Kernel Linux Development: Rate Monotonic and Deadline Monotonic Task Scheduling Algorithms and their Implementation

Rafael Calai and Ricardo Mendes

Instituto Superior de Engenharia do Porto, ISEP
Master in Critical Computing Systems Engineering, MESCC
Real-time Operating Systems Programming, RTOSP

Abstract. The following work describes the addition of a new task scheduler to the Linux Kernel. The chosen algorithms are described in the work of C.L.Liu and J.Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," where *Rate Monotonic* and *Deadline Monotonic* are discussed.
Although the RM algorithm can be "forced" using the FIFO scheduler already in the kernel, the following pages will describe the reasoning, the process and the testing that took place during the implementation a new scheduler.

Keywords: Rate Monotonic, Deadline Monotonic, real-time, scheduler, Linux, kernel

1 Introduction

This work is one of the outcomes of the class assignment M2 from the Real-time Operating Systems Programming (RTOSP) discipline. A guided explanation was given during the classes on adding a new scheduling policy to the Linux kernel. In a step-by-step tutorial, the exercise implemented a simple scheduler that follows the LIFO algorithm - Last-In-First-Out.

Its simplicity removed the "burden" of understanding a complex algorithm and focused the attention on the kernel's structure and the necessary changes to add a new and personalized feature.

Using a linked-list to track the tasks and the agnostic characteristic of the algorithm when referring to the task's specifications - runtime, period, or deadline - meets the point of the previous statement.

For the assignment, we explore the RM and DM schedulers with the support of C.L.Liu and J. Layland article from 1973, "Scheduling algorithms for multiprogramming in a hard-real-time environment".

RM and DM are static-priority scheduling classes, meaning a task's priority will not change during runtime. Compared with dynamic scheduling classes, like Earliest Deadline First (EDF), the RM/DM have a more straightforward

implementation. This choice was secure but also enabled us to be more rigorous in its analyses when juxtaposed with the novelty and complexity of the kernel.

1.1 Notes

Before proceeding, some clarifications need to be made.

- The selection of the topic wasn't made without the analysis of other possibilities. The paper describing Constant Bandwidth Server (CBS)[3] was read and discussed as an alternative. However, its already integration in the scheduler policy Deadline of the kernel, made us consider. [1]
- Only the Rate Monotonic algorithm was implemented. However, given the similarity with the Deadline Monotonic, the paper will mention both.
- Unless stated otherwise, the periodic tasks have the same value for its deadline and its period.
- How the scheduler was added to the kernel follows the main guidelines given during the classes, and uses the programs - *tracer* - and the *Moker Framework* for it. Meaning most of the code will be found in the `./kernel/moker/` folder and not integrated with kernel's codebase.
- The LIFO scheduler remained implemented in the kernel. However, LIFO or RM cannot be enabled at the same time. Some constraints and configurations ensured this invariance.

  Conditional in the source code

```
#ifdef CONFIG_MOKER_SCHED_LIFO_POLICY
\\ code
#elif CONFIG_MOKER_SCHED_RM_POLICY
\\ code
#endif
```

  Kconfig

```
choice
    prompt "Select one scheduler"
    depends on MOKER_FRAMEWORK

    config MOKER_SCHED_RATE_MONOTONIC_POLICY
    bool"MOKER scheduling policy: RM"

    config MOKER_SCHED_LIFO_POLICY
    bool"MOKER scheduling policy: LIFO"
endchoice
```

[1] The presentation of SCHED DEADLINE during the Open IoT Summit gave us also an overview of the main challenges of dealing with aperiodic tasks when discussing real-time systems in a general purpose OS [4]

2 Rate and Deadline Monotonic

As already mentioned, both algorithms - RM and DM - are described in C.L.Liu and J.Layland article [1].

As described by the authors, the algorithms are **preemptive and priority-driven**, meaning that a task with a priority higher than the one currently in the CPU will interrupt it. The priority is defined by the task period - hence the rate in Rate Monotonic - or, in the case of DM, by the deadline.

In RM the priority is inversely proportional to the period: the shorter the period, the higher the priority. With DM, the higher priority is given by the shortest deadline.

2.1 Use cases

RM is used in hard real-time systems, where missing a deadline is considered a failure with critical and sometimes fatal outcomes. Predictability and determinism are essential in evaluating this type of system.

So, by dealing with critical systems, the algorithm must ensure that for a given set of periodic tasks, its schedulability is possible or not. For the calculation, the runtime of a hard-deadline task is given by the worst-case execution time (WCET) and, by doing so, ensures a runtime value that will in principle never be exceeded.

