POLITECNICO DI MILANO

Corso di Laurea Magistrale in Ingegneria Informatica

Comparative analysis of tools for experimenting with single- and multi-processor real-time scheduling

Relatore: Alberto Leva

Correlatore: Federico Terraneo

Tesi di Laurea di:

Matteo Locatelli, matricola 850070

Anno Accademico 2017/2018

Contents

List of Figures

Abstract

At the time of writing, the world of real-time applications is growing bigger at a very fast pace, and it doesn't concern anymore only the industrial, automotive and aerospace fields but has become much more pervasive. Because of this fact, real-time system simulators have become much more important, in order to have tools that are capable of giving meaningful and correct results before the deployment of the application in the "real world". This thesis focuses in particular in the field of the real-time schedulers and the most relevant simulators that can be currently found in the reasearch field. In the course of this paper the following real-time scheduling simulators will be analyzed:

- Cheddar
- RTSim
- Litmus^{RT}
- TrueTime

displaying their most important feature, how to use them and their advantages and disadvantages. The purpose of this thesis is to make some proposal in how these tools could be improved, in anticipation of some development or maybe the creation of a brand new simulation tool.

Italian abstract

In questo momento, il mondo delle applicazioni real-time sta crescendo molto velocemente e non interessa più solamente il campo industriale, automobilistico e aerospaziale, ma è diventato molto più pervasivo e diffuso. Proprio per questo motivo, i simulatori si sistemi real-time sono diventati molto più importanti, in modo da avere degli strumenti che possano dare dei risultati sensati e corretti prima dell'effettiva distribuzione dell'applicazione nel "mondo reale". Questa tesi si concentra in modo particolare nel campo degli scheduler real-time e dei simulatori più importanti che possono essere trovati nell'ambito della ricerca. In questa tesi i seguenti simulatori di scheduling real-time verranno analizzati:

- Cheddar
- RTSim
- Litmus^{RT}
- TrueTime

mostrando le loro caratteristiche principali, come usarli e i loro vantaggi e svantaggi. L'obiettivo di questa tesi è quello di proporre dei modi in cui migliorare questi strumenti, per un loro futuro sviluppo o per la creazione di nuovi strumenti di simulazione.

Chapter 1

1 Introduction

1.1 Real-time systems

The formal definition of a real-time system states that real-time systems are those systems in which the correctness not only depends on the result of the computation, but also on the time at which the results are produced. An example of a real-time system in the real world could be composed of sensors, a computer and actuators: the sensors have to read some values from the environment (i.e. temperature, humidity etc.), the computer processes those readings and send signals to the actuators that perform some predefined action. The important aspect of a real-time system like the one described above (and like many different others) is that the computer must compute the data in a limited amount of time, so that the actions made by the actuators are correct. The definition of real-time systems applies also to schedulers.

1.2 Real-time schedulers

Typically, the task of a scheduler is to assign different tasks that must be executed to the CPU of a computer, in order to complete the execution of those tasks; this means that the purpose of a scheduler is to guarantee the correct and complete execution of the tasks. Real-time schedulers have the same purpose of a non real-time scheduler, but with an additional contraint: the tasks must complete within a certain bound of time. Here follows a list of the terms used in order to fully understand the characteristics of real-time schedulers and tasks:

- Deadline: time limit within which a task must be complete after it is released.
- Release time: time at which a task is ready for execution.
- Priority: tasks with an higher priority must be executed before the ones with lower priority. It's often indicated by a integer number.
- Worst case execution time (WCET): maximum time that the task can use to complete execution.

- Periodic task: a task that activates to be executed at fixed intervals of time.
- Aperiodic task: a task that activates at irregular intervals of time.
- Sporadic task: a task that activates with minimal delay between two successive activations.
- Ready queue: task queue in which are placed the tasks that have already been released and that are ready for execution.
- Release queue: task queue in which are placed the tasks that waits for release.

Many real-time schedulers have been developed and are currently used, the most common are the following ones:

- Round robin (RR): time slices are assigned to each process that is ready for execution (i.e. is placed in the ready queue) in circular order and in order of release time. For example, if task 1, 2 and 3 are ready for execution and they have been released in that order, the first time slice is assigned to task 1, the next one to task 2, the next one to task 3 and then again to task 1. Round Robin is not always used as a real-time scheduler, it is used in the real-time field only if the periods and deadlines of the tasks are much greater than the time slices assigned to the task.
- Fixed priority (FP): every time the scheduler has to make a scheduling decision, it chooses the task with the highest priority among the tasks ready for execution.
- Earliest deadline first (EDF): when a scheduling event occurs, the scheduler chooses the task with the closest deadline among the tasks ready for execution
- Deadline monotonic (DM): tasks are assigned priorities according to their deadline, the closest the deadline the highest the priority. Once assigned a priority to every ready task, the deadline monotonic scheduler works like a fixed priority one. The main difference between EDF scheduler and DM scheduler is that, with DM the scheduling decisions are made considering

the priority which is fixed once assigned, while with DM the scheduling decisions are based on the deadline that can change over time (that's the reason why the EDF is considered a dynamic scheduler).

The terms «scheduling decision» and «scheduling event» refer to the fact that periodically the scheduler takes action to decide which task must be executed by the CPU, accordingly to the scheduling policies as defined above. If there is a task ready for execution that must be run because has an higher «urgency» (that is the priority in case of FP and DM or the deadline in case of EDF) with respect to the running task, the running task is preempted (that is, its execution is suspended and the CPU is no longer assigned to it) and the ready task with the higher «urgency» is run. For example, in the case of a FP scheduler, when a scheduling decision occurs, the scheduler checks if in the ready queue there is a task with an higher priority with respect to the task that is currently running; if there is one such task the scheduler makes a preemption, suspending the execution of the running task and starting the execution of the task in the ready queue with the higher priority.

The scheduling events can be periodic or can be based upon events, this leads to the definition of a kernel using ticks and tickless kernel:

- Kernel with ticks: in this case the scheduling events occurs when a tick happens, that is when the system timer managed by the kernel activates. For example, if the system timer is set to 10 ms, every 10 ms there is a tick, so the scheduler activates and make a scheduling decision. This approach has two main downsides:
  - if the interrupts of the system timer are sent with a too low frequency, the scheduler performances are not optimal, because a lot of time are wasted waiting for the scheduler to make a decision.
  - if the interrupts are sent with a too high frequency, a lot of time is wasted because the scheduler intervenes even when there is no need of a scheduling decision and the scheduler uses computer resources that could be used for task execution instead.

- Tickless kernel: in this case the scheduling decisions are made when an event occurs (for example a task finishes its budget for execution) and not when a fixed timer sends an interrupt. Using a tickless kernel negates the tradeoffs of using ticks, because there isn't anymore a fixed interval at which scheduling events occurs, but the interval can change accordingly to the needs of the situation.

1.3 Overview

This thesis is organized as follows.

In chapter 2, the main tools today available for the simulation of real-time scheduler are examined, especially LitmusRT and TrueTime. First, an overview of the simulation tool is given, then follows a description of the scheduler available and how they are implemented. Lastly, the advantages and disadvantages of each tool are listed and described. In the TrueTime section there is a description of the changes made to the code of the kernel of the simulator, in order to separate the scheduler logic from the kernel and to implement the round robin as an additional scheduler available.

In chapter 3, the conclusions are made, wrapping up what is described in the paper and listing possible future reaserch topics.

Chapter 2

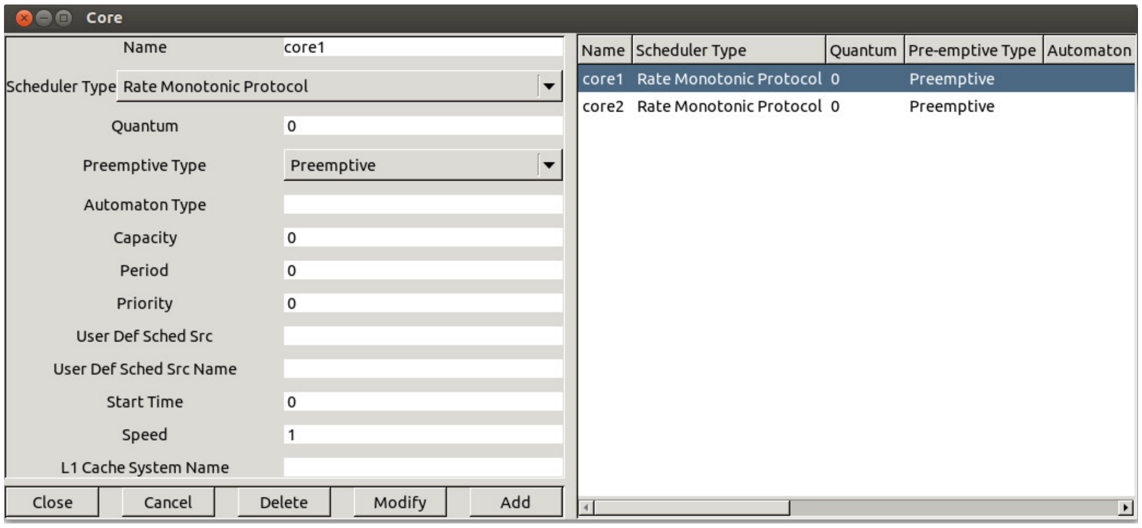
2 Tools for scheduler simulation

In order to assess the correctness and efficiency of a real-time scheduler (and more in general of a real-time system) it is important to simulate its behavior while it runs. While the simulation of a system is important in many scopes, it's particularly significant with real-time systems in order to test them in an ideal condition, because verifying thei correctness when already deployed could be misleading. During the run time execution of a real-time system (and scheduler) could occurs many events that can make the extraction of significant data really hard or even impossible. If we think about a simple real-time system composed, for example, of accelerometers for drone control or encoders to control the position of a robot, once deployed some interferences and disturbances could alter the readings and the developer who created that system cannot know if the wrong readings are caused by interferences, by sensors faults or other reasons. The same happens for real-time schedulers as well: making the testing phase with the scheduler already implemented and deployed could be difficult because often there is no control over the release and creation of new real-time tasks and maybe the behavior of those tasks is unknown. This shows that, when developing a new real-time scheduler, it's important to simulate its behavior to understand if in the best conditions it works as intended. Another important step in the simulation is to also include some disturbances that could affect the behavior of the scheduler, in order to test its performances and check its robustness.

2.1 Cheddar

Cheddar is a real-time scheduling simulation tool developed by a team from the University of Brest. Cheddar has two main features: an editor to model real-time systems that has to be analyzed and a framework to perform the analysis.

In Cheddar, an application consists of a set of processors, buffers, shared resources, messages and tasks which are defined by three parameters: deadline, period and capacity (that express the units of time of the budget the task has every time it is executed). In order to make a scheduling simulation all the features and attributes of the elements listed above must be set. Firstly, a core must be defined using the window showed below



Name	core1
Scheduler Type	Rate Monotonic Protocol
Quantum	0
Preemptive Type	Preemptive
Automaton Type	
Capacity	0
Period	0
Priority	0
User Def Sched Src	
User Def Sched Src Name	
Start Time	0
Speed	1
L1 Cache System Name	

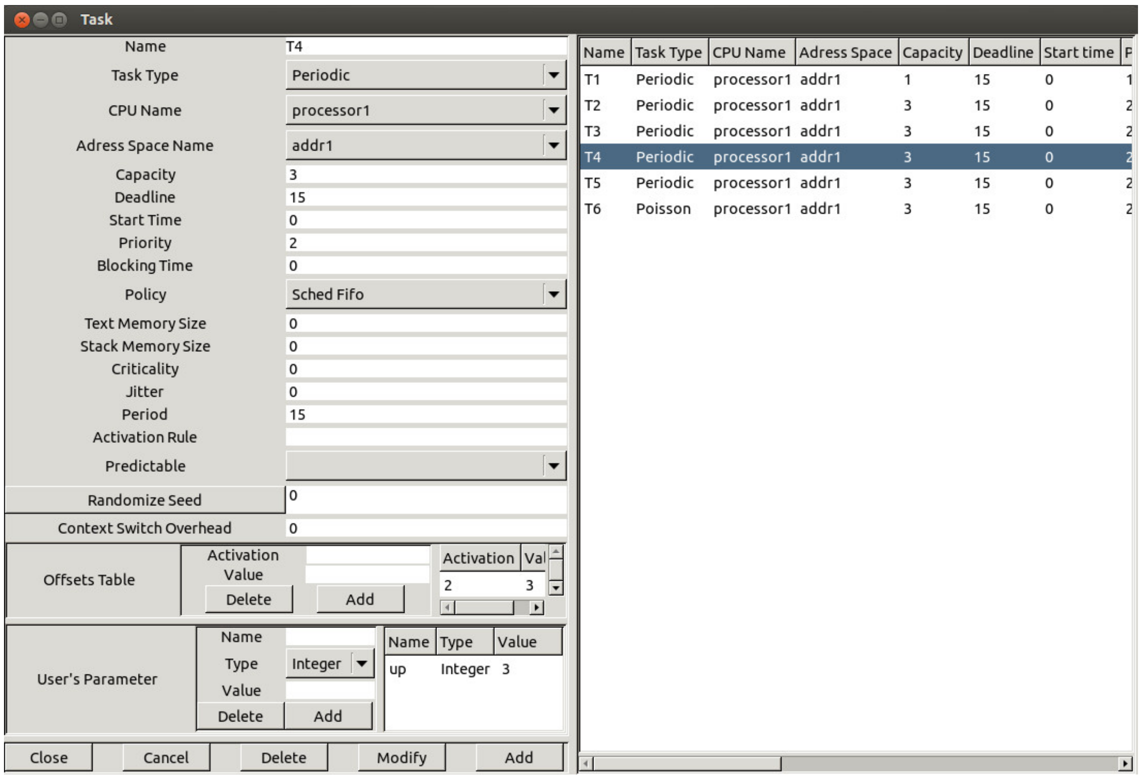
Name	Scheduler Type	Quantum	Pre-emptive Type	Automaton
core1	Rate Monotonic Protocol	0	Preemptive	
core2	Rate Monotonic Protocol	0	Preemptive	

Figure 1: Core definition in Cheddar

The most important fields that we can find in this window are the following:

- Name: the name of the core.
- Scheduler Type: the scheduler used by the core. The user can choose among a set of implemented schedulers that includes
  - Earlies Deadline First
  - Least Laxity First: tasks are scheduled according to their laxity
  - Rate Monotonic
  - Deadline Monotonic
  - Round Robin
  - POSIX 1003.1b: tasks are scheduled according to their priority and the policy of the scheduler, which defines the queueing pattern of the tasks.
- Preemptive type: if the scheduler use preemption or not.

After the definition of the core, a processor must be created and one or more of the cores previously defined have to be added to it. Also, a definition of an address space is needed; an address space models a memory which contains tasks, buffers and shared resources. After the creation of those elements, a task must be defined using the following window



Task

Name	T4
Task Type	Periodic
CPU Name	processor1
Adress Space Name	addr1
Capacity	3
Deadline	15
Start Time	0
Priority	2
Blocking Time	0
Policy	Sched Fifo
Text Memory Size	0
Stack Memory Size	0
Criticality	0
Jitter	0
Period	15
Activation Rule	
Predictable	
Randomize Seed	0
Context Switch Overhead	0

Offsets Table

Activation	Val
2	3

User's Parameter

Name	Type	Value
up	Integer	3

Name	Task Type	CPU Name	Adress Space	Capacity	Deadline	Start time	P
T1	Periodic	processor1	addr1	1	15	0	1
T2	Periodic	processor1	addr1	3	15	0	2
T3	Periodic	processor1	addr1	3	15	0	2
T4	Periodic	processor1	addr1	3	15	0	2
T5	Periodic	processor1	addr1	3	15	0	2
T6	Poisson	processor1	addr1	3	15	0	2

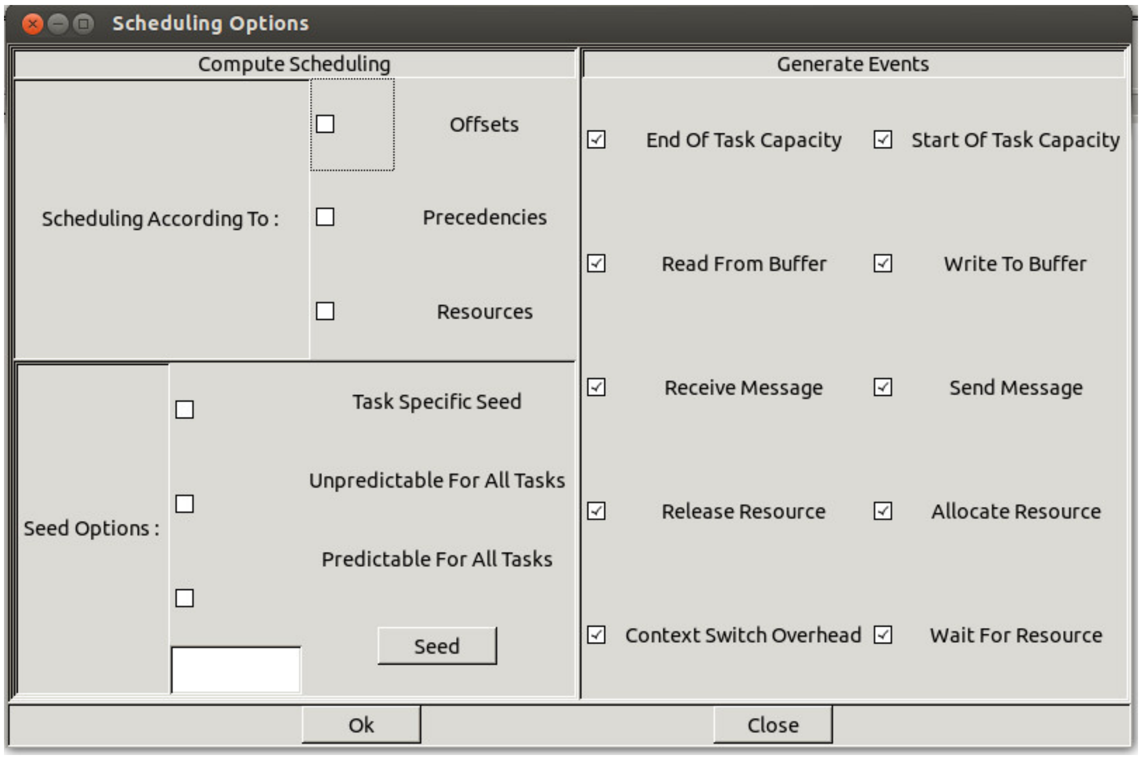
Figure 2: Task definition in Cheddar

The fields that can be found here and that must be filled are:

- Name: the name of the task.
- Task type: describes the type of task that is created. It can be
  - Aperiodic
  - Periodic
  - Sporadic
  - Poisson process: the task is activated many times and the delay between activations is random; the law used to generate these delays is a Poisson (exponential) one
- CPU name: the processor on which the task will run.
- Address space name: the address space in which the task will be placed.

