Operating systems

Real-Time Scheduling

Created by
Enrico Fraccaroli
enrico.fraccaroli@gmail.com

Table of Contents

Real-Time Systems

Real-Time Systems

Definition

Real-Time Operating Systems

Definition

Definition (Real-Time Operating System)

A real-time operating system (RTOS) is a **time-bound** system which has well-defined, fixed **time constraints**.

We distinguish between:

- **Soft** RTOS: which can **usually** or **generally** meet a deadline;
- **Hard** RTOS: which can **deterministically** meet a deadline.

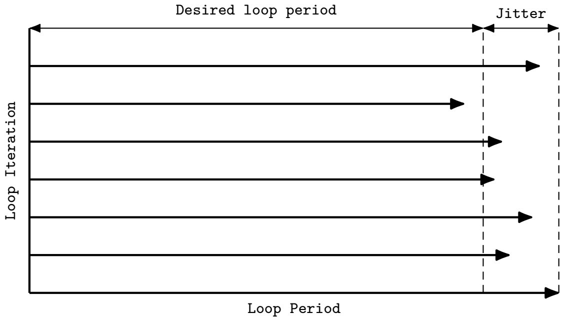
Furthermore, they are either:

1. **Event-driven**: system switches between tasks based on **priorities**;
2. **Time-sharing**: system switches tasks based on **clock interrupts**.

Real-Time Systems

Time consistency

In a RTOS, **consistency** over the amount of time it takes to **accept and complete** an application's task is of utmost importance. The variability of this time-span is called "*jitter*".



In **hard** RTOS, *jitter* is not acceptable, it destroys **determinism**.

Real-Time Policies

Real-Time Policies

In Linux there are three classes of processes (`linux/include/linux/sched.h`):

```
/// Scheduling Policies
#define SCHED_OTHER  0 ///< standard round-robin policy (time-sharing);
#define SCHED_FIFO   1 ///< a first-in, first-out policy (event-driven);
#define SCHED_RR     2 ///< a round-robin policy (event-driven).
```

Linux supports real-time scheduling **out of the box**.

P.S.: That's true, but the only issue is that **latencies** may not satisfy the hard real-time requirements of critical applications.

P.P.S.: If you look at the man page of `sched_setscheduler` system call, it will give you more details about these policies.

Real-Time Policies

Priority and Niceness

Real-Time Policies

Priority and Niceness (1/2)

Going back to what we saw with **MentOs**, each process has a `sched_entity` struct associated with it. Inside this struct we have the `prio` field, with values ranging from 0 to 139, explained as follows:

- 0 to 99 is the real-time “priority” range;
- 100 to 139 is the “niceness” range.

Both `SCHED_FIFO` and `SCHED_RR` have a `prio` ranging from 0 to 99. While `SCHED_OTHER`, has no actual “priority” value, but it has a “niceness” value ranging from 0 to 39 identified by a `prio` ranging from 100 to 139.

It may sound confusing, but to put it simple, we use the **same variable** to manage both **priority** and **niceness**, what changes is the **range**.

Real-Time Policies

Priority and Niceness (2/2)

Numeric Priority	Relative Priority	Tasks Nature	Time Quantum
0	Highest		200 *ms*
.	.	Real-Time Tasks	.
.	.		.
.	.		.
99	.		.
100 [*nice*: 0]	.		.
.	.	Other Tasks	.
.	.		.
.	.		.
139 [*nice*: 39]	Lowest		20 *ms*

Time quantum: the maximum amount of **contiguous CPU time** it may use before **yielding** the CPU to **another process** of the **same priority**.

Real-Time Policies

Preemption

Real-Time Policies

Preemption (1/2)

All runnable processes have entries in the *scheduler database*. The *scheduler database* is an array of 140 lists, **one list for each priority level**.

The scheduler **orders** the processes on each priority level list by placing the process that should:

- **run next**, at the **head** of the list;
- **wait the longest**, at the **tail** of the list.