2.2 Schedulability

As proved in the article, the schedulability of a set of tasks can be determined by its CPU utilization.

The schedulability can be confirmed by calculating the Least Upper Bound (LUB) and comparing it with the utilization factor (U).

$$U(n) = \Sigma_{i=1}^{n} \frac{Ci}{Ti} \leq n(2^{\frac{1}{n}} - 1) \tag{1}$$

where C is the WCET and T the period.

The formula mentioned above represents the comparison made by the sum of each task's utilization time and the LUB for a n number of tasks.

It is worth noting that if the U surpasses the value of 1 - the CPU utilization exceeds 100% - the set is not schedulable.

Yet, in the case of an intermediate value of U,

$$1 \geq U(n) \geq LUB \tag{2}$$

the schedulability is undetermined.

In those cases, we resort to an iterative technique that will diverge or converge from a worst-case response time in a finite number of steps:

$$Rwci(0) = \Sigma_{k \in hp(i)} Ck + Ci \tag{3}$$

$$Rwci(m+1) = \Sigma_{k \in hp(i)} \left\lceil \frac{Rwci(m)}{Tk} \right\rceil * Ck + Ci \tag{4}$$

where *Rwc* is the worst case response time and *hp* is the highest priority task.

The equation 4 shows us some of the characteristics of the *critical instant* when dealing with preemptive and fixed priorities. *Synchronous release*, where all tasks are launched at the exact same moment, will mean a worst response time for the tasks, specially for the task with the least priority, usually more prone to miss their deadline.

For large sets, the LUB to processor utilization factor in RM is around 70%. This means that in the worst case, the CPU will be idle 30% of the time.

The article also points out that there are strong suggestions that make U=1 not feasible.[2].

2.3 Main assumptions

Before exploring the changes made in the kernel, we want to list some assumptions that are mentioned in the article [1] but were also defined by us during research and implementation.

The assumptions for the discussing of the topic are the following:

- All tasks have hard deadlines and are periodic, with constant intervals between requests;
- Each task must be completed before the following request occurs;
- All tasks are independent, meaning a task does not depend on each other. The completion or not of the previous task has no impact on the next one;
- Each task's duration is constant and does not vary with time. Here, the runtime of the task is given by the WCET;
- Any nonperiodic tasks in the system are special; they do not have hard deadlines and usually run outside the projected scheduling tasks as normal processes managed by the Operating System (OS).

Given the complexity of a real-life situation, there is a challenge in granting a stable periodicity of a request and a constant runtime.

> "Perhaps the most important and least defensible of these are that all tasks have periodic requests, and that runtimes are constant."[1]

That is why the mention of the WCET is used here instead of runtime, and that is why the existence of aperiodic tasks not defined in the set is expected and foreseen.

[2] In the real world, there is always some bandwith of the CPUs time that is used by processes, i.e. OS daemons

3 Scheduler Implementation

3.1 Previous analysis

The first step was to analyzed the changes made in the kernel during the implementation of the LIFO scheduler in the RTOSP classes. The objective was to better grasp the work's complexity. Figure 1 is one of the results of this analysis.