- Capacity: bound on execution time of the task.
- Deadline: the deadline of the task.
- Start time: time of the first activation of the task.
- Priority: the fixed priority of the task.
- Blocking time: bound on a shared resource waiting time.
- Policy: defines how a task is chosen when many tasks have the same priority.
- Text memory size and stack memory size: size of the resources (text memory and stack memory) used by the task during simulation.
- Criticality level: defines how much a task is critical, that is the need to complete the task as soon as possible.
- Jitter: maximum delay of the wake up time of the task.
- Period: time between task activations, that is the time that passes between the moment at which the task activates and the next activation.
- Activation rule: rule that specifies what a task must do when activates, this rule is user defined.
- Predictable/Unpredictable: in case of a poisson process task, if «predictable» is selected the value in the seed field is used to generate the delay of activations time, if «unpredictable» is selected the seed is computed at simulation time.
- Context switch overhead: defines the overhead used when a context switch happens.
- Offsets table: defines the wake up time of a certain activation of the task.
- User's defined parameters: this table contains user defined parameters that are used by user defined scheduler.

When all the tasks are defined and added to the task pool, the scheduling options can be defined using the following window:



Compute Scheduling

Scheduling According To :

☐ Offsets
☐ Precedencies
☐ Resources

Seed Options :

☐ Task Specific Seed
☐ Unpredictable For All Tasks
☐ Predictable For All Tasks

Generate Events

✓ End Of Task Capacity ✓ Start Of Task Capacity
✓ Read From Buffer ✓ Write To Buffer
✓ Receive Message ✓ Send Message
✓ Release Resource ✓ Allocate Resource
✓ Context Switch Overhead ✓ Wait For Resource

Figure 3: Cheddar scheduling options

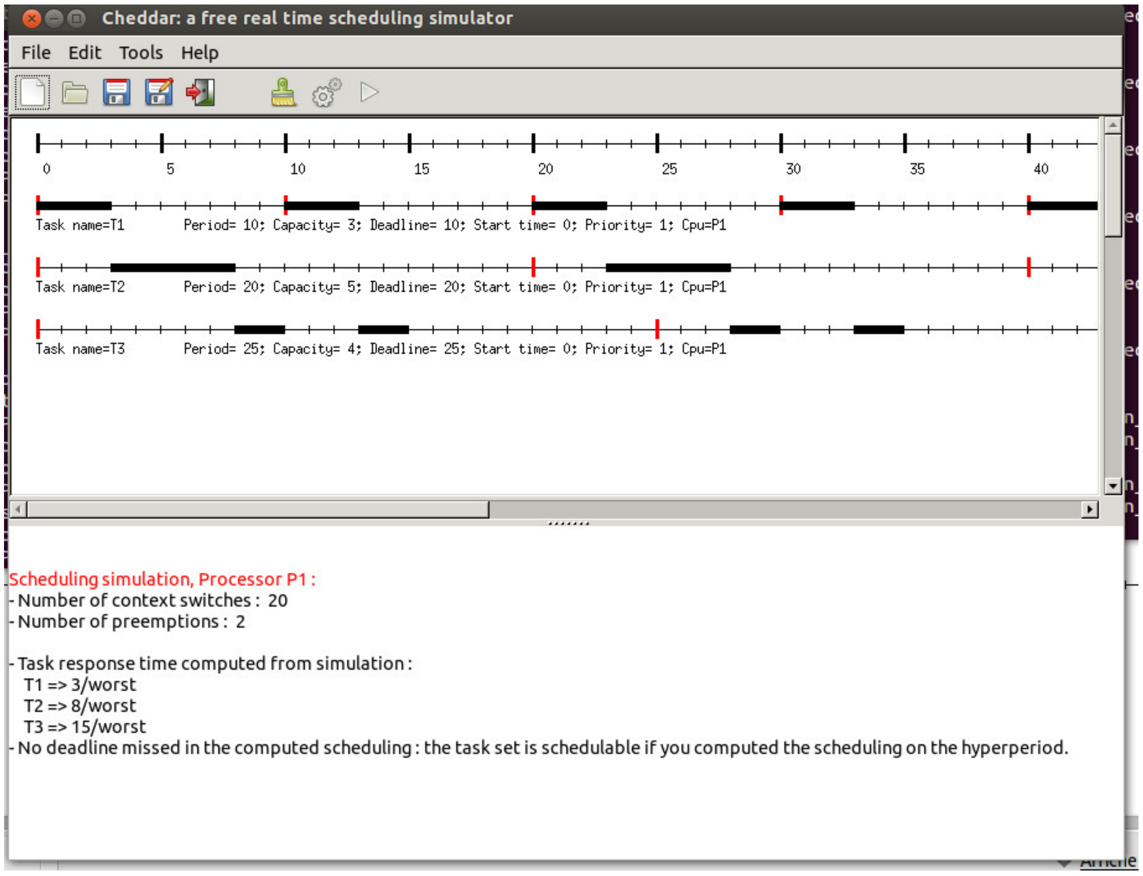
The options that can be customized with this window are:

- Offsets: the simulation considers the offsets defined by the user when the tasks placed in the task pool are created.
- Precedencies: the simulation takes care of the dependencies between tasks during the simulation (these dependencies can be defines with a dedicated window)
- Resources: if checked, the access to shared resources by tasks are simulated.
- Seed options: using these options the user can define how random activation timer are generated.
- Generate events: with these checkbox the user can decide which events will be generated in the event table during the simulation.

When all the necessary settings are done, one of the two of the analysis tool provided by Cheddar can be used:

- Feasibility analysis tool, that computes the feasibility of the defined tasks without the actual scheduling of the tasks.
- Simulation analysis tool, that computes the scheduling of the tasks and shows the scheduling diagram and some information regarding the scheduler simulation.

Speaking of the simulation analysis tool, when the simulation is run a window containing all the information regarding the scheduling simulation is shown:



```
Task name=T1    Period= 10; Capacity= 3; Deadline= 10; Start time= 0; Priority= 1; Cpu=P1
Task name=T2    Period= 20; Capacity= 5; Deadline= 20; Start time= 0; Priority= 1; Cpu=P1
Task name=T3    Period= 25; Capacity= 4; Deadline= 25; Start time= 0; Priority= 1; Cpu=P1
```

```
Scheduling simulation, Processor P1 :
- Number of context switches :  20
- Number of preemptions :  2

- Task response time computed from simulation :
  T1 => 3/worst
  T2 => 8/worst
  T3 => 15/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the hyperperiod.
```

Figure 4: Cheddar simulation results

In the upper part of the window, the simulation diagram is shows.

Intuitively, the first row represents the simulation time. Below that, there is a row for each task that has been inserted to the task pool. The vertical red lines represents the wake up times of the task, while the black rectangles represents the execution time of the task.

The lower part of the window shows some interesting acts about the simulation, such as the number of context switches and preemptions and, above all, if the task set is schedulable or not.

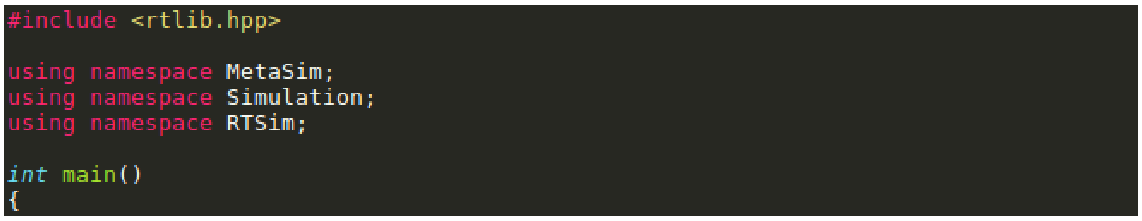
One of the main advantages in using Cheddar is that is very flexible: there are a lot of options that can be used to customize the simulation (many of them are not even shown in the pictures above), it also allows to simulate the access to shared resources like memories and buffer which is not a common feature for real-time simulators. Also, using the graphical user interface to make the configurations is very easy and user friendly, so even if someone is not able to write code, he/she can use all the features included in Cheddar without any problem.

2.2 RTSim

RTSim is a real-time system simulator developed at Retis Lab of the Scuola Superiore Sant'Anna. RTSim has four components:

- metasim: a generic C++ library used for the simulation of discrete event systems.
- rtlib: based on metasim, is the core library of RTSim. It is used for the simulation of scheduling algorithms and real-time tasks.
- ctrlib: a library for the simulation of real-time control systems.
- jtracer: a java tool for the visualisation of schedule traces produced by a rtlib program.

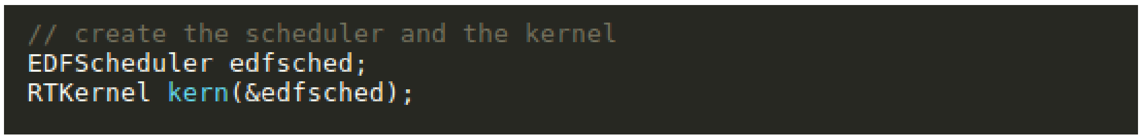
To make the simulation, the user has to write a C++ source file, using the *main()* function and including all the needed libraries (all the examples shown from this point onward refers to the implementation of an EDF scheduler).



```
#include <rtlib.hpp>

using namespace MetaSim;
using namespace Simulation;
using namespace RTSim;

int main()
{
```

Figure 5: RTSim needed headers and namespaces

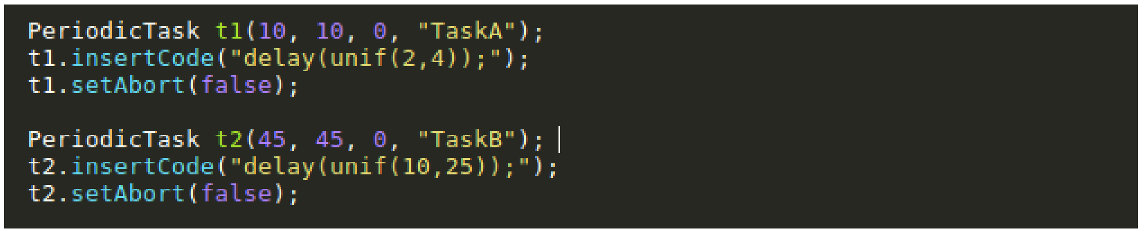
Then, the chosen scheduler must be created and has to be associated with the kernel. Note that there is a class for each implemented scheduler: EDFScheduler for Earliest Deadline First, FPScheduler for Fixed Priority, etc.



```
// create the scheduler and the kernel
EDFScheduler edfsched;
RTKernel kern(&edfsched);
```

Figure 6: RTSim scheduler definition

After that, the tasks that have to run in the simulation must be defined and the code they run is associated to them.

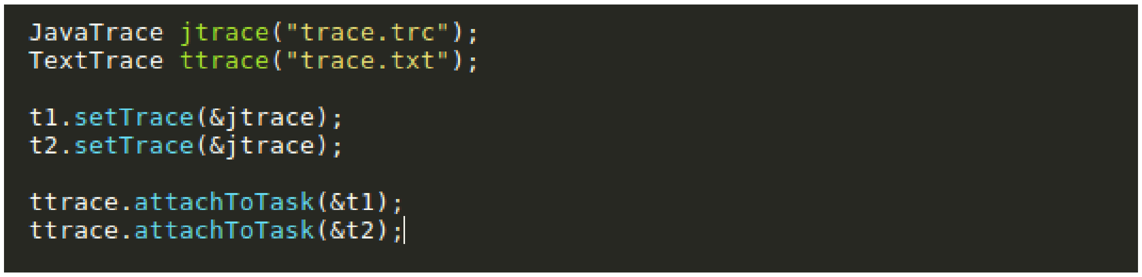


```
PeriodicTask t1(10, 10, 0, "TaskA");
t1.insertCode("delay(unif(2,4));");
t1.setAbort(false);

PeriodicTask t2(45, 45, 0, "TaskB");
t2.insertCode("delay(unif(10,25));");
t2.setAbort(false);
```

Figure 7: RTSim task definition

In the case illustrated above, two tasks are created; they are both periodic task. The first argument of the periodic task constructor represents the period, the second argument is the realtive deadline, the third argument is the starting time and the last argument is the name of the task. The *t1.insertCode("delay(unif(2,4));")* piece of code specifies that the task has a variable execution time that varies between 2 and 4 milliseconds. Apart from periodic tasks, real-time tasks can be created too using class *RTTask* instead of *PeriodicTask*.

Next, the traces that will be used later for the visualisation of the scheduling scheme must be created and associated to the tasks

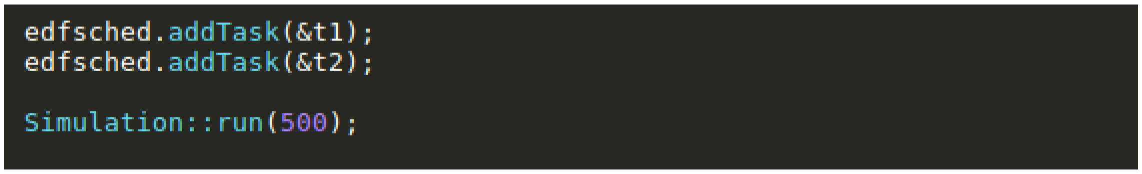


```
JavaTrace jtrace("trace.trc");
TextTrace ttrace("trace.txt");

t1.setTrace(&jtrace);
t2.setTrace(&jtrace);

ttrace.attachToTask(&t1);
ttrace.attachToTask(&t2);
```

Figure 8: RTSim tracing definition

There are two ways to accomplish the tracing for the visualisation: using a *JavaTrace* object and then associating the tasks to this object using the *setTrace* method, or using a *TextTrace* object and associating the tasks using the *attachToTask* method. This two procedures are equivalent, so only one of them can be used and the tracing visualisation will be performed correctly.

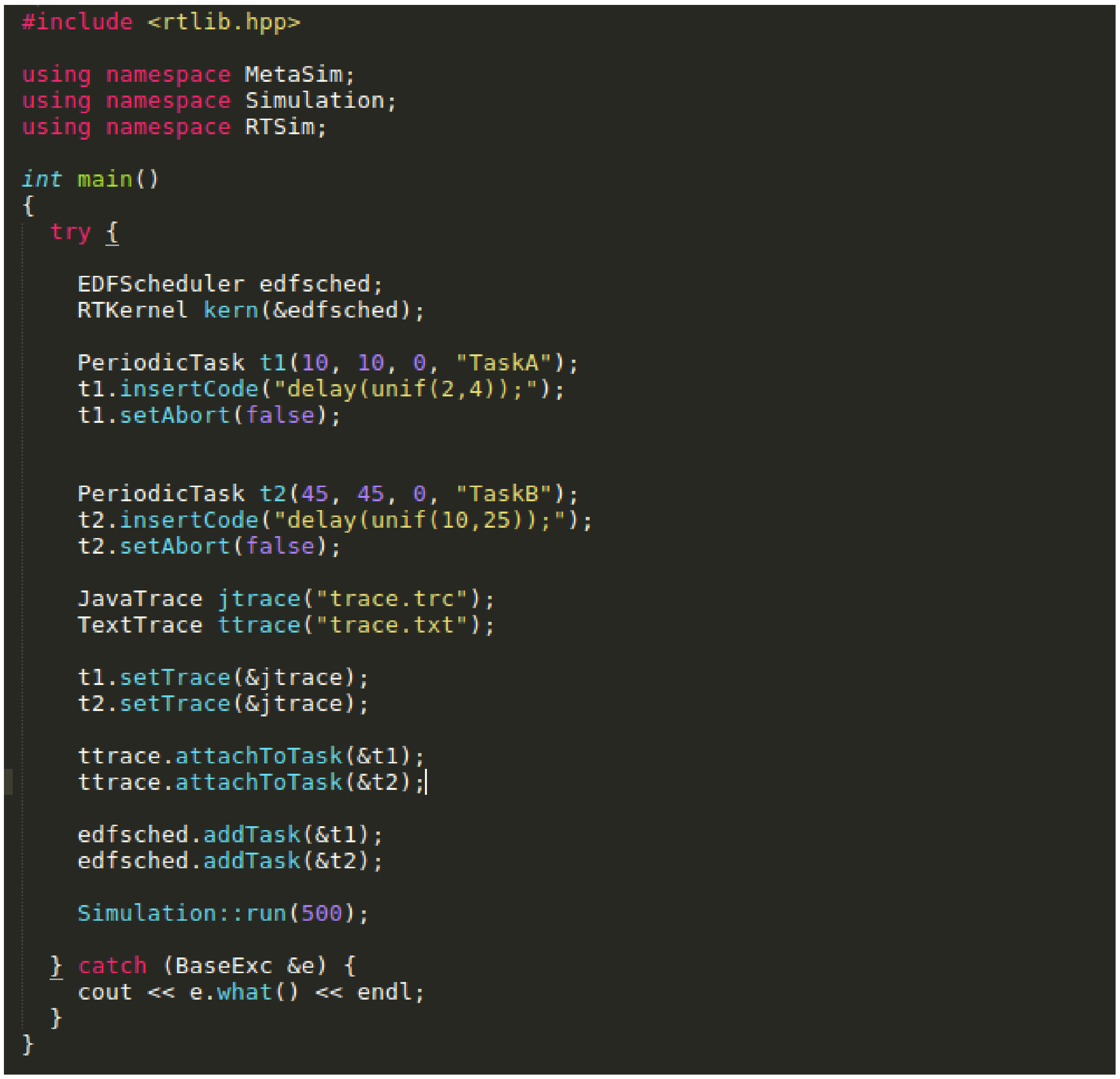
Finally, the code to specify that the tasks will be scheduled using the instantiated scheduler is added, and then the simulation is started using the command *Simulation::run(500)*, where 500 is the number of units of time for which the simulation will run.



```
edfsched.addTask(&t1);
edfsched.addTask(&t2);

Simulation::run(500);
```

