS will throttle
4      * the thread after 20ms per period. */
5     while(1);
6 }
```

# Linux commands and tools

## chrt

Can change real-time attributes of processes, e.g., scheduling policies and their attributes.

## cyclictst

Performs timer latency tests with various different modes of operation.

- ▶ Part of the [rt-tests](#) suite [rt-tests], which also includes signaltest, fwlatdetect, and many more.
- ▶ Can set processor affinity or mlockall
- ▶ For different scheduling policies and priorities, with different timers, like nanosleep or POSIX timers.

## latencytop

For visualizing and debugging system latencies and their causes

- ▶ For kernel and user space

# Dependent tasks

In practice, real-time tasks are often dependent:

- ▶ Precedence of tasks: Task *A*'s job needs to be **done before** task *B*'s job can run, e.g., because it depends on *A*'s result.
- ▶ Mutual exclusion: Two or more tasks access the same **mutual-exclusive** resource.
  - ▶ A high-priority task is indirectly delayed by a low-priority task.
  - ▶ This can make a real-time task miss its deadline.

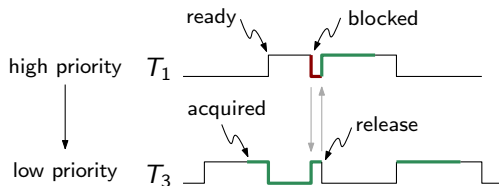
# Priority inversion

Mutual exclusion can lead to the phenomenon called **priority inversion**:

- ▶ A task of lower priority can effectively make a task of higher priority to wait.

Example:

- ▶ Tasks  $T_1$ ,  $T_2$ ,  $T_3$  with decreasing priority on a single processor.
- ▶  $T_3$  acquires a lock on a mutex first.  $T_1$  is blocked when acquiring the same mutex.  $T_3$  quickly releases the lock in favor of  $T_1$  in a well-designed system.



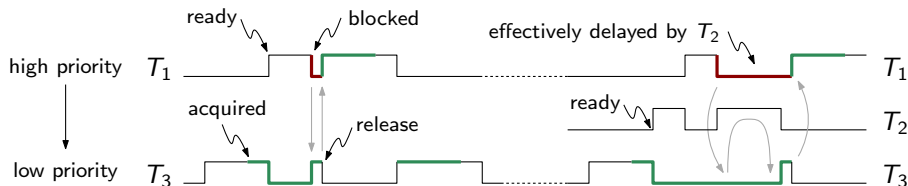
# Priority inversion

Mutual exclusion can lead to the phenomenon called [priority inversion](#):

- ▶ A task of lower priority can effectively make a task of higher priority to wait.

Example:

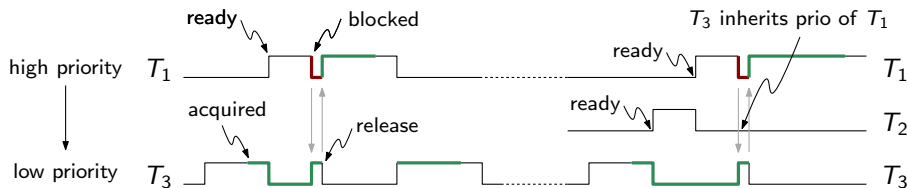
- ▶ Tasks  $T_1$ ,  $T_2$ ,  $T_3$  with decreasing priority on a single processor.
- ▶  $T_3$  acquires a lock on a mutex first.  $T_1$  is blocked when acquiring the same mutex.  $T_3$  quickly releases the lock in favor of  $T_1$  in a well-designed system.
- ▶  $T_2$  delays  $T_3$  from giving up the lock, which makes  $T_1$  to wait despite its priority: [Priority inversion](#).



# Priority inheritance

The Linux kernel implements **priority inheritance** to avoid priority inversion:

- ▶  $T_1$  is blocked because  $T_3$  holds the lock. Then  $T_3$  temporarily inherits the higher priority of  $T_1$ .
- ▶ So a mid-priority task  $T_2$  cannot delay  $T_3$  from releasing the lock.



# References I

- [Cor23] Jonathan Corbet. “The real realtime preemption end game”. In: *LWN.net* (Nov. 2023). URL: <https://lwn.net/Articles/951337/>.
- [Oli18a] Daniel Bristot de Oliveira. “Deadline scheduling part 1 — overview and theory”. In: *LWN.net* (Jan. 2018). URL: <https://lwn.net/Articles/743740/>.
- [Oli18b] Daniel Bristot de Oliveira. “Deadline scheduling part 2 — details and usage”. In: *LWN.net* (Jan. 2018). URL: <https://lwn.net/Articles/743946/>.
- [RMF19] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. “The Real-Time Linux Kernel: A Survey on PREEMPT\_RT”. In: *ACM Comput. Surv.* 52.1 (Feb. 2019), 18:1–18:36. DOI: 10.1145/3297714.
- [rt-tests] *A collection of latency testing tools for the linux(-rt) kernel*. URL: <https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/about/>.