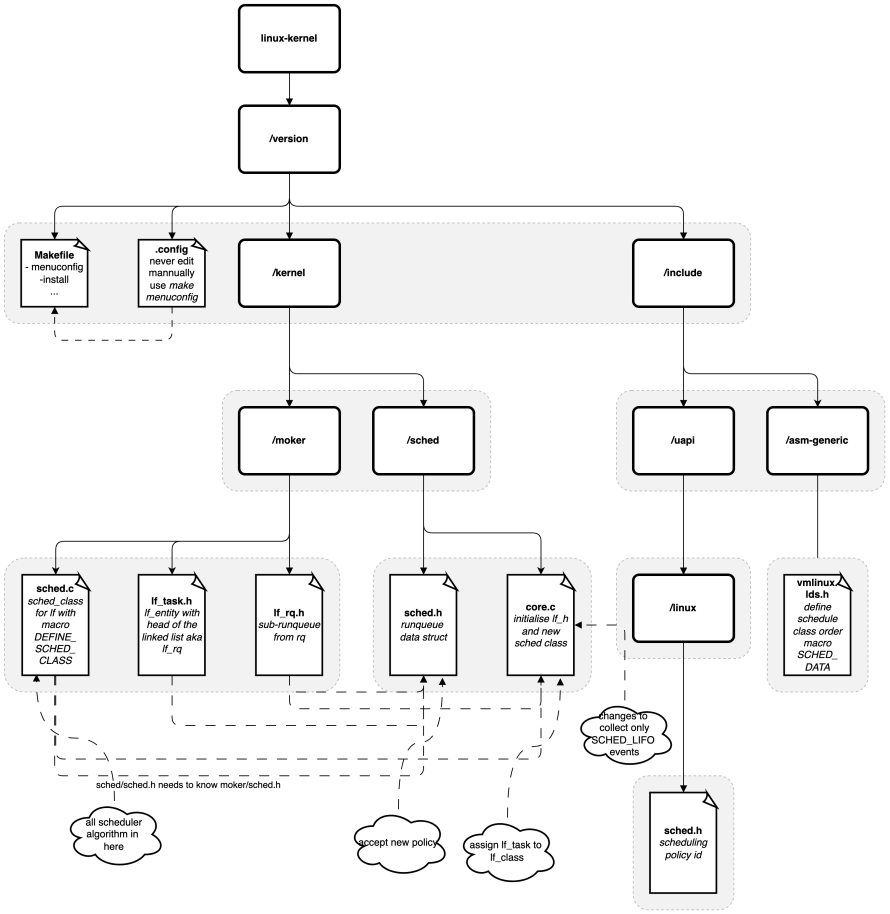

Fig. 1. Edit files during LIFO implementation

As we can see, most of the scheduler code is found in the `./kernel/moker/`, while the `./kernel/sched/sched.h` and `./kernel/sched/core.c` files are the

home of the kernels API that enables us to interact with the existing scheduling policies and data structures.

An effort was also made to understand the main data structures in play:

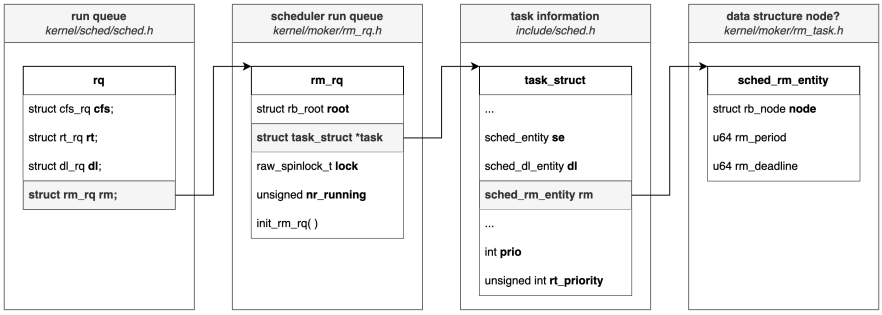

Fig. 2. Main data structures in RM implementation

Figure 2 gives already a glimpse of the implemented code. The visualization of the `struct`'s relation between each other helped us interacting with the `sched-class` API.

3.2 Implementation

RM Run Queue: The chosen data structure to organize the task by priority was the *red-black tree*. Although a linked-list, used in the LIFO exercise, would also do the job, we searched the kernel for a more adequate and optimized way to handle the information without unnecessary overhead.

Looking at the *SCHED_DEADLINE*, we found the `linux/lib/rbtree.c` file with the implementation of a red-black tree data structure [6].

The red-black tree is a particular case of the binary search tree and is more suitable for fast storage and retrieval of information [5]. Its main characteristic is the way that it balances itself. This is an essential feature because, in the case of binary trees, the performance is highly coupled to its shape. It is easy to imagine a binary tree that would look like a linked-list, and so, removing any advantage of this particular data structure.

The implementation of the *rbtree* in the Linux kernel also promoted its usage. Moreover, after reading the available documentation, the decision was final.

Red-Black Tree: The `RB_ROOT` macro is used to initializes the tree.

RM run queue initialization

```
void init_rm_rq(struct rm_rq *rq){
  rq->root = RB_ROOT;
  raw_spin_lock_init(&rq->lock);
  rq->task = NULL;
  rq->nr_running = 0;
}
```

As we can see, the macro is called during the RM run queue initialization.

The traversal of the tree is straightforward. Starting from the root and for each node, we check the position of a new node by comparing its value with the current node. If the new node has a higher value, we turn left; if not, we turn right. This repeats until we find the correct position to insert the new node.