Figure 9: Run the simulatio in RTSim

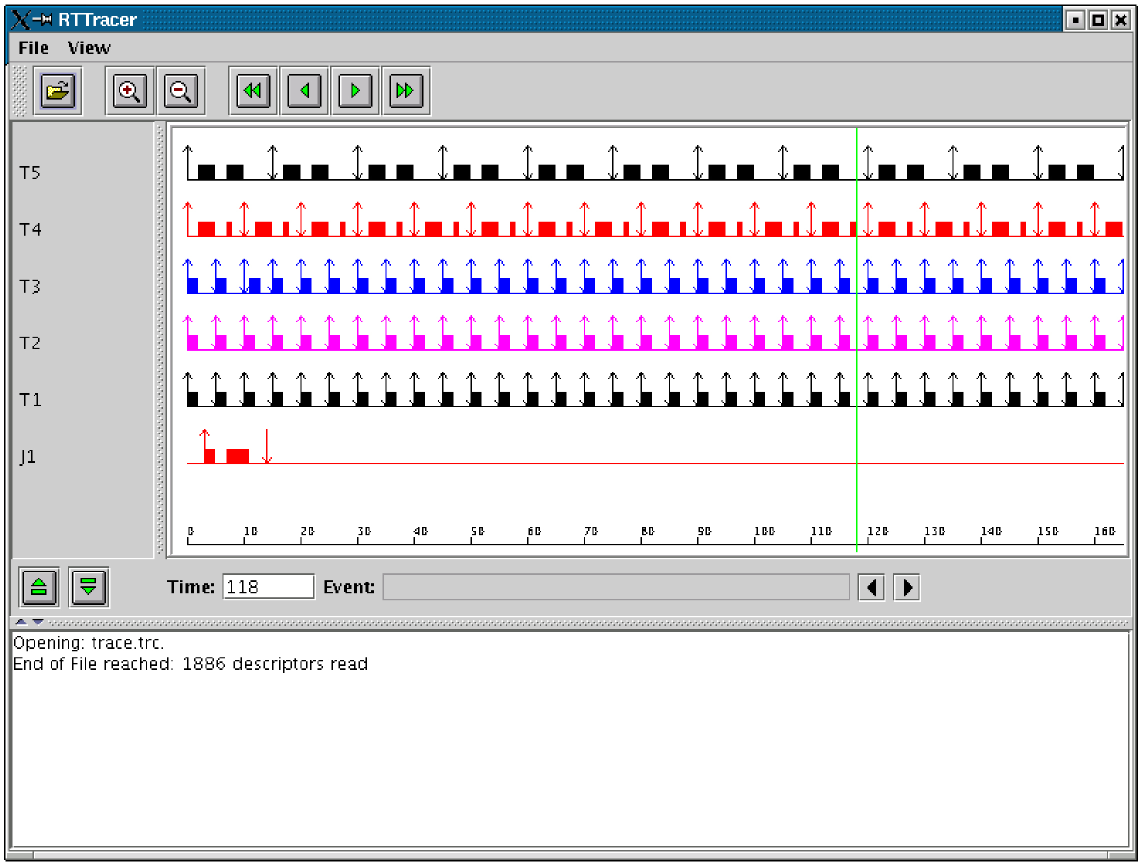
With all the code written, the main program will look something like this:



```
#include <rtlib.hpp>

using namespace MetaSim;
using namespace Simulation;
using namespace RTSim;

int main()
{
  try {

    EDFScheduler edfsched;
    RTKernel kern(&edfsched);

    PeriodicTask t1(10, 10, 0, "TaskA");
    t1.insertCode("delay(unif(2,4));");
    t1.setAbort(false);


    PeriodicTask t2(45, 45, 0, "TaskB");
    t2.insertCode("delay(unif(10,25));");
    t2.setAbort(false);

    JavaTrace jtrace("trace.trc");
    TextTrace ttrace("trace.txt");

    t1.setTrace(&jtrace);
    t2.setTrace(&jtrace);

    ttrace.attachToTask(&t1);
    ttrace.attachToTask(&t2);

    edfsched.addTask(&t1);
    edfsched.addTask(&t2);

    Simulation::run(500);

  } catch (BaseExc &e) {
    cout << e.what() << endl;
  }
}
```

Figure 10: RTSim complete simulation code

When the code is complete, the simulation che be run and the visualisation of the tracing can be seen. For example, this is the trace of a simulation using five tasks and the Earliest Deadline First scheduler.



```
Opening: trace.trc.
End of File reached: 1886 descriptors read
```

Figure 11: RTSim simulation results

In the scheduling trace, the vertical arrows represents the wake up time of each task.

RTSim is, in a way, both easier and harder to use with respect to Cheddar: harder because it requires that the user know how to write programs in C++ (even though the coding is very simple), easier because, in order to run a complete simulation, Cheddar requires the user to configure a lot of options and some of them might not be of immediate understanding for someone who doesn't use very often real-time systems and schedulers. While using RTSim can be useful to perform a simple and immediate scheduler simulation, it doesn't go very deep in the topic of real-time scheduling: Cheddar is a bit more complete since it allow the user to customize a lot of features of the simulation and gives away a lot more information, such as the scheduling feasibility, the number of context switches and the number of preemptions.

2.3 Litmus$^{\text{RT}}$

Litmus$^{\text{RT}}$ (Litmus for short) is a Linux kernel modified to support multiprocessor real-time scheduling; so unlike the previous cases with Cheddar and RTSim, Litmus is not a simulator but a real kernel.

Litmus adds to the Linux kernel a lot of scheduling policies which are implemented as scheduler plugins. The following scheduler plugins are currently working in Litmus:

- Linux: this is a dummy plugin, it just disables the real-time functionality handing over the control of the kernel to the default Linux kernel.

- Partitioned Fixed Priority scheduler (P-FP): with this plugin each task has a fixed priority and it's assigned to a certain processor. The scheduling decisions are made considering the priority: every time the scheduler must decide which task to run, it makes a certain processor run the highest priority task assigned to it.

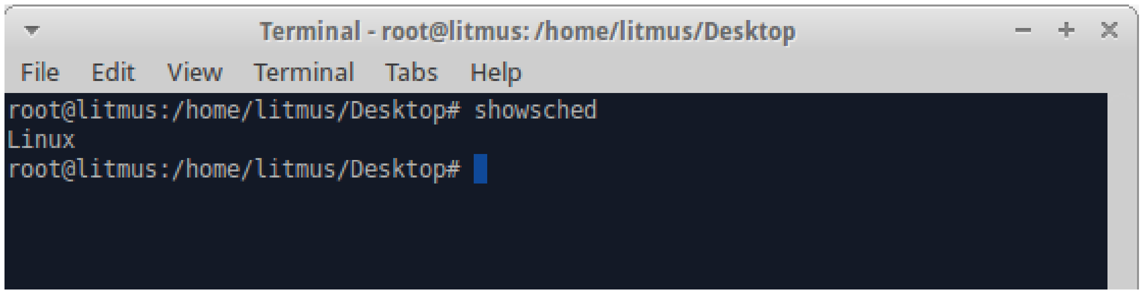
- Partitioned Dynamic Priority Earliest Deadline First (PSN-EDF): this scheduler works like just an EDF scheduler, the only peculiar feature is that (being it partitioned), the scheduler makes each processor run the task with the closest deadline among the tasks that are assigned to it.

- Global Earliest Deadline First (GSN-EDF): this scheduler is like the previous one (PSN-EDF), with the only difference that it is not partitioned, but it's global. This means that the tasks are not assigned to a particular processor and so the scheduler makes each processor run the task with the closest deadline among all tasks.

- Clustered Earliest Deadline First (C-EDF): this scheduler is a hybrid of PSN-EDF and GSN-EDF. Some non-overlapping sets of processors are scheduled independently with a global policy. These clusters are built using cache topology; the cluster value can be "L1", "L2", "L3" or "ALL", where "ALL" is equivalent to global scheduling (so the scheduling policy will be the same as GSN-EDF) and "L1" is equivalent to partitioned scheduling (the scheduling policy will be the same as PSN-EDF).

- Proportionate Fair (PFAIR): this scheduling policy is based on the PD^2 algorithm.
- Reservation Based (P-RES): this scheduling policy is based on reservations, that are "object" to which tasks are assigned to. Litmus supports three types of reservations (the way reservations work will be explained later on in this chaper):
  - periodic polling server
  - sporadic polling server
  - table-driven reservations

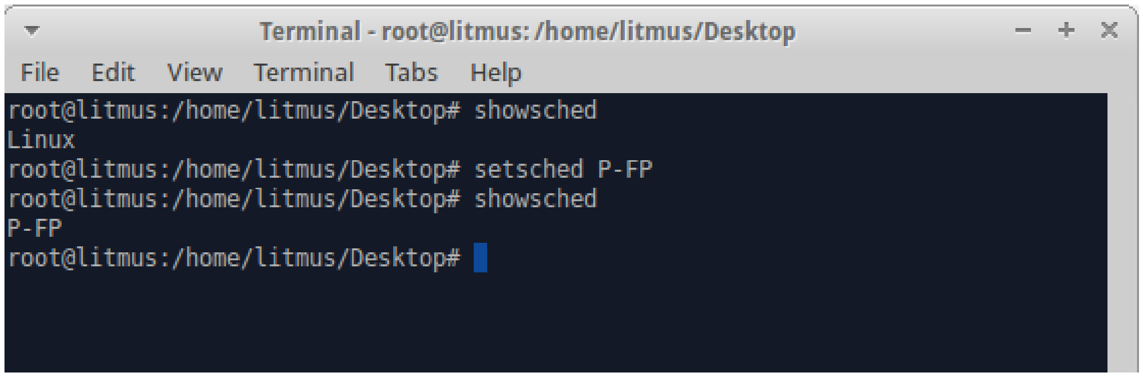
While using Litmus, shell commands can be used to perform almost all the operations regarding the schedulers management. For example, the command *showsched* can be used in order to check the currently active scheduler plugin.


Terminal - root@litmus: /home/litmus/Desktop
File Edit View Terminal Tabs Help

```
root@litmus:/home/litmus/Desktop# showsched
Linux
root@litmus:/home/litmus/Desktop# 
```

Figure 12: Litmus showsched command

The command setsched can be used to make Litmus use one of the defined plugins, using the name in paratheses as shown in the list above. The following picture shows how to make the kernel use Partioned Fixed-Priority (P-FP) scheduler.


Terminal - root@litmus: /home/litmus/Desktop
File Edit View Terminal Tabs Help

```
root@litmus:/home/litmus/Desktop# showsched
Linux
root@litmus:/home/litmus/Desktop# setsched P-FP
root@litmus:/home/litmus/Desktop# showsched
P-FP
root@litmus:/home/litmus/Desktop# 
```

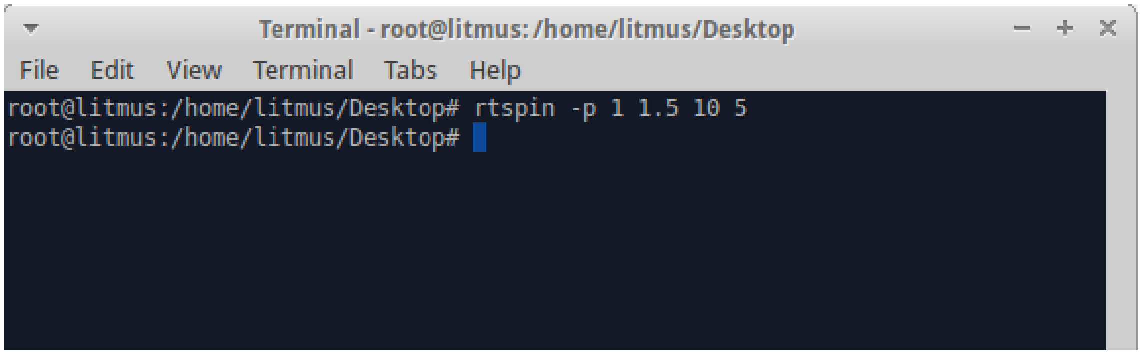
Figure 13: Litmus setsched command

2.3.1 Real-time tasks in Litmus

In Litmus, real-time tasks are characterized by these parameters, that must be set before the task is run:

- Worst-case execution time (WCET): this is the budget of a tasks, that is the maximum execution time the task can have every time is executed.
- Period: the time interval between two successive activations of a task.
- Deadline: as explained in section 1.2, this is the time limit within which a task must complete after it is released.
- Partition: in the case of partitioned schedulers (such as P-FP and PSN-EDF), this parameter specify the partition the task belongs to.

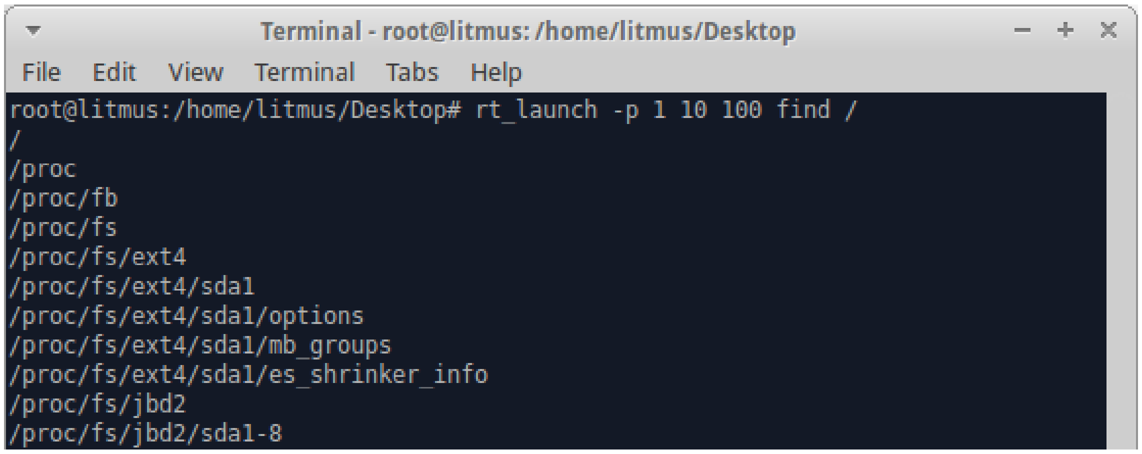
Real-time tasks can be executed in Litmus by using one of the two commands available: *rtspin* and *rt_launch*. *rtspin* creates a "dummy" real-time task that loops for the duration specified by the user and is useful for simulating CPU workloads and testing. For example, the following command must be used in order to run a real-time task for 5 seconds, on core 1 with a period of 10 ms and a WCET of 1.5 ms:



```
root@litmus:/home/litmus/Desktop# rtspin -p 1 1.5 10 5
root@litmus:/home/litmus/Desktop# 
```

Figure 14: Litmus rtspin command

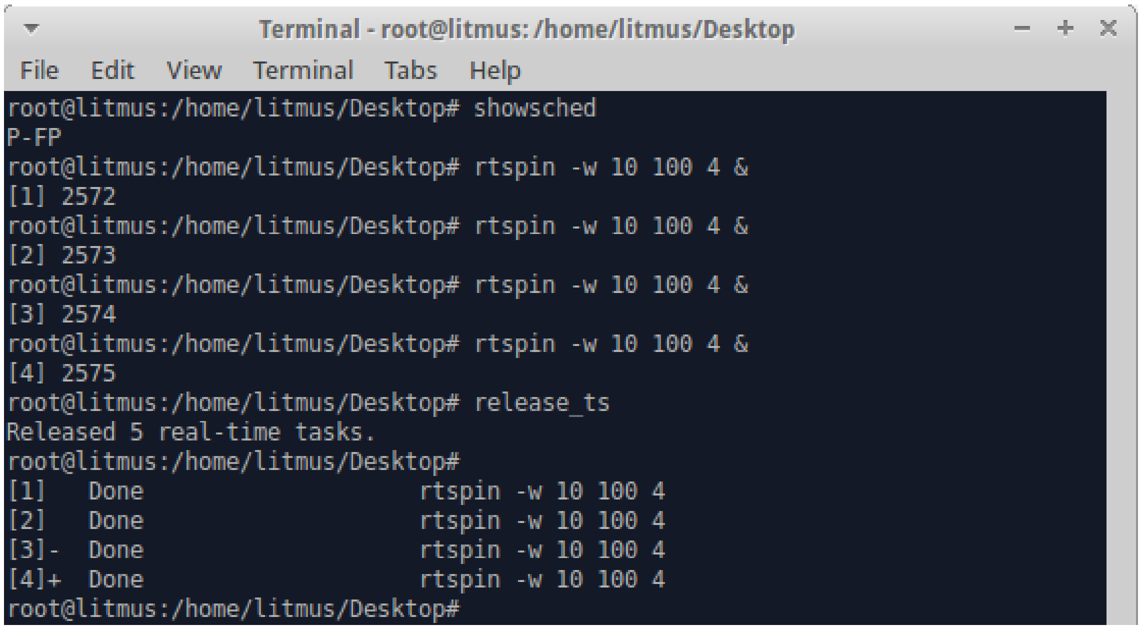
The other command used to run real-time tasks in Litmus is *rt_launch*. This command is useful to run any task in real-time mode, even code that has not been designed to run in a real-time environment. For example, the following command must be used in order to run the find program as a real-time task on processor 1, with a period of 100 ms and a budget of 10 ms:



```
root@litmus:/home/litmus/Desktop# rt_launch -p 1 10 100 find /
/
/proc
/proc/fb
/proc/fs
/proc/fs/ext4
/proc/fs/ext4/sda1
/proc/fs/ext4/sda1/options
/proc/fs/ext4/sda1/mb_groups
/proc/fs/ext4/sda1/es_shrinker_info
/proc/fs/jbd2
/proc/fs/jbd2/sda1-8
```

Figure 15: Litmus rt_launch command

Litmus gives the user the possibility to make many tasks commence execution only after all the tasks have been initialized; in this way all tasks share the same starting time and this is important for the simulation of the tasks execution. In order to do this, when defining the execution of a real-time task with *rtspin* or *rt_launch*, the user must use the *-w* option (that means that the task must wait for a synchronous time release) add a *$* character at the end of the command, in this way the task waits in background for the release. After all the tasks have been deined and initialized, the user can use the *release_ts* command in order to start all the previously defined tasks.