Preemptive Priority Scheduler

The scheduler updates the *scheduler database*, whenever an event occurs. If **a process** in the database now has a **higher priority** than that of the **running process**, the running process is **preempted** and placed back into the *scheduler database*. Then, the **highest priority process** is made the **running** process.

Let us go back at the priority lists...

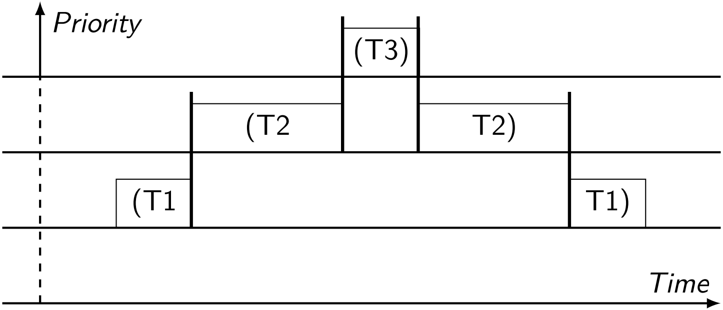
When a process is placed into a priority list in the scheduler database, it is placed at the **tail** of the list **unless it has just been preempted**.

If it has just been preempted, the processes scheduling policy determines whether it is inserted at the head (real-time scheduling policy) or the tail (timeshare scheduling policy).

Real-Time Policies

Policies Behaviour

A SCHED_FIFO process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, or it calls sched_yield.



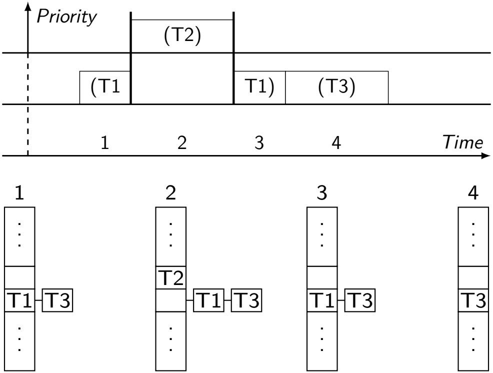
SCHED_RR is a simple enhancement of SCHED_FIFO, and the same rules of SCHED_FIFO are applied. However, each process is only allowed to run for a **maximum time quantum**.

We distinguish between two cases:

- If a SCHED_RR process has been running for a time period equal to or longer than the time quantum, it will be put at the **tail** of the list for its priority.
- A SCHED_RR process that has been preempted by a higher priority process and subsequently **resumes** execution as a running process will **complete** the unexpired portion of its round-robin time quantum.

Real-Time Policies

Behaviour SCHED_RR (2/2)



Implementation Steps in MentOs

Implementation Steps

Before implementing the real algorithm we need to extend the data-structures of MentOs, to manage the whole mechanism.

First, you need to get accustomed with the `list_head` data structure. It is used to **manage arrays** inside the kernel. The following **guide** contains the section *Kernel doubly-linked list*, which explains how the `list_head` works:
https://mentos-team.github.io/MentOS/doc/fundamental_concepts.pdf

These lists are required to build the 140 lists array of the scheduler.

Implementation Steps

Second, I would suggest checking what the `struct sched_entity` contains:

```
struct sched_entity {
    int prio; // priority
    time_t start_runtime; // start execution time
    time_t exec_start; // last context switch time
    time_t sum_exec_runtime; // overall execution time
    time_t vruntime; // weighted execution time
}
```

and how its fields are updated.