Here is the implementation of the insert function:

```
void rm_rb_insert(struct rb_root *root, struct sched_rm_entity *entity) {
  __u64 period;
  struct rb_node **new = &(root->rb_node);
  struct rb_node *parent = NULL;

  period = entity->rm_period;

  /* Search place where to put new node */
  while (*new) {
    parent = *new;

    if(period < rb_entry(parent, struct sched_rm_entity, node)->rm_period) {
      new = &parent->rb_left;
    } else {
      new = &parent->rb_right;
    }
  }

  /* Add new node and rebalance tree. */
  rb_link_node(&entity->node, parent, new);
  rb_insert_color(&entity->node, root);
}
```

As visible in the function above, we jump from one node to another, starting with the tree's root. Since we are talking about RM, the task period we want to insert is compared to the one we are pointing to. If the new task's period is less, meaning higher priority, we jump to the next node on the left side. Otherwise, the jump is to the right side.

The last two calls in the function are part of the rb_tree API and help us add the new node to the tree and to rebalance the data structure.

Another two essential functions used during the implementation of the RM scheduler are the following:

```
rb_erase(&data->node, &mytree);
struct rb_node *rb_first(struct rb_root *tree);
```

The first function is self-explanatory and is used during the dequeue of a task. The node linked to the task is correctly deleted from the tree by calling this function.

The second function retrieves the leftmost node. In our use case, this means retrieving the task with the highest priority in the run queue.

Sched Class: Most of the code concerning the implementation of the algorithm can be found in one file, `./kernel/moker/sched_rm.c`.

By defining the implementation of the functions stated in the emphsched_class data structure, we will be able to have a fully functional scheduler.

In this assignement we focused primarily on two functions, the *enqueue_task* and the *dequeue_task*.

```
static void enqueue_task_rm(struct rq *rq, struct task_struct *p, int flags) {
  struct sched_rm_entity *entity = NULL;
  struct rb_node *left = NULL;

  raw_spin_lock(&rq->rm.lock);
  rm_rb_insert(&rq->rm.root, &p->rm);
  left = rb_first(&rq->rm.root);

  if (!left) {
    rq->rm.task = NULL;
  } else {
    entity = rb_entry(left, struct sched_rm_entity, node);
    rq->rm.task = container_of(entity, struct task_struct, rm);
  }

  rq->rm.nr_running++;
  add_nr_running(rq, 1);
  raw_spin_unlock(&rq->rm.lock);
}
```

The *enqueue_task* takes advantage of the *rm_rb_insert* function described above.

First we lock the resource that we want to access. Next we insert the new task in the *rbtree* based on its period. After that, the variable that points to the task with the highest priority must be updated.

At the end the lock is released.

The dequeue function is very similiar.

```
static void dequeue_task_rm(struct rq *rq, struct task_struct *p, int flags) {
  struct rb_node *left = NULL;
```

```
  struct sched_rm_entity *entity = &p->rm;

  raw_spin_lock(&rq->rm.lock);
  rb_erase(&entity->node, &rq->rm.root);

  left = rb_first(&rq->rm.root);

  if (!left) {
    rq->rm.task = NULL;
  } else {
    entity = rb_entry(left, struct sched_rm_entity, node);
    rq->rm.task = container_of(entity, struct task_struct, rm);
  }

  rq->rm.nr_running--;
  sub_nr_running(rq, 1);
  raw_spin_unlock(&rq->rm.lock);
}
```

In both cases, the variable that stores the pointer to the next highest priority is updated by calling the *rb_first* function.

3.3 Interaction with the Scheduler

After the implementation of the new scheduler, the programm that launched the tasks also needed to be changed.

Given the nature of the LIFO scheduler, the characteristics of a task didn´t have any weight in its priority.

Because of that, a different system call was used, `sched_setattr`. With this system call the scheduling policy and the attributes for a task can be sent to the scheduler.

Still in contrast with the LIFO exercise, the scheduler needs to know how to parse the sent information comming from the system call. This mapping, from `sched_attr` to the entity storing the task information, is made by the following function:

Mapper

```
void __setparam_rm(struct task_struct *p, const struct sched_attr *attr)
{
  struct sched_rm_entity *rm_se = &p->rm;
  rm_se->rm_runtime = attr->sched_runtime;
  rm_se->rm_deadline = attr->sched_deadline;
  rm_se->rm_period = attr->sched_period;
}
```

Only the **deadline** and the **period** of the task are used in the RM/DM scheduler algorithm, even do we only expect tasks with a deadline coincident with the period. Although unnecessary, the information about the **runtime** is also being sent.