Terminal - root@litmus: /home/litmus/Desktop
File Edit View Terminal Tabs Help

```
root@litmus:/home/litmus/Desktop# showsched
P-FP
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[1] 2572
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[2] 2573
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[3] 2574
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[4] 2575
root@litmus:/home/litmus/Desktop# release_ts
Released 5 real-time tasks.
root@litmus:/home/litmus/Desktop#
[1]   Done                    rtspin -w 10 100 4
[2]   Done                    rtspin -w 10 100 4
[3]-  Done                    rtspin -w 10 100 4
[4]+  Done                    rtspin -w 10 100 4
root@litmus:/home/litmus/Desktop#
```

Figure 16: Litmus release_ts behavior

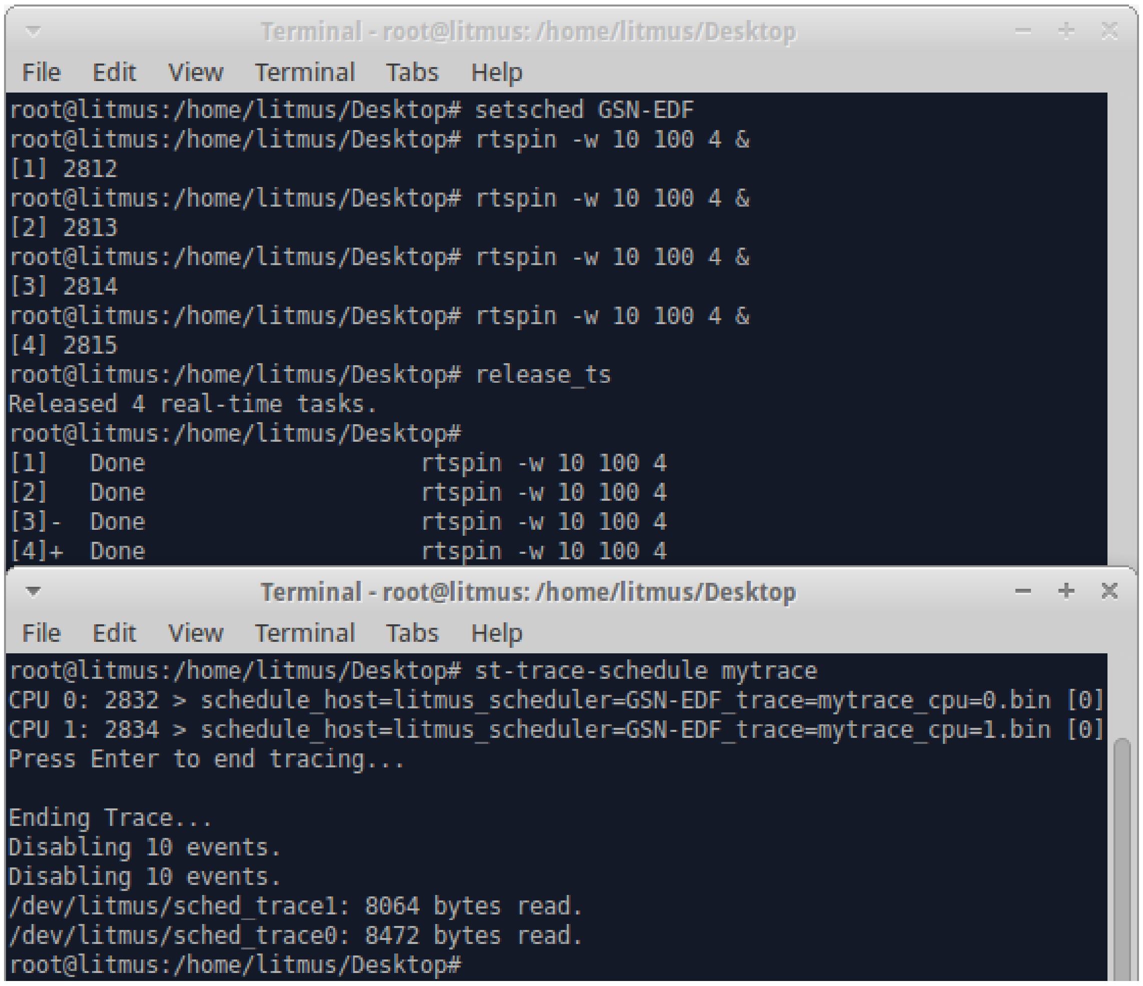
2.3.2 Tracing in Litmus

Litmus allows the user to trace both schedule and system overheads (like context switches costs and scheduling costs).

Schedule tracing In order to trace a schedule, the user has to use the *st-trace-schedule* command. The outline to make a complete schedule tracing is the following:

- issue the *st-trace-schedule* command to start recording the schedule.

- Initialize all the tasks that will run using *rtspin* or *rt_launch*, making them wait for a synchronous release using the *-w* option.
- Use the *release_ts* command in order to release all the previously defined tasks.
- Stop *st-trace-schedule* to stop the tracing of the schedule when the tasks have completed their execution.


Terminal - root@litmus: /home/litmus/Desktop
File Edit View Terminal Tabs Help

```
root@litmus:/home/litmus/Desktop# setsched GSN-EDF
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[1] 2812
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[2] 2813
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[3] 2814
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[4] 2815
root@litmus:/home/litmus/Desktop# release_ts
Released 4 real-time tasks.
root@litmus:/home/litmus/Desktop#
[1]   Done                    rtspin -w 10 100 4
[2]   Done                    rtspin -w 10 100 4
[3]-  Done                    rtspin -w 10 100 4
[4]+  Done                    rtspin -w 10 100 4
```

Terminal - root@litmus: /home/litmus/Desktop
File Edit View Terminal Tabs Help

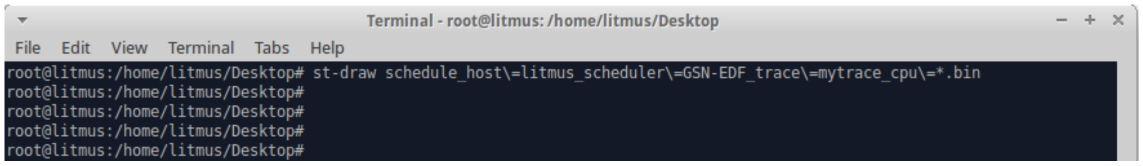
```
root@litmus:/home/litmus/Desktop# st-trace-schedule mytrace
CPU 0: 2832 > schedule_host=litmus_scheduler=GSN-EDF_trace=mytrace_cpu=0.bin [0]
CPU 1: 2834 > schedule_host=litmus_scheduler=GSN-EDF_trace=mytrace_cpu=1.bin [0]
Press Enter to end tracing...

Ending Trace...
Disabling 10 events.
Disabling 10 events.
/dev/litmus/sched_trace1: 8064 bytes read.
/dev/litmus/sched_trace0: 8472 bytes read.
root@litmus:/home/litmus/Desktop#
```

Figure 17: Litmus, how to use the st-trace-schedule command

The *st-trace-schedule* tool records event while the tasks run, in order to have a complete decription on which scheduling decisions are made. In particular, the events recorded by the tracing tool are:

- the start of execution of a task.
- The preemption of a task
- The suspension of a task
- The resuming of a task

When the tracing is run, a file containing the above events is created for each cpu used to run the tasks.
In order to have a clearer and more readable representation of the traced schedule, it's possible for the user to use a tool to draw the recorded schedule. This tool is called *st-draw* and is based on pycairo; the command syntax is the following:



```
root@litmus:/home/litmus/Desktop# st-draw schedule_host\=litmus_scheduler\=GSN-EDF_trace\=mytrace_cpu\=*.bin
root@litmus:/home/litmus/Desktop#
root@litmus:/home/litmus/Desktop#
root@litmus:/home/litmus/Desktop#
root@litmus:/home/litmus/Desktop#
```

Figure 18: Litmus st-draw command

Where *schedule_host*$\backslash$=*litmus_scheduler*$\backslash$=*GNS-EDF*$\backslash$=*mytrace_cpu*$\backslash$=**.bin* indicates that the tool must draw the schedule of all the files created by the tracing tool (to obtain the name of each file, just replace the * with the number associated to the CPU. For example in this case the kernel uses to CPUs, so there are two files, one with 0 and one with 1 where the + is located). The product of the *st-draw* tool is drawing of the schedule that looks like this:

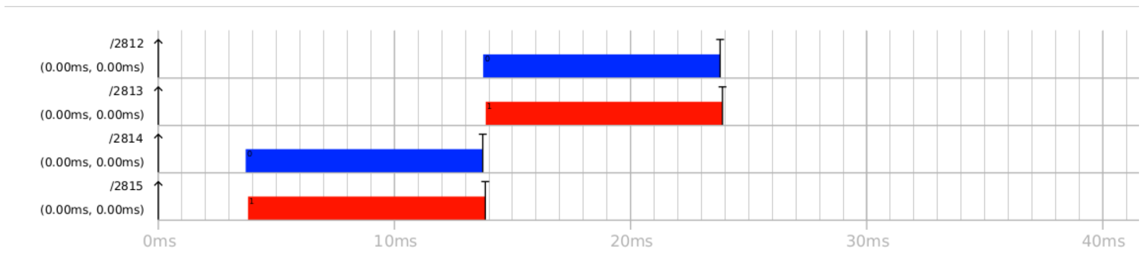


Figure 19: Results of the schedule tracing in Litmus

In this drawing we can see a graph for each task that has been executed, with the task id at the left of the graph (in this case the ids are 2812, 2813, 2814, 2815). The colored rectangles represents the time intervals in which the tasks are executed, and are in different color depending on which CPU has executed the task. For example, in this case tasks 2812 and 2814 have been executed by CPU 0, while tasks 2813 and 2815 have been executed by CPU 1. With this drawing the user can easily verify that the task settings specified during task definition with rtspin are respected and the scheduling decision are done accordingly. Zooming out a bit, we can also see that all the tasks have a period of 100 ms and that the end of the period is marked with a vertical arrow.

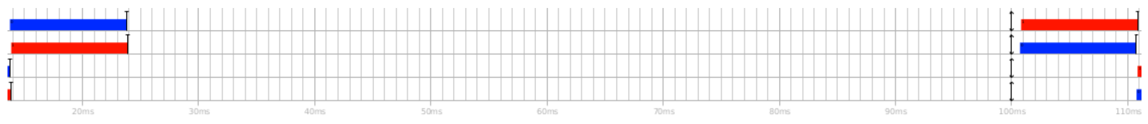
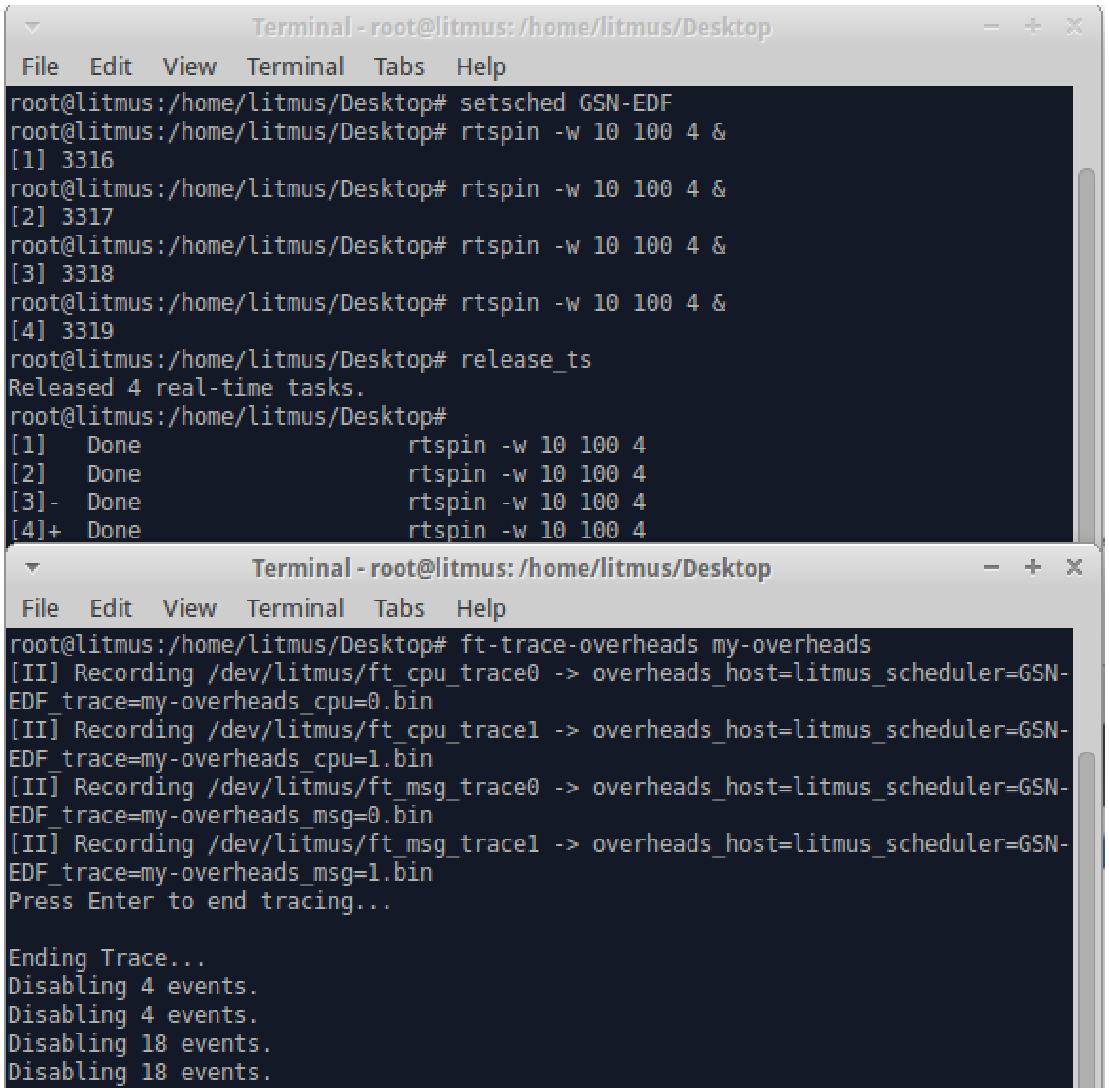


Figure 20: Showing the task period in the results of the schedule tracing

Overhead tracing In order to trace overhead during a schedule, Litmus allows the user to use the Feather-Trace tool that is used in the following way (the operations are very similar to the ones used for schedule tracing):

- issue the *ft-trace-overheads* command to start recording overheads.
- Initialize all the tasks that will run using *rtspin* or *rt_launch*, making them wait for a synchronous release using the *-w* option.
- Use the *release_ts* command in order to release all the previously defined tasks.
- Stop *ft-trace-overheads* to stop the tracing of the overheads when the tasks have completed their execution.


Terminal - root@litmus: /home/litmus/Desktop
File Edit View Terminal Tabs Help

```
root@litmus:/home/litmus/Desktop# setsched GSN-EDF
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[1] 3316
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[2] 3317
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[3] 3318
root@litmus:/home/litmus/Desktop# rtspin -w 10 100 4 &
[4] 3319
root@litmus:/home/litmus/Desktop# release_ts
Released 4 real-time tasks.
root@litmus:/home/litmus/Desktop#
[1]   Done                    rtspin -w 10 100 4
[2]   Done                    rtspin -w 10 100 4
[3]-  Done                    rtspin -w 10 100 4
[4]+  Done                    rtspin -w 10 100 4
```

Terminal - root@litmus: /home/litmus/Desktop
File Edit View Terminal Tabs Help

```
root@litmus:/home/litmus/Desktop# ft-trace-overheads my-overheads
[II] Recording /dev/litmus/ft_cpu_trace0 -> overheads_host=litmus_scheduler=GSN-
EDF_trace=my-overheads_cpu=0.bin
[II] Recording /dev/litmus/ft_cpu_trace1 -> overheads_host=litmus_scheduler=GSN-
EDF_trace=my-overheads_cpu=1.bin
[II] Recording /dev/litmus/ft_msg_trace0 -> overheads_host=litmus_scheduler=GSN-
EDF_trace=my-overheads_msg=0.bin
[II] Recording /dev/litmus/ft_msg_trace1 -> overheads_host=litmus_scheduler=GSN-
EDF_trace=my-overheads_msg=1.bin
Press Enter to end tracing...

Ending Trace...
Disabling 4 events.
Disabling 4 events.
Disabling 18 events.
Disabling 18 events.
```

Figure 21: Litmus, how to use the ft-trace-overheads command

With Feather-Trace, two files are created for each CPU: one for CPU local events like context switches and one for messages such as reschedule notifications. The files produces by Feather-Trace contains a list of event records, which consists of the following fields:

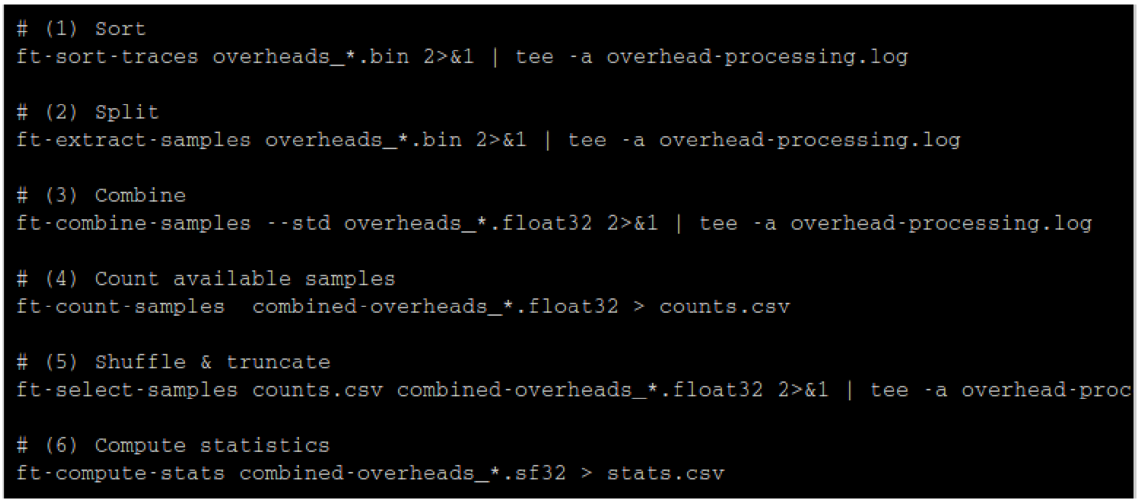
- an event ID

- the ID of the CPU on which the event occurred
- a sequence number
- the PID of the process that is related to that event
- the type of process related to that event
- a timestamp
- a flag that shows if any interrupts occurred since the previously recorded event
- a counter of the interrupts occurred since the previously recorded event

Once the overhead trace files are generated by the tool, the user can apply several other tools in order to analyse the data and extract meaningful information from them. Among the several tool available, the most significant are the high level tool, which are the following:

- Sorting: using the command *ft-sort-traces*, the user can sort the files in order to ensure that there are not out-of-order samples.
- Extracting samples: using the command *ft-extract-samples*, the user can extract the samples from the files and also discards any samples that were disturbed by interrupts.
- Combining samples: using the command *ft-combine-samples*, the user can combine many data files into one for further computation.
- Counting samples: using the command *ft-count-samples*, the script determines the minimum number of smaples recorded.
- Random sample selection: using the command *ft-select-samples*, the user can extract from the file generated by *ft-count-samples* a random set of samples that will be used for further computation. This procedure is used in order to obtain unbiased statistics.
- Computing simple statistics: using the command *ft-compute-stats*, the user can obtain the maximum, average, median, minimum, standard deviation and variance of the traced overheads.