Implementation Steps

Third, I would suggest checking the content of `mentos/inc/process/prio.h`.

```
#define MAX_NICE +19
#define MIN_NICE -20
#define NICE_WIDTH (MAX_NICE - MIN_NICE + 1)

#define MAX_RT_PRIO 100
#define MAX_PRIO (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO (MAX_RT_PRIO + NICE_WIDTH / 2)

#define NICE_TO_PRIO(nice) ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio) ((prio)-DEFAULT_PRIO)

#define USER_PRIO(p) ((p)-MAX_RT_PRIO)

static const int prio_to_weight[NICE_WIDTH];
```

and check the `sys_vfork` function to see how the `new_process->se.prio` is initialized.

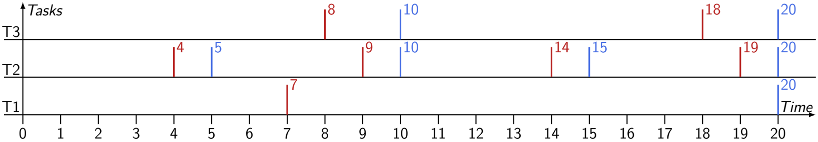
Backup Slides

Backup Slides

Earliest Deadline First (EDF)

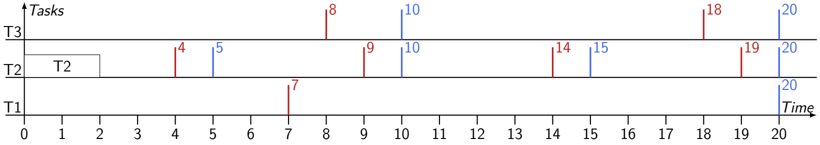
Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10



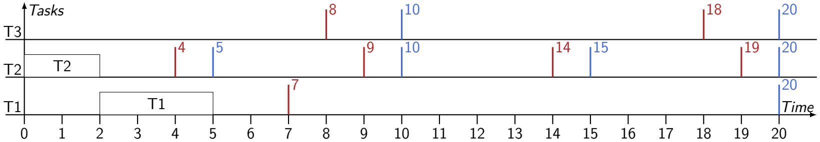
Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10



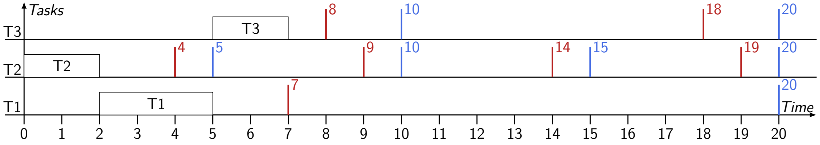
Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10



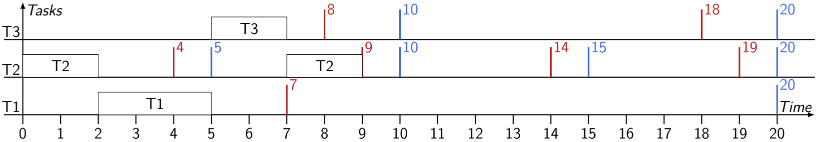
Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10



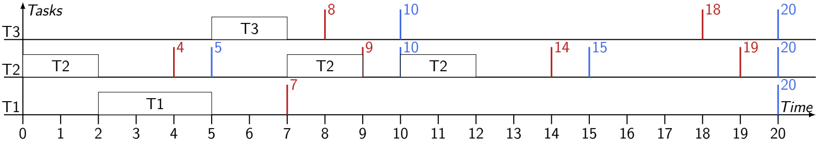
Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10



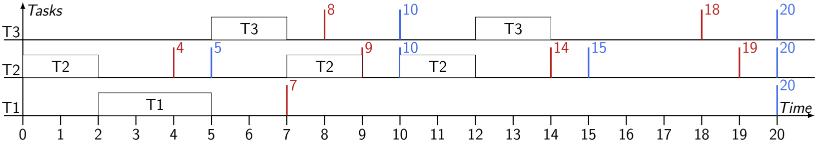
Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10