During the testing of the RM implementation, the value of those three variables will be added as attributes. Below we can find a code snippet of the process.

task.c

```
...

unsigned long long C, T, O, time0, release;

...

task_id = atoi(argv[1]);
C = (unsigned long long)atoll(argv[2]);
T = (unsigned long long)atoll(argv[3]);
O = (unsigned long long)atoll(argv[4]) + OFFSET;
time0 = (unsigned long long)atoll(argv[5]);
njobs = atoi(argv[6]);

attr.sched_policy = SCHED_RM;
attr.sched_runtime = C;
attr.sched_period = T;
attr.sched_deadline = O;
attr.sched_util_min = 0;
attr.sched_util_max = 95;

if ((sched_setattr(getpid(), &attr, flags)) < 0)
{
  perror("ERROR:sched_setscheduler failed");
  exit(-1);
}

...
```

4 Testing and Comparation

In this section, we will test the RM implementation by launching a set of tasks and determine whether they are schedulable using the tools given by Liu and Layland and running them in our modified Linux kernel.

4.1 Task Set 1 - Theoretical

Table 1. Set 1

Task	Ci	Ti
1	0,5	3
2	1	4
3	2	6
4	2	7

For the task set above we will first calculate the utilization factor:

$$U(n) = \Sigma_{i=1}^{n} \frac{Ci}{Ti}; \tag{5}$$

$$U = \frac{0,5}{3} + \frac{1}{4} + \frac{2}{6} + \frac{2}{7} \approx 1,03$$

As we can see, the set is not schedulable because it surpasses the CPU total utilization bandwidth. It is also evident in the Gantt chart below that Task 4 will miss its deadline during a synchronous release.

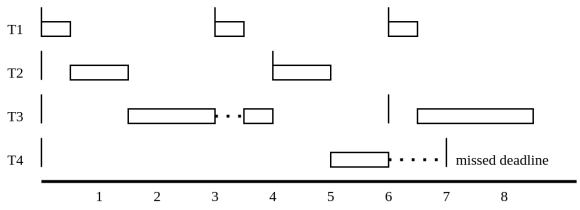

Fig. 3. Gantt chart - Set 1

4.2 Task Set 1 - Practical

Although we already known that the Set 1 is not schedulable, we will run it in our modified kernel and review the data collected.

Bash - launching the tasks

```
$ sudo ./launcher taskset1.txt
TASK:4 2000000000 7000000000 7000000000 3
```

```
TASK:3 2000000000 6000000000 6000000000 3
TASK:2 1000000000 4000000000 4000000000 3
TASK:1 500000000 3000000000 3000000000 3

...

cat /proc/moker_trace > trace1.csv
```

All tasks are being launched with the same time offset. This will simulate a *synchronous release* and run the worst-case scenario.

The output of the tracing mechanism is stored in a *.csv* file so we can better understand the behavior of the scheduler.

Some of the information was removed to facilitate the reading, including the timestamp.

Each line is formated as: **event, policy, pid, comm, obs**

The **policy** column indicates the scheduler, for example, 8 for our SCHED_RM and 0 for SCHED_NORMAL.

The **com** column indicates the process name, and the **obs** column some observations made by us to help the analysis.

Tasks are release at the same time:

```
...
ENQ_RQ,8,120,7558,1,task --> Enq task2
ENQ_RQ,8,120,7557,1,task --> Enq task3
ENQ_RQ,8,120,7559,1,task --> Enq task1
ENQ_RQ,8,120,7556,1,task --> Enq task4
SWT_AY,0,120,7616,0,PK-Backend
```

Task 1 starts running, confirming its highest priority:

```
SWT_TO,8,120,7559,0,task --> Runnning task1
DEQ_RQ,8,120,7559,1,task <-- Deq task1
SWT_AY,8,120,7559,1,task
SWT_TO,8,120,7558,0,task --> Runnning task2
DEQ_RQ,8,120,7558,1,task <-- Deq task2
SWT_AY,8,120,7558,1,task
```