All these tools are best used in chain, creating a toolchain that will provide the user meaningful data extracted from the overhead tracing.



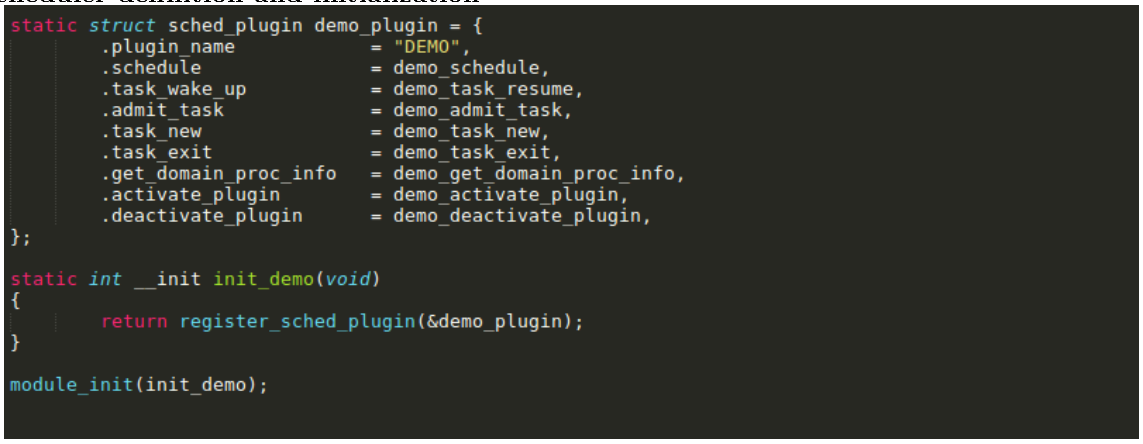
```
# (1) Sort
ft-sort-traces overheads_*.bin 2>&1 | tee -a overhead-processing.log

# (2) Split
ft-extract-samples overheads_*.bin 2>&1 | tee -a overhead-processing.log

# (3) Combine
ft-combine-samples --std overheads_*.float32 2>&1 | tee -a overhead-processing.log

# (4) Count available samples
ft-count-samples  combined-overheads_*.float32 > counts.csv

# (5) Shuffle & truncate
ft-select-samples counts.csv combined-overheads_*.float32 2>&1 | tee -a overhead-proc

# (6) Compute statistics
ft-compute-stats combined-overheads_*.sf32 > stats.csv
```

Figure 22: Overhead tracing toolchain in Litmus

2.3.3 Creating a new plugin in Litmus

Litmus allows the creation of new scheduler plugins that can later be added to the list of available scheduler, in order to let the user test the performance of the custom scheduler he/she defined. In order to do this, the user has to create a file in the folder */opt/litmus-rt/litmus* with a .c extention, for example *sched_demo.c*. Also, the user has to add *sched_demo.o* in the *obj-y* list in the *Makefile* situated in the *litmus* directory. After these preliminar actions, the user can then write the code of the scheduler. In this section every part of the code needed by the scheduler will be decribed and explained.

Scheduler definition and initialization

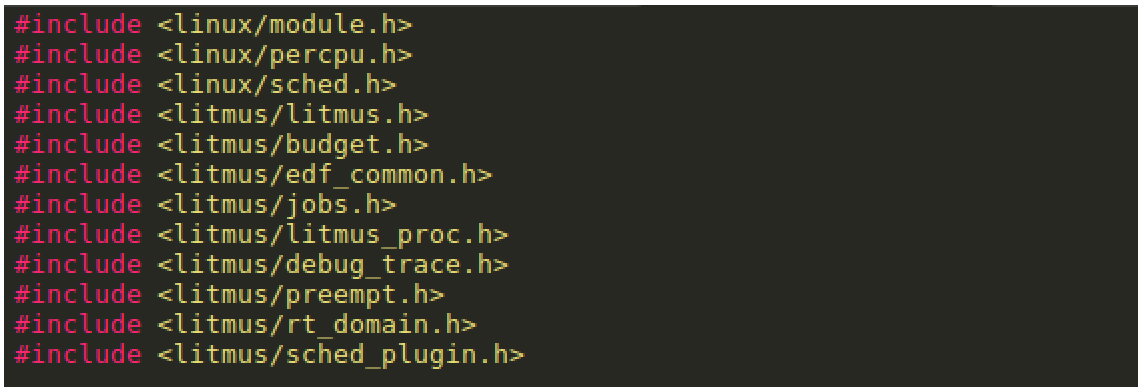


```
static struct sched_plugin demo_plugin = {
        .plugin_name            = "DEMO",
        .schedule               = demo_schedule,
        .task_wake_up           = demo_task_resume,
        .admit_task             = demo_admit_task,
        .task_new               = demo_task_new,
        .task_exit              = demo_task_exit,
        .get_domain_proc_info   = demo_get_domain_proc_info,
        .activate_plugin        = demo_activate_plugin,
        .deactivate_plugin      = demo_deactivate_plugin,
};

static int __init init_demo(void)
{
        return register_sched_plugin(&demo_plugin);
}

module_init(init_demo);
```

Figure 23: Litmus, scheduler definition and initialization

Since all the plugins in Litmus are kernel modules, the *module_ init(init_ demo)* macro is used in order to tell the compiler which function must be called during initialization. In this case the function called is *init_ demo*, which is defined above. The *init_ demo* function calls the *register_ sched_ plugin* function, that adds this plugin to the list of available schedulers in Litmus. Above the *init_ demo* function, there is a struct called *demo_ plugin* which contains all the information of the custom defined plugin and all the functions that must be called in case a particular event occurs. For example, while using the demo plugin, when a task wakes up the *demo_ task_ resume* function will be called, or when a scheduling decision must be made, the *demo_ schedule* function will be called.

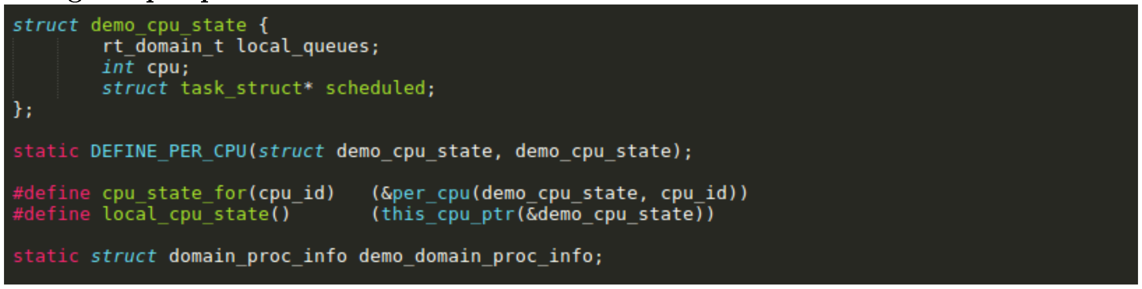
In order to make the following functions work, several headers must be included in the plugin file.



```
#include <linux/module.h>
#include <linux/percpu.h>
#include <linux/sched.h>
#include <litmus/litmus.h>
#include <litmus/budget.h>
#include <litmus/edf_common.h>
#include <litmus/jobs.h>
#include <litmus/litmus_proc.h>
#include <litmus/debug_trace.h>
#include <litmus/preempt.h>
#include <litmus/rt_domain.h>
#include <litmus/sched_plugin.h>
```

Figure 24: Litmus, headers needed for the definition of a new scheduler

Defining the per-processor state

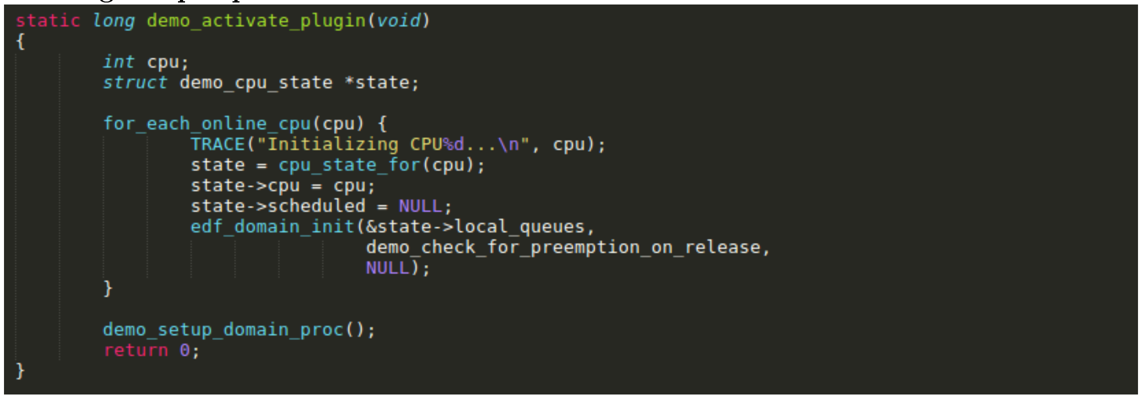


```
struct demo_cpu_state {
        rt_domain_t local_queues;
        int cpu;
        struct task_struct* scheduled;
};

static DEFINE_PER_CPU(struct demo_cpu_state, demo_cpu_state);

#define cpu_state_for(cpu_id)   (&per_cpu(demo_cpu_state, cpu_id))
#define local_cpu_state()       (this_cpu_ptr(&demo_cpu_state))

static struct domain_proc_info demo_domain_proc_info;
```

Figure 25: Litmus, defining the per-processor state

In order to make the plugin work, the user must define the variables that define the state of each processor. This is done using the struct *demo_ cpu_ state*, which contains: the variable *local_ queues* that represents the two queues needed by the scheduler (the ready queue and the release queue), the *cpu* variable that indicated the cpu id, the *scheduled* variable that shows the task that is currently scheduled. The *DEFINE_ PER_ CPU* is a macro that allocate the state of the plugin for each cpu, and the macros below wrap Linux's per-cpu data structure, they are not strictly needed but are used in order to make the code more readable.

Initializing the per-processor state



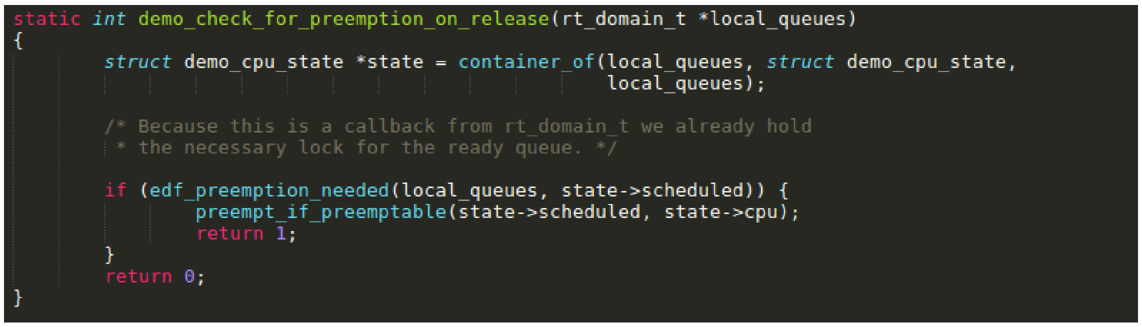
```
static long demo_activate_plugin(void)
{
        int cpu;
        struct demo_cpu_state *state;

        for_each_online_cpu(cpu) {
                TRACE("Initializing CPU%d...\n", cpu);
                state = cpu_state_for(cpu);
                state->cpu = cpu;
                state->scheduled = NULL;
                edf_domain_init(&state->local_queues,
                                demo_check_for_preemption_on_release,
                                NULL);
        }

        demo_setup_domain_proc();
        return 0;
}
```

Figure 26: Litmus, initializing the per-processor state

In order to initialize the state for each processor, the *demo_ activate_ plugin* function is used, which is the function that is called when the plugin is activated (that is when the user issues the *setsched* command). The only thing that this function does is to initialize the *demo_ cpu_ state* that was defined before for each cpu. The important instructions in this piece of code are:

- *state = cpu_ state_ for(cpu)* which use the before defined macro to initialize the ste for every cpu.
- *state->scheduled = NULL* that means that there isn't any currently scheduled task because the plugin is just activated.
- *edf_ domain_ init(&state->local_ queues, demo_ check_ for_ preemption_ on_ release, NULL)* which is the function defined for the EDF scheduler that initializes the two local queues.

The *demo_ check_ for_ preemption_ on_ release* is a function that is called when a task is released, and it check if the currently scheduled task must be preempted in order to make the just released task run. The function is defined in this way:

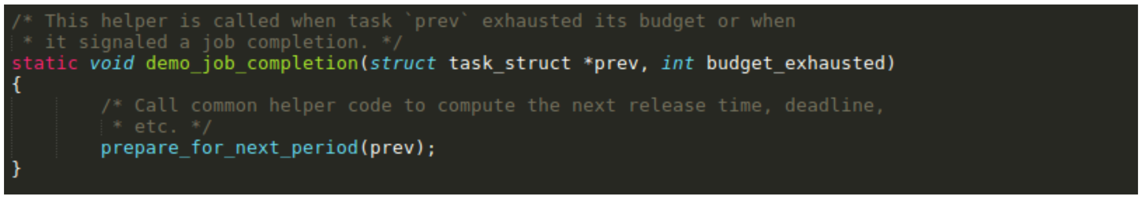


```c
static int demo_check_for_preemption_on_release(rt_domain_t *local_queues)
{
	struct demo_cpu_state *state = container_of(local_queues, struct demo_cpu_state,
	                                            local_queues);

	/* Because this is a callback from rt_domain_t we already hold
	 * the necessary lock for the ready queue. */

	if (edf_preemption_needed(local_queues, state->scheduled)) {
		preempt_if_preemptable(state->scheduled, state->cpu);
		return 1;
	}
	return 0;
}
```

Figure 27: Litmus, demo_check_for_preemption_on_release function definition

edf_preemption_needed and *preempt_if_preemptable* are EDF specific functions, the first one checks if the preemption is needed (that is, if there is a task in the ready queue with a closer deadline with respect to the running task), the second one performs the preemption.