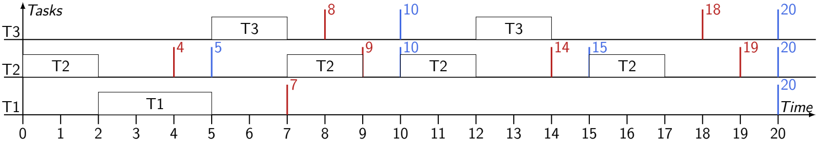
Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10



Earliest Deadline First (EDF)

	Burst Time	Deadline	Period
T1	3	7	20
T2	2	4	5
T3	2	8	10

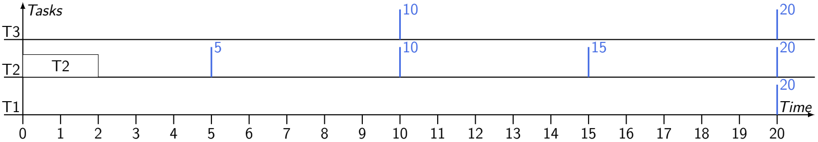


Backup Slides

Rate Monotonic (RM)

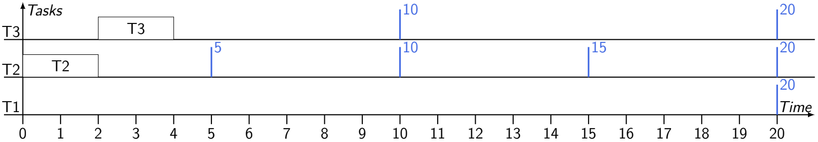
Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10



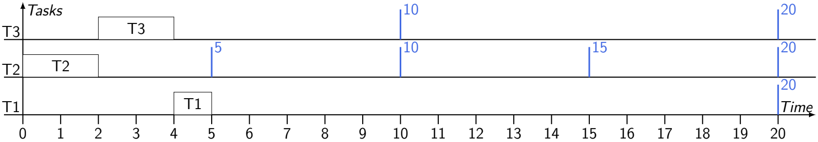
Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10



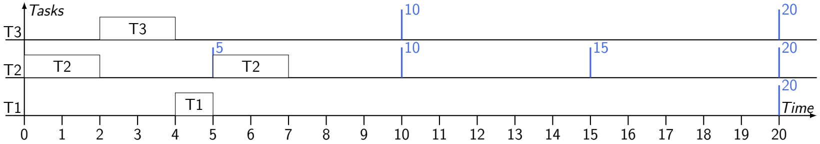
Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10



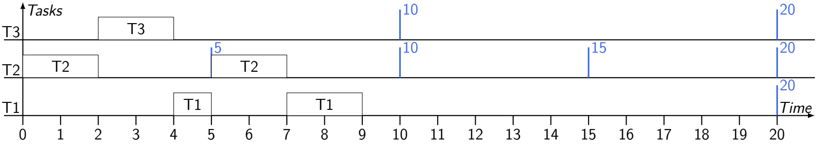
Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10



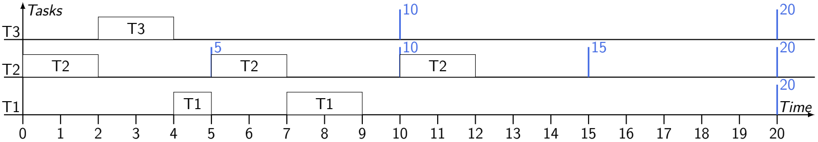
Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10



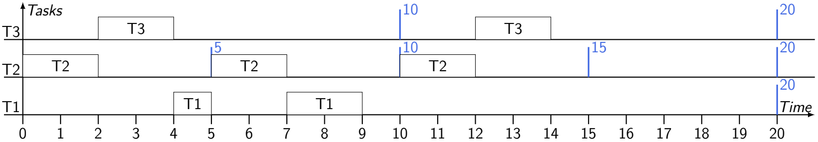
Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10



Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10



Rate Monotonic (RM)

	Burst Time	Period
T1	3	20
T2	2	5
T3	2	10