Task 3 starts running, but is preempted before finishing:

```
SWT_TO,8,120,7557,0,task --> Runnning task3
ENQ_RQ,8,120,7559,1,task --> Enq task1
SWT_AY,8,120,7557,0,task --> Task3 preempted <--
SWT_TO,8,120,7559,0,task --> Runnning task1
DEQ_RQ,8,120,7559,1,task <-- Deq task1
SWT_AY,8,120,7559,1,task
```

```
SWT_TO,8,120,7557,0,task --> Runnning task3
DEQ_RQ,8,120,7557,1,task <-- Deq task3
SWT_AY,8,120,7557,1,task
SWT_TO,8,120,7556,0,task --> Runnning task4
ENQ_RQ,8,120,7558,1,task --> Enq task2
SWT_AY,8,120,7556,0,task !!! Task4 preempted !!!
SWT_TO,8,120,7558,0,task --> Runnning task2
DEQ_RQ,8,120,7558,1,task <-- Deq task2
SWT_AY,8,120,7558,1,task
SWT_TO,8,120,7556,0,task --> Runnning task4
ENQ_RQ,8,120,7559,1,task --> Enq task1
ENQ_RQ,8,120,7557,1,task --> Enq task3
SWT_AY,8,120,7556,0,task !!! Task4 preempted !!!
SWT_TO,8,120,7559,0,task --> Runnning task1
DEQ_RQ,8,120,7559,0,task <-- Deq task1
ENQ_RQ,8,120,7559,0,task --> Enq task1 all jobs had finished
DEQ_RQ,8,120,7559,128,task <-- Deq task1
SWT_AY,8,120,7559,128,task
SWT_TO,8,120,7557,0,task --> Runnning task3
SWT_AY,8,120,7557,0,task
```

Task 3 is preempted here by a kernels task:

```
SWT_TO,2,0,4176,0,rtkit-daemon !!! Task3 preempted !!!
SWT_AY,2,0,4176,1,rtkit-daemon
SWT_TO,8,120,7557,0,task --> Runnning task3
ENQ_RQ,8,120,7558,1,task --> Enq task2
SWT_AY,8,120,7557,0,task
SWT_TO,8,120,7558,0,task --> Runnning task2
DEQ_RQ,8,120,7558,0,task <-- Deq task2
ENQ_RQ,8,120,7558,0,task --> Enq task2 all jobs had finished
DEQ_RQ,8,120,7558,128,task <-- Deq task2
SWT_AY,8,120,7558,128,task
SWT_TO,8,120,7557,0,task --> Runnning task3
DEQ_RQ,8,120,7557,1,task <-- Deq task3
SWT_AY,8,120,7557,1,task
SWT_TO,8,120,7556,0,task --> Runnning task4
ENQ_RQ,8,120,7557,1,task --> Enq task3
SWT_AY,8,120,7556,0,task
SWT_TO,8,120,7557,0,task --> Runnning task3
DEQ_RQ,8,120,7557,0,task <-- Deq task3
ENQ_RQ,8,120,7557,0,task --> Enq task3 all jobs had finished
DEQ_RQ,8,120,7557,128,task <-- Deq task3
SWT_AY,8,120,7557,128,task
```

Task 4 already missing its deadline:

```
SWT_TO,8,120,7556,0,task --> Runnning task4
```

```
DEQ_RQ,8,120,7556,1,task <-- Deq task4
SWT_AY,8,120,7556,1,task
SWT_TO,0,120,13,0,rcu_preempt
ENQ_RQ,8,120,7556,1,task --> Enq task4
SWT_AY,0,120,6553,0,gnome-shell
...
```

To understand the tracer output, a Gantt chart was also plotted:

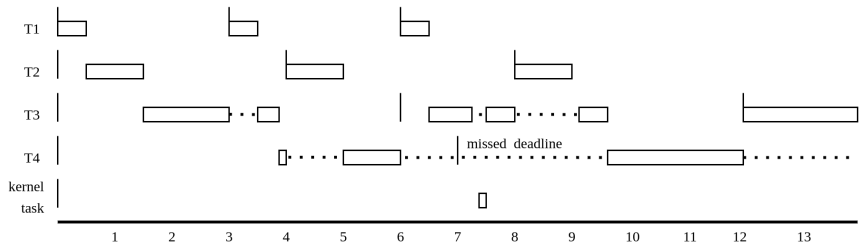

Fig. 4. Gantt chart - Set 1 running in the Linux kernel

As already expected, Task 4 missed its deadline. What is also interesting is the interaction of not-expected tasks during the process.

This highlights what already was mentioned in this report: the calculations do not consider aperiodic tasks related to the kernel's procedures. For example, Task 3 is preempted by a higher priority job, named *rtkit-daemon* and scheduled by the SCHED_RR policy.

Another interesting behavior is the start of Task 4 first job taking place before expected. This is because of the hardware peculiarities where the software was run. Tasks 1, 2 and 3 do not execute precisely the expected runtime. This give space for the CPU being fed by another task outside the ones that were started by us.