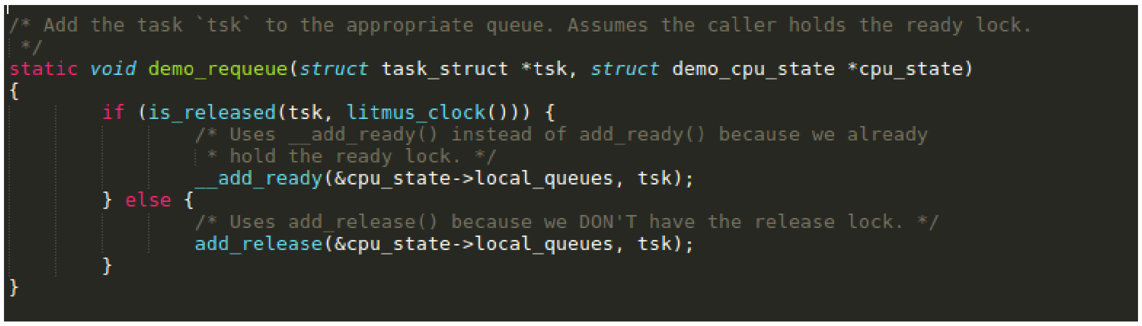
Scheduling logic In this section, the function that together form the scheduling logic are analyzed. First, the user needs to define the *demo_job_completion* function, which is called when a job completes. It just call the helper function *prepare_for_next_period* that is a general purpose function used in the majority of Limus schedulers, it does all the required procedures to ensure that the scheduler is ready for the next period.



```c
/* This helper is called when task `prev` exhausted its budget or when
 * it signaled a job completion. */
static void demo_job_completion(struct task_struct *prev, int budget_exhausted)
{
	/* Call common helper code to compute the next release time, deadline,
	 * etc. */
	prepare_for_next_period(prev);
}
```

Figure 28: Litmus, demo_job_completion function definition

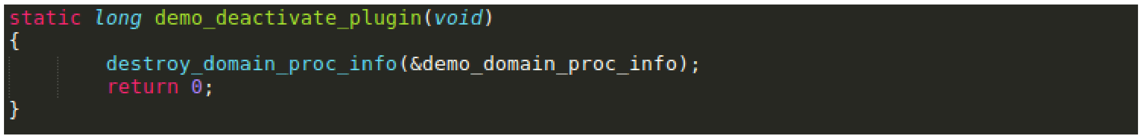
Next, the user must define the *demo_requeue* function, which is used to put a task in the correct queue. It simply put the task in the ready queue is the task is already released, otherwise it places the task in the release queue.



```
/* Add the task `tsk` to the appropriate queue. Assumes the caller holds the ready lock.
 */
static void demo_requeue(struct task_struct *tsk, struct demo_cpu_state *cpu_state)
{
        if (is_released(tsk, litmus_clock())) {
                /* Uses __add_ready() instead of add_ready() because we already
                 * hold the ready lock. */
                __add_ready(&cpu_state->local_queues, tsk);
        } else {
                /* Uses add_release() because we DON'T have the release lock. */
                add_release(&cpu_state->local_queues, tsk);
        }
}
```

Figure 29: Litmus, demo_requeue function definition

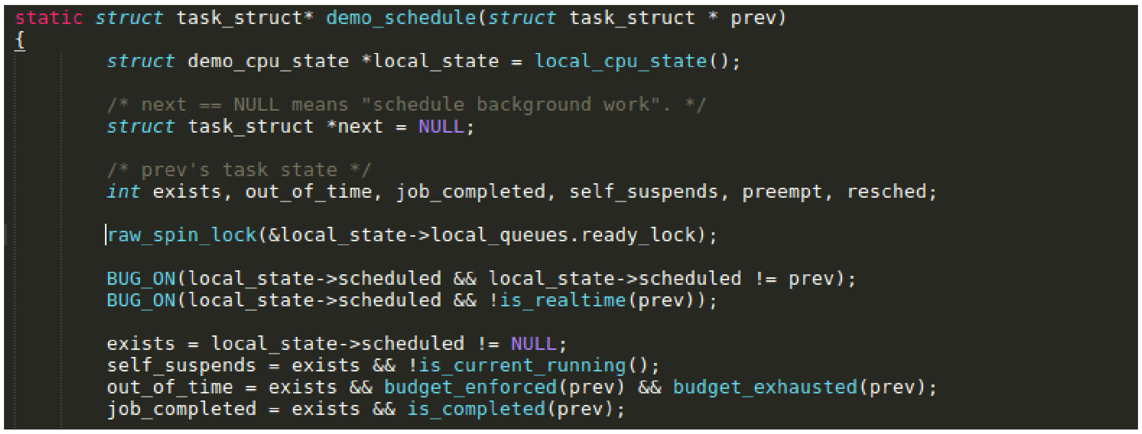
Also, a function used when the user deactives the plugin (that is, when another scheduler is chosen with the setsched command) must be defined. It is c*alled demo_deactivate_plugin* and it just erases all the information regarding the scheduler.



```
static long demo_deactivate_plugin(void)
{
        destroy_domain_proc_info(&demo_domain_proc_info);
        return 0;
}
```

Figure 30: Litmus, demo_deactivate_plugin function definition

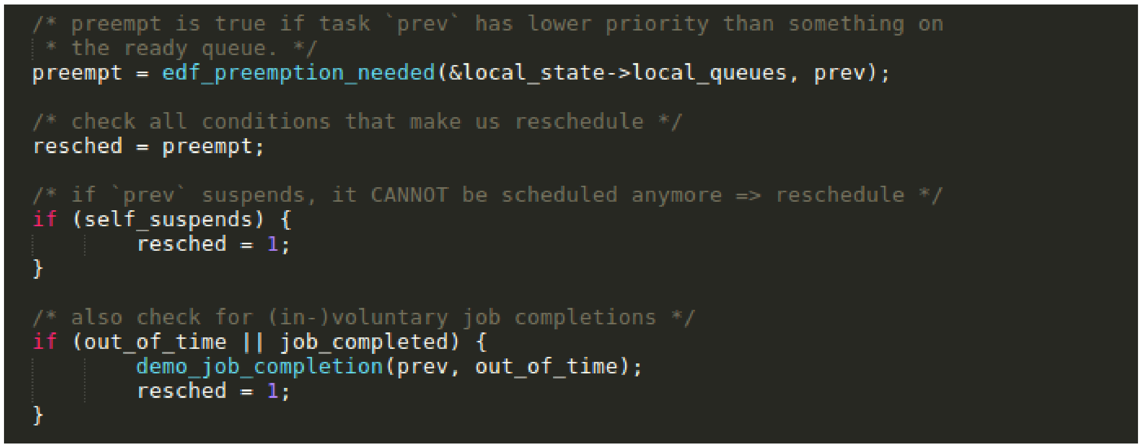
Regarding the effective scheduling logic, it is defined in the *demo_schedule* function. First, the *raw_spin_lock* instruction allows the scheduler to obtain th lock on the ready queue, then the function checks what states the previously scheduled task was (*exists*, *self_suspends*, *out_of_time* or *job_completed*).



```
static struct task_struct* demo_schedule(struct task_struct * prev)
{
        struct demo_cpu_state *local_state = local_cpu_state();

        /* next == NULL means "schedule background work". */
        struct task_struct *next = NULL;

        /* prev's task state */
        int exists, out_of_time, job_completed, self_suspends, preempt, resched;

        raw_spin_lock(&local_state->local_queues.ready_lock);

        BUG_ON(local_state->scheduled && local_state->scheduled != prev);
        BUG_ON(local_state->scheduled && !is_realtime(prev));

        exists = local_state->scheduled != NULL;
        self_suspends = exists && !is_current_running();
        out_of_time = exists && budget_enforced(prev) && budget_exhausted(prev);
        job_completed = exists && is_completed(prev);
```

Figure 31: Litmus, demo_schedule function definition

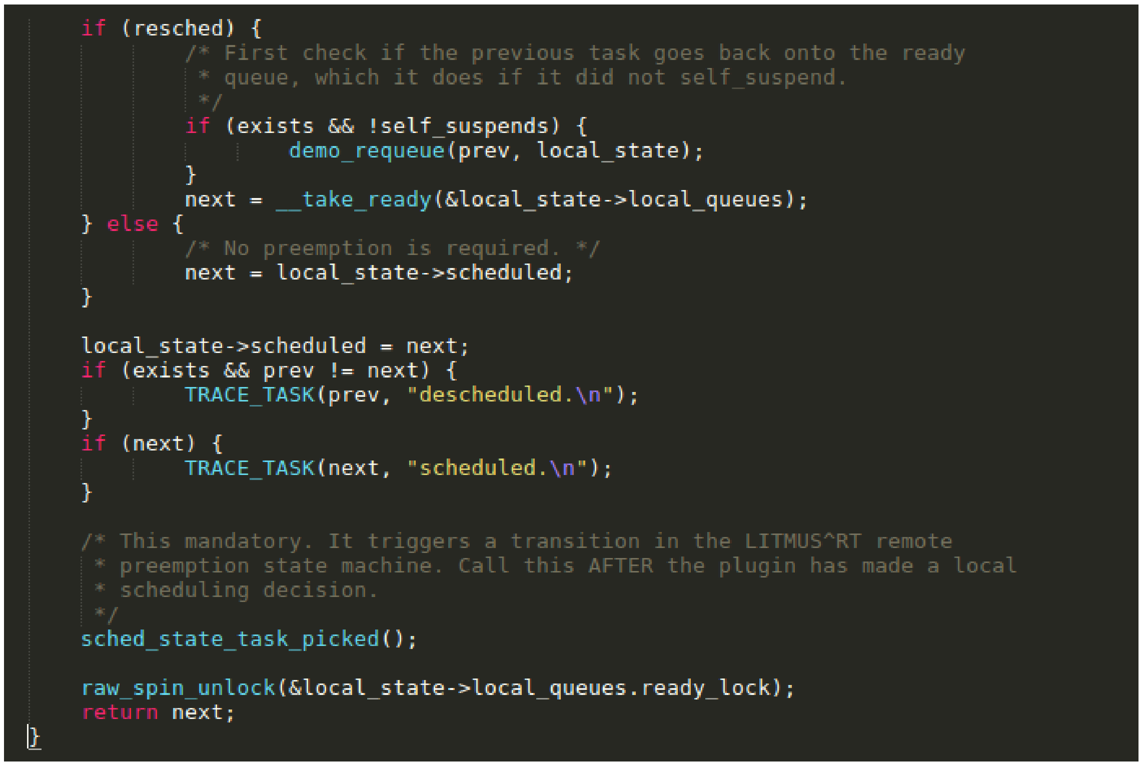
After that, the function checks if there is an higher priority task in the ready queue that has to be executed, if that is the case the currently running task is preempted and the higher priority task is executed. The running task is preempted if it has suspended execution, if it's out of time, if it has completed the job or (in the case of EDF) if there is a ready task with a closer deadline (this case is handled by the *edf_preemption_needed function*).



```
/* preempt is true if task `prev` has lower priority than something on
 * the ready queue. */
preempt = edf_preemption_needed(&local_state->local_queues, prev);

/* check all conditions that make us reschedule */
resched = preempt;

/* if `prev` suspends, it CANNOT be scheduled anymore => reschedule */
if (self_suspends) {
        resched = 1;
}

/* also check for (in-)voluntary job completions */
if (out_of_time || job_completed) {
        demo_job_completion(prev, out_of_time);
        resched = 1;
}
```

Figure 32: Litmus, preemption implementation

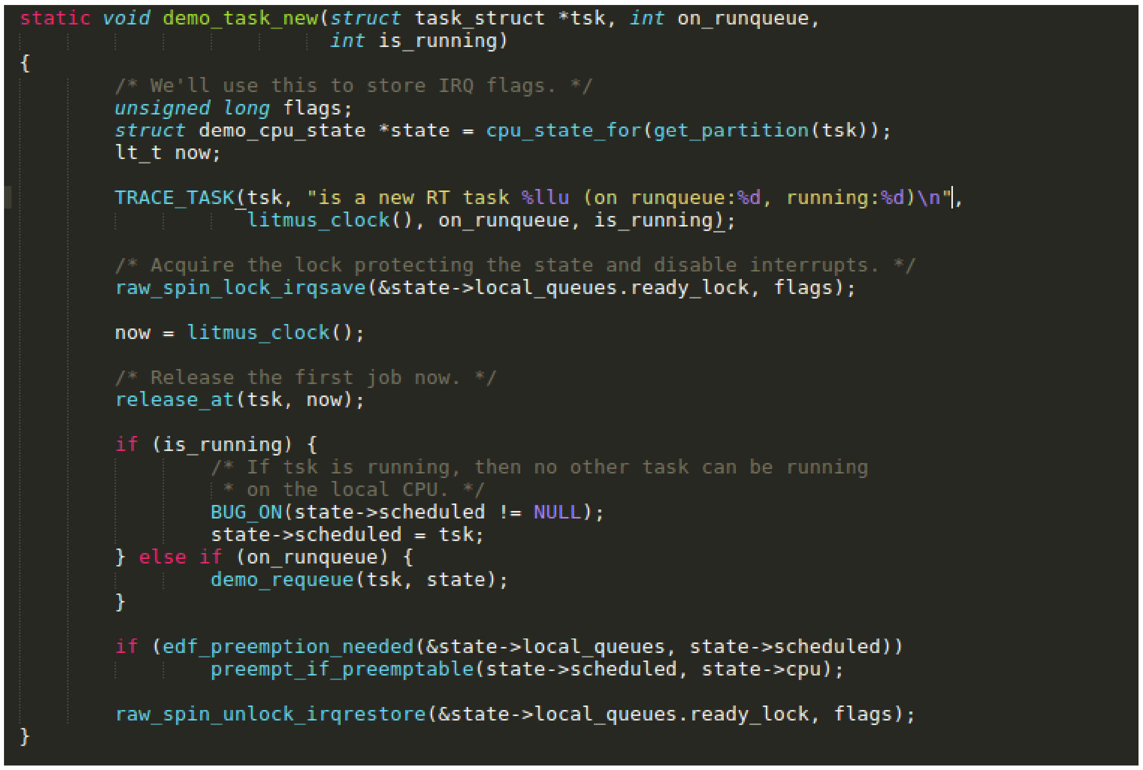
Then, if the currently running task must be preempted, it is placed in the ready or release queue depending on its state and the higher priority task in put in running mode. Finally, the lock on the ready queue is released.



```
	if (resched) {
		/* First check if the previous task goes back onto the ready
		 * queue, which it does if it did not self_suspend.
		 */
		if (exists && !self_suspends) {
			demo_requeue(prev, local_state);
		}
		next = __take_ready(&local_state->local_queues);
	} else {
		/* No preemption is required. */
		next = local_state->scheduled;
	}

	local_state->scheduled = next;
	if (exists && prev != next) {
		TRACE_TASK(prev, "descheduled.\n");
	}
	if (next) {
		TRACE_TASK(next, "scheduled.\n");
	}

	/* This mandatory. It triggers a transition in the LITMUS^RT remote
	 * preemption state machine. Call this AFTER the plugin has made a local
	 * scheduling decision.
	 */
	sched_state_task_picked();

	raw_spin_unlock(&local_state->local_queues.ready_lock);
	return next;
}
```

Figure 33: Litmus, moving the preempted task into the right queue

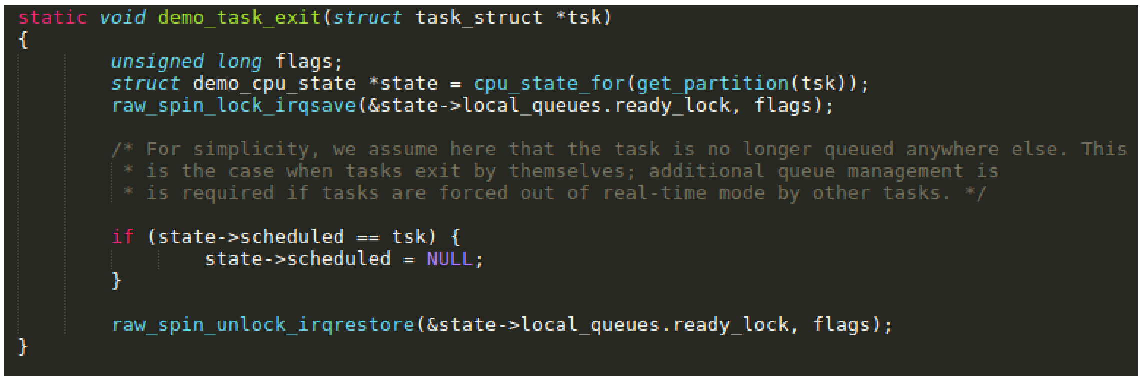
Task management This section explains the functions that must be defined in order to make the scheduler manage correctly the tasks that have to be executed. First, the function *demo_task_new* manages the new real-time tasks, that is the tasks that, while previously not real-time, becomes real-time. When this happens, this new real-time tasks is placed in running state if it's already running or it's put in the correct queue using the *demo_requeue* previously defined. After that, the function checks if there is need for preemption (that is, if the new real-time task is put in the ready queue and has a closer deadline than the running task, the running task must be preempted).



```c
static void demo_task_new(struct task_struct *tsk, int on_runqueue,
                          int is_running)
{
        /* We'll use this to store IRQ flags. */
        unsigned long flags;
        struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));
        lt_t now;

        TRACE_TASK(tsk, "is a new RT task %llu (on runqueue:%d, running:%d)\n",
                   litmus_clock(), on_runqueue, is_running);

        /* Acquire the lock protecting the state and disable interrupts. */
        raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);

        now = litmus_clock();

        /* Release the first job now. */
        release_at(tsk, now);

        if (is_running) {
                /* If tsk is running, then no other task can be running
                 * on the local CPU. */
                BUG_ON(state->scheduled != NULL);
                state->scheduled = tsk;
        } else if (on_runqueue) {
                demo_requeue(tsk, state);
        }

        if (edf_preemption_needed(&state->local_queues, state->scheduled))
                preempt_if_preemptable(state->scheduled, state->cpu);

        raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}
```

Figure 34: Litmus, demo_task_new function definition

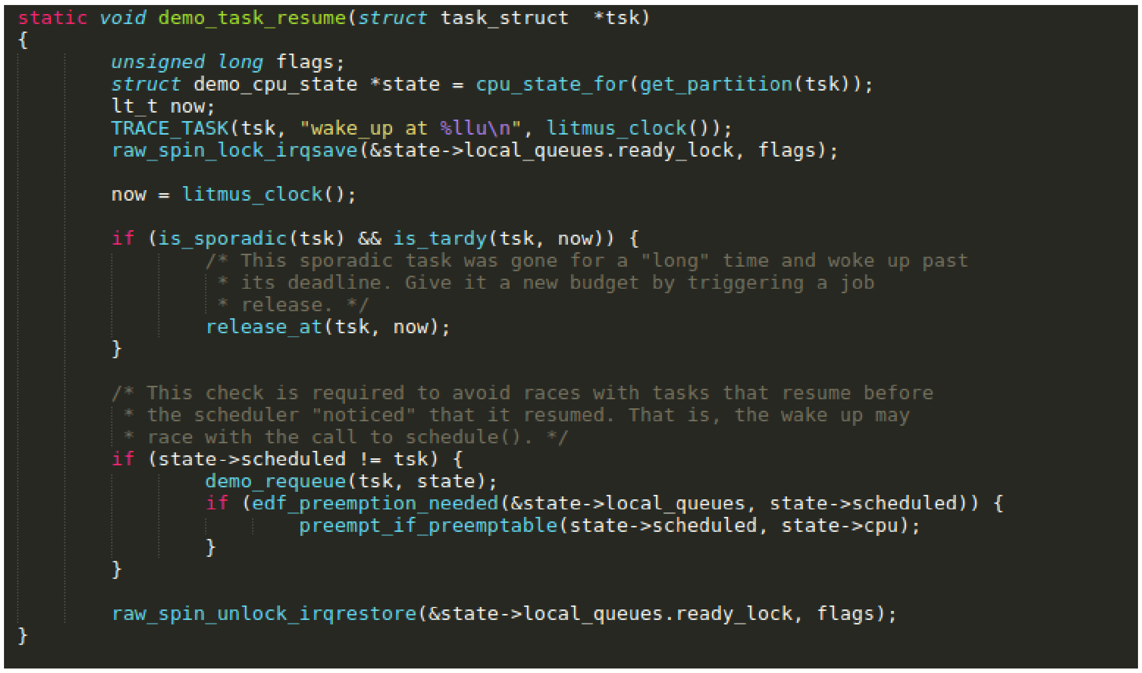
There is also the need for a function that maintains the scheduler state when a task exits. For this purpose, the *demo_task_exit* is defined and it simply put back to NULL the *scheduled* variable, indicating that there isn't a running task anymore.



```c
static void demo_task_exit(struct task_struct *tsk)
{
        unsigned long flags;
        struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));
        raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);

        /* For simplicity, we assume here that the task is no longer queued anywhere else. This
         * is the case when tasks exit by themselves; additional queue management is
         * is required if tasks are forced out of real-time mode by other tasks. */

        if (state->scheduled == tsk) {
                state->scheduled = NULL;
        }

        raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}
```