4.3 Task Set 2 - Theoretical

Table 2. Set 2

Task	Ci	Ti
1	0,5	3
2	1	4
3	2	6

For the next task set the utilization factor is below 1:

$$U = \frac{0,5}{3} + \frac{1}{4} + \frac{2}{6} \approx 0,74 \tag{6}$$

and when comparing it with the LUB of 3 tasks:

$$\begin{aligned} LUB(n) &= n(2^{\frac{1}{n}} - 1); \\ LUB &= 3 * (2^{\frac{1}{3}} - 1) \approx 0,78 \\ U &\leq LUB \Leftrightarrow 0,74 \leq 0,78 \end{aligned} \tag{7}$$

As we can see, the task set is schedulable. This can be also confirmed by drawing a Gantt chart:

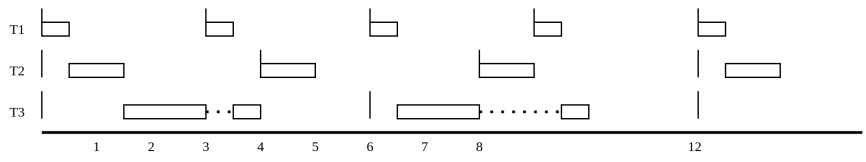

Fig. 5. Gantt chart - Set 2

4.4 Task Set 2 - Practical

For Set 2, we expect a different behavior even when taking some not-welcomed aperiodic tasks.

Bash - launching the tasks

```
$ sudo ./launcher taskset2.txt
TASK:3 2000000000 6000000000 6000000000 3
TASK:2 1000000000 4000000000 4000000000 3
TASK:1 500000000 3000000000 3000000000 3

...

cat /proc/moker_trace > trace2.csv
```

The format of the *.csv* file is the same as the one saw in the first set.

Here we show the tasks being added to the queue before, but because of the offset introduced, they are immediately dequeued:

```
ENQ_RQ,8,120,7760,0,task --> Enq task1 **Creating task on the system
DEQ_RQ,8,120,7760,1,task --> Deq task1
SWT_AY,8,120,7760,1,task
SWT_TO,0,120,8,0,kworker/u2:0
ENQ_RQ,8,120,7758,0,task --> Enq task3 **Creating task on the system
DEQ_RQ,8,120,7758,1,task --> Deq task3
SWT_AY,8,120,7758,1,task
SWT_TO,0,120,8,0,kworker/u2:0
ENQ_RQ,8,120,7759,0,task --> Enq task2 **Creating task on the system
DEQ_RQ,8,120,7759,1,task --> Deq task2
SWT_AY,8,120,7759,1,task
SWT_TO,0,120,8,0,kworker/u2:0
```

Synchronous release of all the tasks:

```
ENQ_RQ,8,120,7760,1,task --> Enq task1
ENQ_RQ,8,120,7758,1,task --> Enq task3
ENQ_RQ,8,120,7759,1,task --> Enq task2
SWT_AY,0,120,0,0,swapper
SWT_TO,8,120,7760,0,task --> Runnning task1
DEQ_RQ,8,120,7760,1,task --> Deq task1
SWT_AY,8,120,7760,1,task
SWT_TO,8,120,7759,0,task --> Runnning task2
DEQ_RQ,8,120,7759,1,task --> Deq task2
SWT_AY,8,120,7759,1,task
```

Task 3 is preempted by Task 1:

```
SWT_TO,8,120,7758,0,task --> Runnning task3
ENQ_RQ,8,120,7760,1,task --> Enq task1
SWT_AY,8,120,7758,0,task --> Task3 preempted <--
SWT_TO,8,120,7760,0,task --> Runnning task1
DEQ_RQ,8,120,7760,1,task --> Deq task1
SWT_AY,8,120,7760,1,task
SWT_TO,8,120,7758,0,task --> Runnning task3
DEQ_RQ,8,120,7758,1,task --> Deq task3
SWT_AY,8,120,7758,1,task --> Runnning other task **
SWT_TO,0,120,383,0,gmain
ENQ_RQ,8,120,7759,1,task --> Enq task2
SWT_AY,0,120,0,0,swapper
SWT_TO,8,120,7759,0,task --> Runnning task2
DEQ_RQ,8,120,7759,1,task --> Deq task2
SWT_AY,8,120,7759,1,task
SWT_TO,0,120,351,0,avahi-daemon --> Runnning other task **
ENQ_RQ,8,120,7760,1,task --> Enq task1
ENQ_RQ,8,120,7758,1,task --> Enq task3
SWT_AY,0,120,0,0,swapper
```

```
SWT_TO,8,120,7760,0,task --> Runnning task1
DEQ_RQ,8,120,7760,0,task --> Deq task1 all three jobs finished
ENQ_RQ,8,120,7760,0,task --> Enq task1 all jobs had finished
DEQ_RQ,8,120,7760,128,task --> Deq task1
SWT_AY,8,120,7760,128,task
SWT_TO,8,120,7758,0,task --> Runnning task3
ENQ_RQ,8,120,7759,1,task --> Enq task2
SWT_AY,8,120,7758,0,task --> Task3 preempted <--
SWT_TO,8,120,7759,0,task --> Runnning task2
DEQ_RQ,8,120,7759,0,task --> Deq task2 all three jobs finished
ENQ_RQ,8,120,7759,0,task --> Enq task2 all jobs had finished
DEQ_RQ,8,120,7759,128,task --> Deq task2
SWT_AY,8,120,7759,128,task
SWT_TO,8,120,7758,0,task --> Runnning task3
DEQ_RQ,8,120,7758,1,task --> Deq task3
SWT_AY,8,120,7758,1,task
SWT_TO,0,120,20,0,kcompactd0 --> Runnning other task **
ENQ_RQ,8,120,7758,1,task --> Enq task3
SWT_AY,0,120,0,0,swapper
SWT_TO,8,120,7758,0,task --> Runnning task3
DEQ_RQ,8,120,7758,0,task --> Deq task3 all three jobs finished
ENQ_RQ,8,120,7758,0,task --> Enq task3 all jobs had finished
DEQ_RQ,8,120,7758,128,task --> Deq task3
SWT_AY,8,120,7758,128,task
SWT_TO,0,120,431,0,in:imklog
```

Here is the Gantt chart:

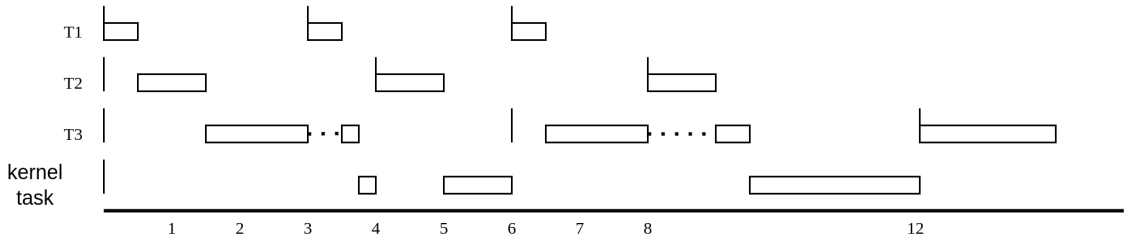

Fig. 6. Gantt chart - Set 2 running in the Linux kernel

The chart is almost a carbon copy of what was drawn during the theoretical chapter.

We observe now that when the tasks are not running, the kernel uses that time to run other processes.

5 Conclusions

First and foremost, the implementation of the RM scheduler policy was achived and proven.

During the process some optimizations could have been made.

One example is the use of a *cached rbtree*. This would speed up the calls to retrieve the leftmost node by replacing an expensive iteration through the tree for a pointer fetch [7]. SCHED_DEADLINE uses it in its implementation.

Another important observation is the dangerous overhead of running real-time systems in a general-purpose OS. The "rouge" processes impact the schedulability of a set, and when talking about hard deadlines, this type of OS looks not to be the best choice.

In light of the above observation, the SCHED_DEADLINE implementation in the Linux kernel looks to have its place between the other policies. The implementation of the Deadline scheduler is based on EDF and includes the implementation of a CBS to better deal with aperiodic tasks.

CBS helps to deal with aperiodic tasks by enforcing bandwidth isolation and by doing so, also isolating more important jobs from interference.

References

1. C.L.Liu, J.Layland: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the Association of Computing Machinery, Vol.20, Nº.1, (1973)
2. M. Kerrisk: The Linux Programming Interface, No Starch Press, (2010)
3. L. Abeni, G. Buttazzo: Integrating multimedia applications in hard real-time systems, Proceedings of the 19th IEEE Real-Time Systems Symposium, p.4, (1998)
4. The Linux Foundation. (2017, April 4). Using Sched-Deadline [video] YouTube. `https://www.youtube.com/watch?v=TDR-rgWopgM`
5. Wikipedia. `https://en.wikipedia.org/wiki/Rbtree`
6. The Linux Kernel Archives. `https://www.kernel.org/doc/Documentation/rbtree.txt`
7. The Linux Kernel Archives. `https://www.kernel.org/doc/Documentation/core-api/rbtree.rst`