Figure 35: Litmus, demo_task_exit function definition

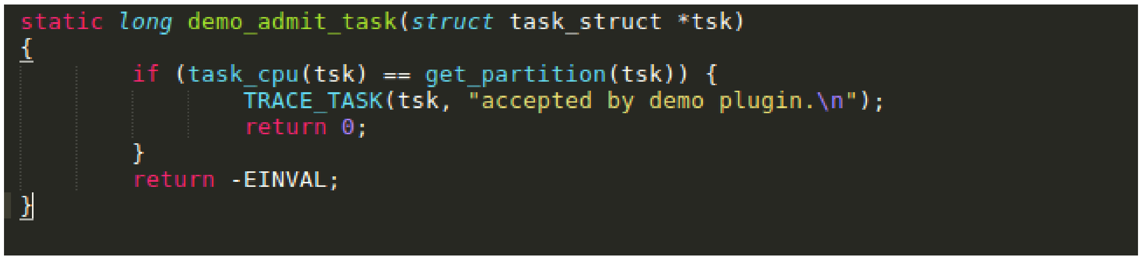
The *demo_task_resume* function manages the resuming task. If the resuming task is a sporadic one and resumes after its deadline, it is realeased assigning it a new budget (with the *release_at* function). In any case the task is placed in the correct queue and then the function checks if there is need for preemption as always.



```c
static void demo_task_resume(struct task_struct  *tsk)
{
	unsigned long flags;
	struct demo_cpu_state *state = cpu_state_for(get_partition(tsk));
	lt_t now;
	TRACE_TASK(tsk, "wake_up at %llu\n", litmus_clock());
	raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);

	now = litmus_clock();

	if (is_sporadic(tsk) && is_tardy(tsk, now)) {
		/* This sporadic task was gone for a "long" time and woke up past
		 * its deadline. Give it a new budget by triggering a job
		 * release. */
		release_at(tsk, now);
	}

	/* This check is required to avoid races with tasks that resume before
	 * the scheduler "noticed" that it resumed. That is, the wake up may
	 * race with the call to schedule(). */
	if (state->scheduled != tsk) {
		demo_requeue(tsk, state);
		if (edf_preemption_needed(&state->local_queues, state->scheduled)) {
			preempt_if_preemptable(state->scheduled, state->cpu);
		}
	}

	raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
}
```

Figure 36: Litmus, demo_task_resume function definition

Finally, the demo_admit_task function is needed to accept real-time tasks, provided that they are located on the right core.



```c
static long demo_admit_task(struct task_struct *tsk)
{
	if (task_cpu(tsk) == get_partition(tsk)) {
		TRACE_TASK(tsk, "accepted by demo plugin.\n");
		return 0;
	}
	return -EINVAL;
}
```

Figure 37: Litmus, demo_admit_task function definition

2.3.4 Litmus advantages and disadvantages

Litmus has some relevant advantages compared to the previously analyzed tools, Cheddar and RTSim; the most important one is that Litmus is not a simulated environment, but it is a real Linux kernel. This means that any test excuted using Litmus has high reliability and many elements that could be difficult to replicate in a simulated environment (such as disturbances) are naturally present in a real and not simulated system.

The main downside of Litmus is its procedure for new plugin creation. In fact, Litmus lets the user define a new scheduler as we just saw, but it does not allow for a complete customization of the new scheduler. For example, the Litmus plugin is not capable of changing the budget of real-time tasks at runtime, since the budget must be defined when the tasks are created using *rtspin* or *rt_launch*. This is a major flaw that, unfortunately, does not give the user complete freedom in the scheduler definition.

2.4 TrueTime

TrueTime is a real-time control system simulator based on Matlab and Simulink. It includes modules for the simulation of many different types of real-time systems: it has a module for network simulation (that can simulate CSMA/CD, token bus, switched ethernet and many other protocols), a module for wireless network simulation (that can simulate WLAN and ZigBee among the other protocols) and a module for the kernel simulation, which is the one that will be analyzed here. In particular, in order to fully understand how the TrueTime kernel works and how it manages the scheduler, the *threeservos* example will be used for reference. Opening the Simulink file of the *threeservos* example, the following window is displayed:

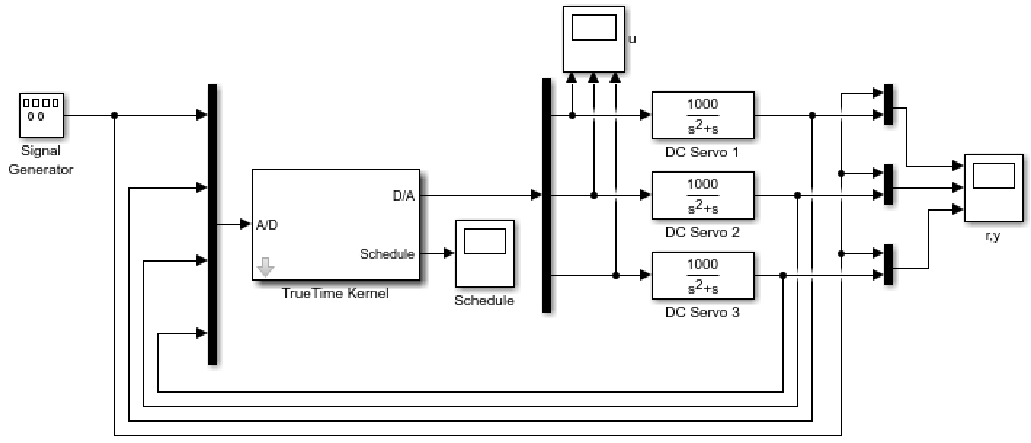


Figure 38: TrueTime real-time control system

Here we can recognize the structure of a real-time control system, where the tasks are simulated using the transfer functions on the right side of the picture. The most interesting Simulink block is the one called *TrueTime kernel*, that contains all the kernel logic used for the simulation. Once the simulation is run, the code associated with the *TrueTime kernel* block (which will be explained in the next section) is executed and the result of the scheduling decisions will be displayed in the *Schedule* monitor. In this example, there are many schedulers that the user can choose, like Deadline Monotonic and Earliest Deadline First schedulers. For instance, if the user chooses to run the simulation and then opens the *Schedule* monitor to see the result of the scheduling decision, this is what he/she can see (also depending on the tasks parameters which will be explained in the next section)

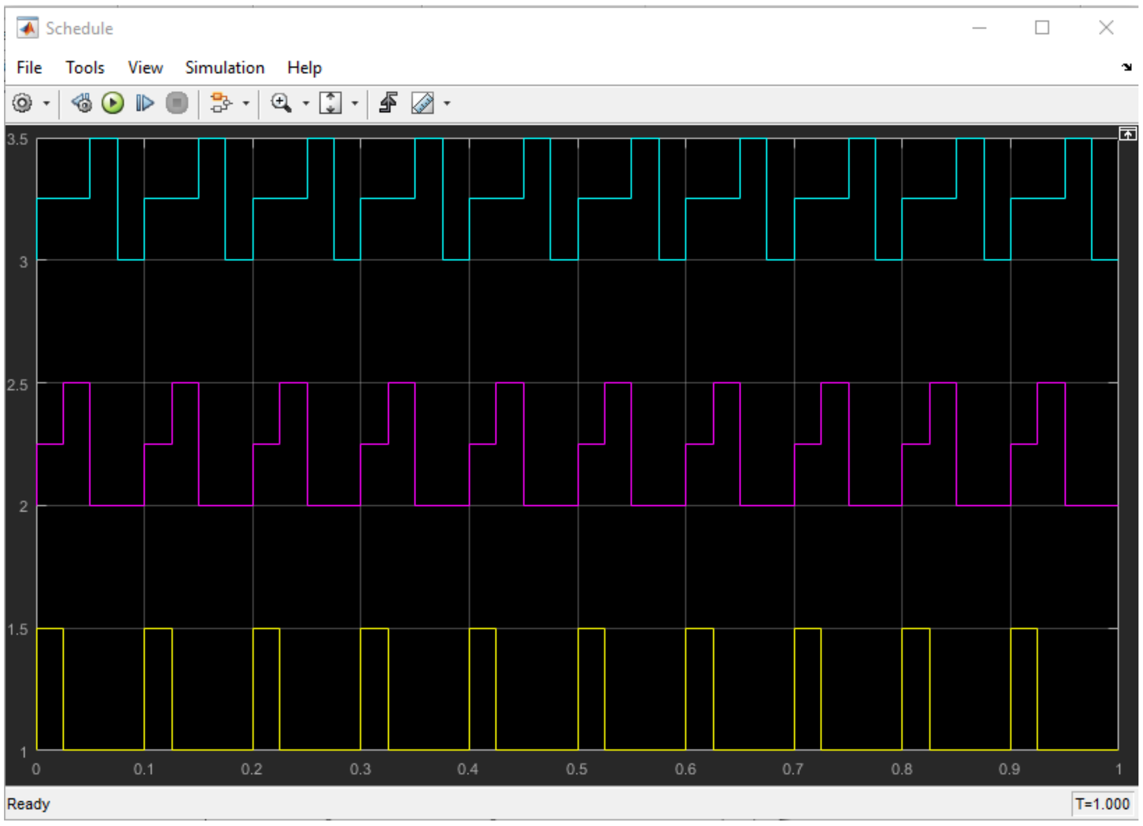


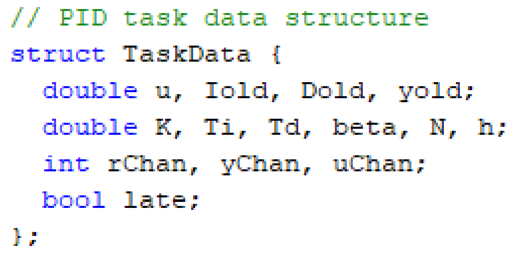
Figure 39: TrueTime, result of the scheduling simulation

In the *Schedule* monitor we can see a row for the scheduling decisions of each task and a different value on the graph means that the task is in a certain state at that time. Explaining the concept more clearly: the "purple" task is not released when the graph value is 2, it is in ready state when the graph value is 2.25 and it is in running state when the graph value is 2.5. The same applies also to the other two tasks.

2.4.1 The TrueTime kernel block

In this section the TrueTime kernel block code is explained in more detail, presenting the most relevant parts of the code in order to better understand how the kernel works and how the scheduling decisions are made. The most important aspect to understand is that the code associated to the TrueTime kernel block is used to define the tasks, their parameters and the schedulers used in the simulation. The true kernel and scheduler logics are not written here, but are written instead in external C++ modules that will be explained in more detail later, in the section that explains how the separation between kernel and scheduler has been done.

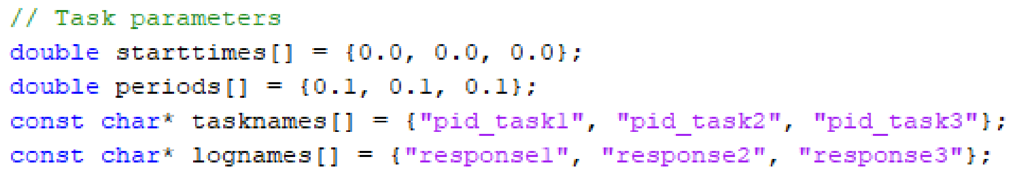
First, in the TrueTime kernel block code the data structure for the tasks must be defined. It contains all the parameters related to the transfer function, since the tasks are simulated using a transfer function.



```
// PID task data structure
struct TaskData {
  double u, Iold, Dold, yold;
  double K, Ti, Td, beta, N, h;
  int rChan, yChan, uChan;
  bool late;
};
```

Figure 40: TrueTime kernel block code, TaskData definition

After the task data structure has been defined, some variables containing the task parameters (like periods and start times) must be defined and initialized in this way:

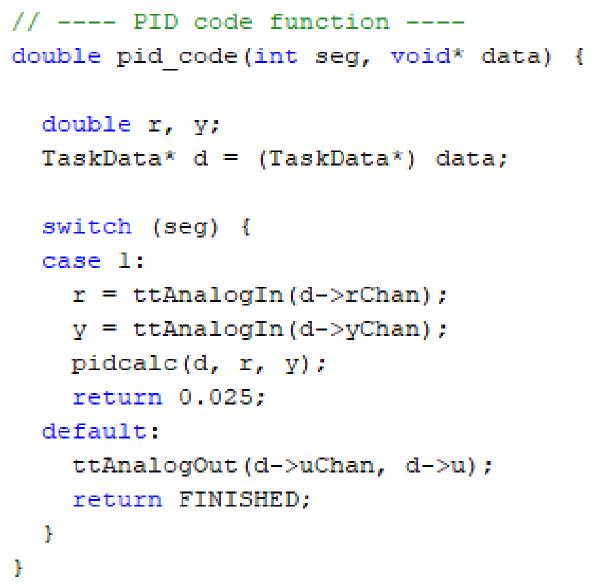


```
// Task parameters
double starttimes[] = {0.0, 0.0, 0.0};
double periods[] = {0.1, 0.1, 0.1};
const char* tasknames[] = {"pid_task1", "pid_task2", "pid_task3"};
const char* lognames[] = {"response1", "response2", "response3"};
```

Figure 41: TrueTime kernel block code, task parameters definition

Obviously in this case the task parameters are defined as arrays, and each element of the array corresponds to a certain task.

After the definition of all the data structure and parameters regarding the tasks, the PID code function must be written.

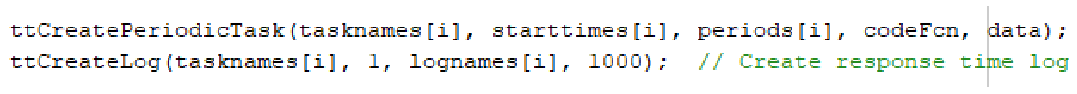


```c
// ---- PID code function ----
double pid_code(int seg, void* data) {

  double r, y;
  TaskData* d = (TaskData*) data;

  switch (seg) {
  case 1:
    r = ttAnalogIn(d->rChan);
    y = ttAnalogIn(d->yChan);
    pidcalc(d, r, y);
    return 0.025;
  default:
    ttAnalogOut(d->uChan, d->u);
    return FINISHED;
  }
}
```

Figure 42: TrueTime kernel block code, PID code function

The most interesting part of this piece of code is the switch that uses the concept of segment (written as *seg* in the code). In TrueTime the segment represents the piece of code that should be executed by the simulated task; the code execution is also simulated using the transfer function that takes the input of the PID block, use them in the *pidcalc* function and returns the execution time; all this is done if the segment value is 1, that is the task has not yet completed execution. If *seg* has a value different from 1 it means that the task has finished execution, so the outputs of the PID block are written.

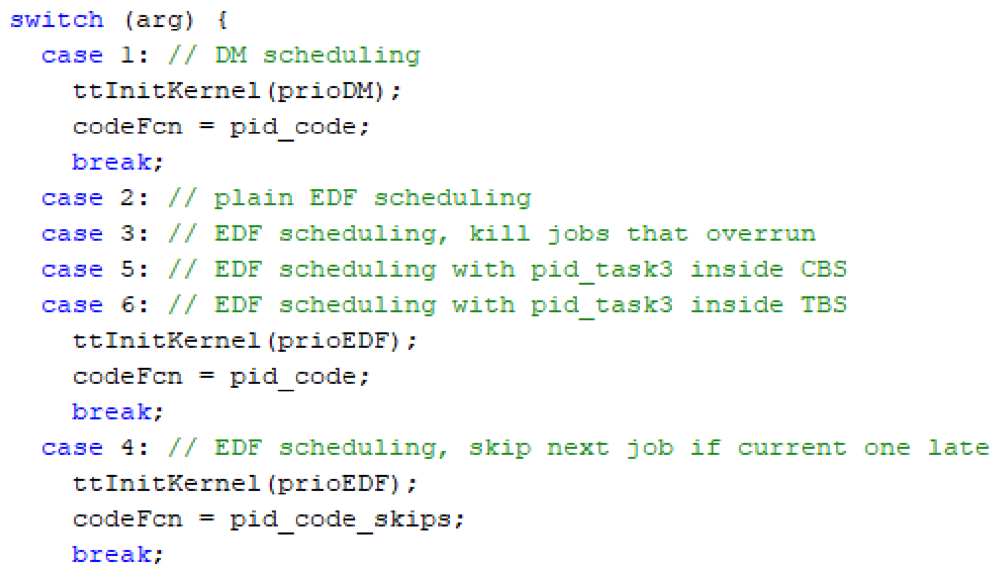
After the previous steps, the tasks must be created using the *ttCreatePeriodicTask* function, passing the task parameters previously defined:



```c
ttCreatePeriodicTask(tasknames[i], starttimes[i], periods[i], codeFcn, data);
ttCreateLog(tasknames[i], 1, lognames[i], 1000);  // Create response time log
```

Figure 43: TrueTime kernel block code, task creation

Then, in the main function of the TrueTime kernel block, the scheduler must be set depending on the user choice. The user can choose the scheduler opening the kernel block and inserting a numerical value, depending on this value the scheduler used in the simulation will differ.



```
switch (arg) {
  case 1: // DM scheduling
    ttInitKernel(prioDM);
    codeFcn = pid_code;
    break;
  case 2: // plain EDF scheduling
  case 3: // EDF scheduling, kill jobs that overrun
  case 5: // EDF scheduling with pid_task3 inside CBS
  case 6: // EDF scheduling with pid_task3 inside TBS
    ttInitKernel(prioEDF);
    codeFcn = pid_code;
    break;
  case 4: // EDF scheduling, skip next job if current one late
    ttInitKernel(prioEDF);
    codeFcn = pid_code_skips;
    break;
```

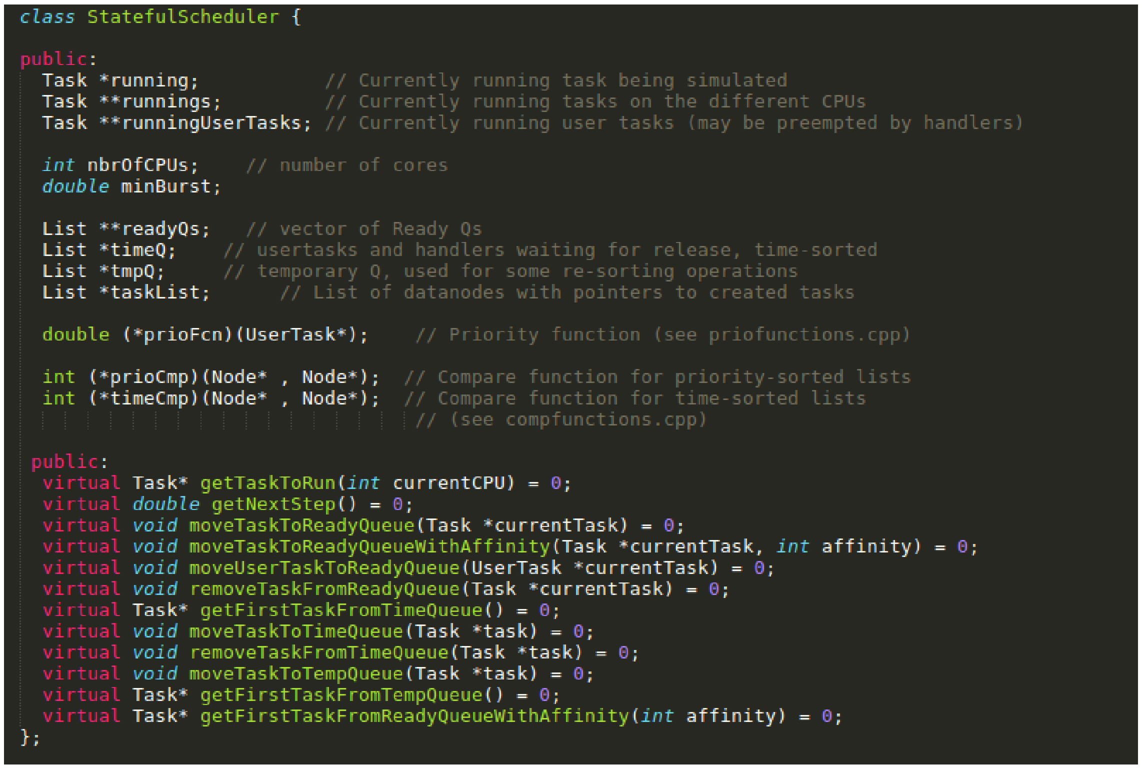
Figure 44: TrueTime kernel block code, scheduler selection

In this case, for instance, with value 1 the simulation will run with the Dead Monotonic scheduler, with value 2 the simulation will run with the Earliest Deadline First scheduler and so on. *prioDM* and *prioEDF* are priority functions that tell the kernel how to manage the ready queue, they will be explained in more datail later.

2.4.2 Separating the schedulers from the kernel

This section will explain the procedure used in order to separate the kernel logic from the scheduler logic in the TrueTime kernel. This has been done because, before this change, the two logic were not clearly discernable and it was difficult to understand which instructions were relevant to the scheduler and which were relevant only to the kernel. In this way the code has become much more readable and, in future, implementing new feature will be much easier, particularly if it is a scheduler feature. In fact, this operation was an intermediate step in order to make the implementation of the Round Robin scheduler easier.

First, the *StatefulScheduler* header file has been defined. It contains the *StatefulScheduler* class that is the main (and general) scheduler class from which the other schedulers will inherit.



```
class StatefulScheduler {

public:
  Task *running;           // Currently running task being simulated
  Task **runnings;         // Currently running tasks on the different CPUs
  Task **runningUserTasks; // Currently running user tasks (may be preempted by handlers)

  int nbrOfCPUs;    // number of cores
  double minBurst;

  List **readyQs;   // vector of Ready Qs
  List *timeQ;    // usertasks and handlers waiting for release, time-sorted
  List *tmpQ;     // temporary Q, used for some re-sorting operations
  List *taskList;      // List of datanodes with pointers to created tasks

  double (*prioFcn)(UserTask*);    // Priority function (see priofunctions.cpp)

  int (*prioCmp)(Node* , Node*);  // Compare function for priority-sorted lists
  int (*timeCmp)(Node* , Node*);  // Compare function for time-sorted lists
                                  // (see compfunctions.cpp)

 public:
  virtual Task* getTaskToRun(int currentCPU) = 0;
  virtual double getNextStep() = 0;
  virtual void moveTaskToReadyQueue(Task *currentTask) = 0;
  virtual void moveTaskToReadyQueueWithAffinity(Task *currentTask, int affinity) = 0;
  virtual void moveUserTaskToReadyQueue(UserTask *currentTask) = 0;
  virtual void removeTaskFromReadyQueue(Task *currentTask) = 0;
  virtual Task* getFirstTaskFromTimeQueue() = 0;
  virtual void moveTaskToTimeQueue(Task *task) = 0;
  virtual void removeTaskFromTimeQueue(Task *task) = 0;
  virtual void moveTaskToTempQueue(Task *task) = 0;
  virtual Task* getFirstTaskFromTempQueue() = 0;
  virtual Task* getFirstTaskFromReadyQueueWithAffinity(int affinity) = 0;
};
```

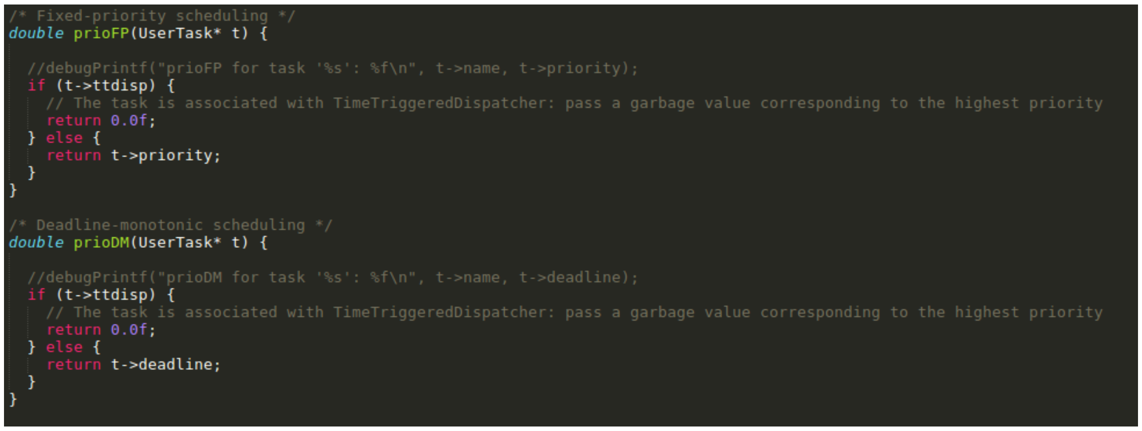
Figure 45: TrueTime, StatefulScheduler class definition

This class contains the parameters that were previously defined in the kernel class but are relevant to the scheduler logic. In particular, we can see:

- the *running* field, that contains the currently running task
- the *readyQs* field, which represents the queues of ready tasks
- the *timeQ* field, which represents the queue of tasks that are waiting for release

- the *prioFcn* field, which is a pointer to the priority function used by the scheduler to manage the ready queue.

The concept of the priority function needs a little more explanation to be clear. In order to determine the task that must be run, the scheduler (and formerly the kernel) usese the ready queue: the first task in the ready queue is the one that must be in running state. This means that at every moment of the simulation the ready queue must be ordered according to the scheduling policy. For instance, in the case of Fixed Priority the queue must be ordered according to the tasks priority, or in the case of Earliest Deadline First the queue must be ordered according to the tasks deadline. The priority function is used to compare the tasks in the ready queue and order them according to the scheduling policy. As we can see in the picture below, the priority function for the Fixed Priority scheduler uses the task priority, while the priority function for the Deadline Monotonic scheduler uses the deadline.

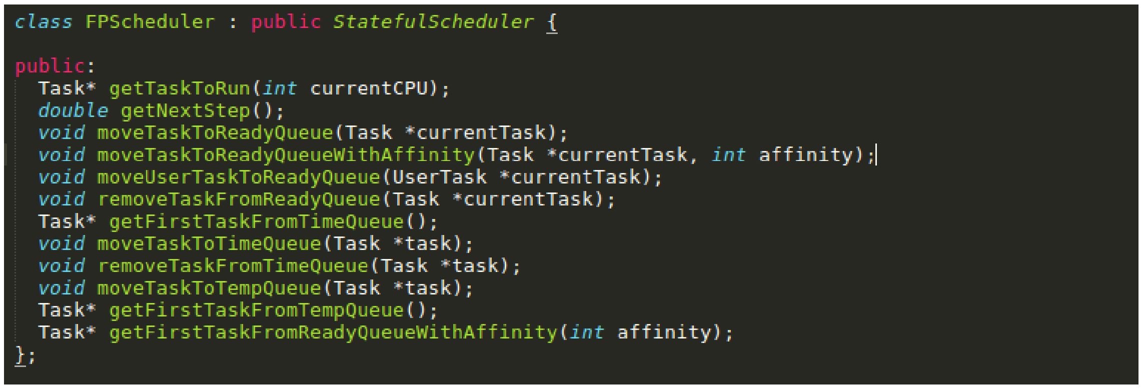


```
/* Fixed-priority scheduling */
double prioFP(UserTask* t) {

  //debugPrintf("prioFP for task '%s': %f\n", t->name, t->priority);
  if (t->ttdisp) {
    // The task is associated with TimeTriggeredDispatcher: pass a garbage value corresponding to the highest priority
    return 0.0f;
  } else {
    return t->priority;
  }
}

/* Deadline-monotonic scheduling */
double prioDM(UserTask* t) {

  //debugPrintf("prioDM for task '%s': %f\n", t->name, t->deadline);
  if (t->ttdisp) {
    // The task is associated with TimeTriggeredDispatcher: pass a garbage value corresponding to the highest priority
    return 0.0f;
  } else {
    return t->deadline;
  }
}
```

Figure 46: TrueTime, priority functions implementation

The methods contained in the *StatefulScheduler* class are all virtual method since they are implemented in the scheduler subclasses, so they will be explained shortly.

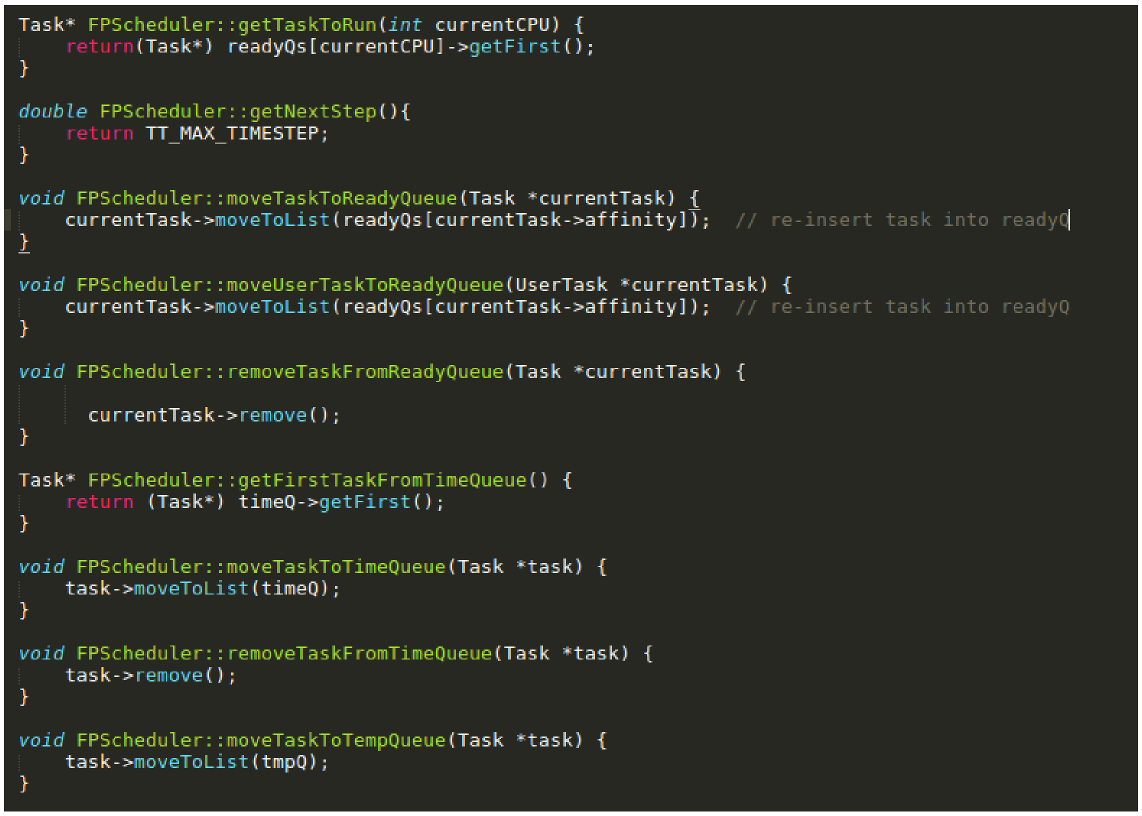
After the definition of the *StatefulScheduler* header, a scheduler subclasses must be defined for each scheduler implemented in TrueTime. Since the already present scheduler are Fixed Priority, Earliest Deadline First and Rate Monotonic, three subclasses have to be defined. The following pictures shows the Fixed Priority scheduler class (called *FPScheduler*) as defined in the header.



```
class FPScheduler : public StatefulScheduler {

public:
  Task* getTaskToRun(int currentCPU);
  double getNextStep();
  void moveTaskToReadyQueue(Task *currentTask);
  void moveTaskToReadyQueueWithAffinity(Task *currentTask, int affinity);
  void moveUserTaskToReadyQueue(UserTask *currentTask);
  void removeTaskFromReadyQueue(Task *currentTask);
  Task* getFirstTaskFromTimeQueue();
  void moveTaskToTimeQueue(Task *task);
  void removeTaskFromTimeQueue(Task *task);
  void moveTaskToTempQueue(Task *task);
  Task* getFirstTaskFromTempQueue();
  Task* getFirstTaskFromReadyQueueWithAffinity(int affinity);
};
```

Figure 47: TrueTime, FPScheduler class implementation

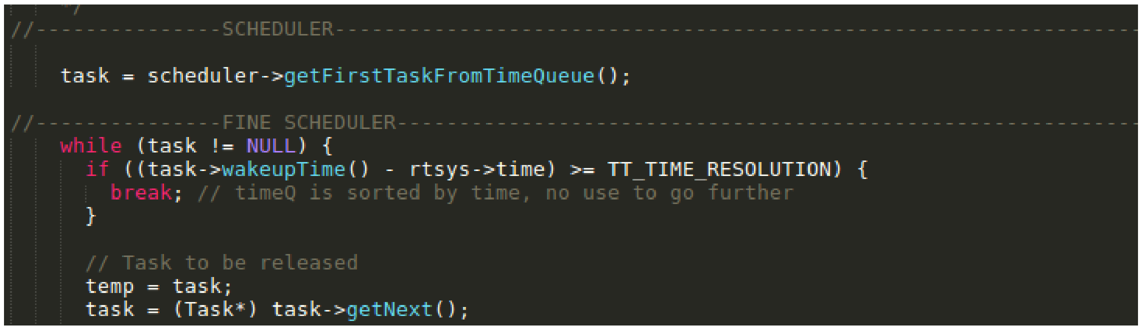
The class does not have any fields since the needed fields are inherited from the *StatefulScheduler* superclass. The methods shown are implemented in the source file; in order to not change the TrueTime kernel behavior, these methods are only wrapper that wrap the instructions used when there was not a separation between kernel and scheduler.



```
Task* FPScheduler::getTaskToRun(int currentCPU) {
    return(Task*) readyQs[currentCPU]->getFirst();
}

double FPScheduler::getNextStep(){
    return TT_MAX_TIMESTEP;
}

void FPScheduler::moveTaskToReadyQueue(Task *currentTask) {
    currentTask->moveToList(readyQs[currentTask->affinity]);  // re-insert task into readyQ
}

void FPScheduler::moveUserTaskToReadyQueue(UserTask *currentTask) {
    currentTask->moveToList(readyQs[currentTask->affinity]);  // re-insert task into readyQ
}

void FPScheduler::removeTaskFromReadyQueue(Task *currentTask) {

      currentTask->remove();
}

Task* FPScheduler::getFirstTaskFromTimeQueue() {
    return (Task*) timeQ->getFirst();
}

void FPScheduler::moveTaskToTimeQueue(Task *task) {
    task->moveToList(timeQ);
}

void FPScheduler::removeTaskFromTimeQueue(Task *task) {
    task->remove();
}

void FPScheduler::moveTaskToTempQueue(Task *task) {
    task->moveToList(tmpQ);
}
```

Figure 48: TrueTime, FPScheduler methods implementation

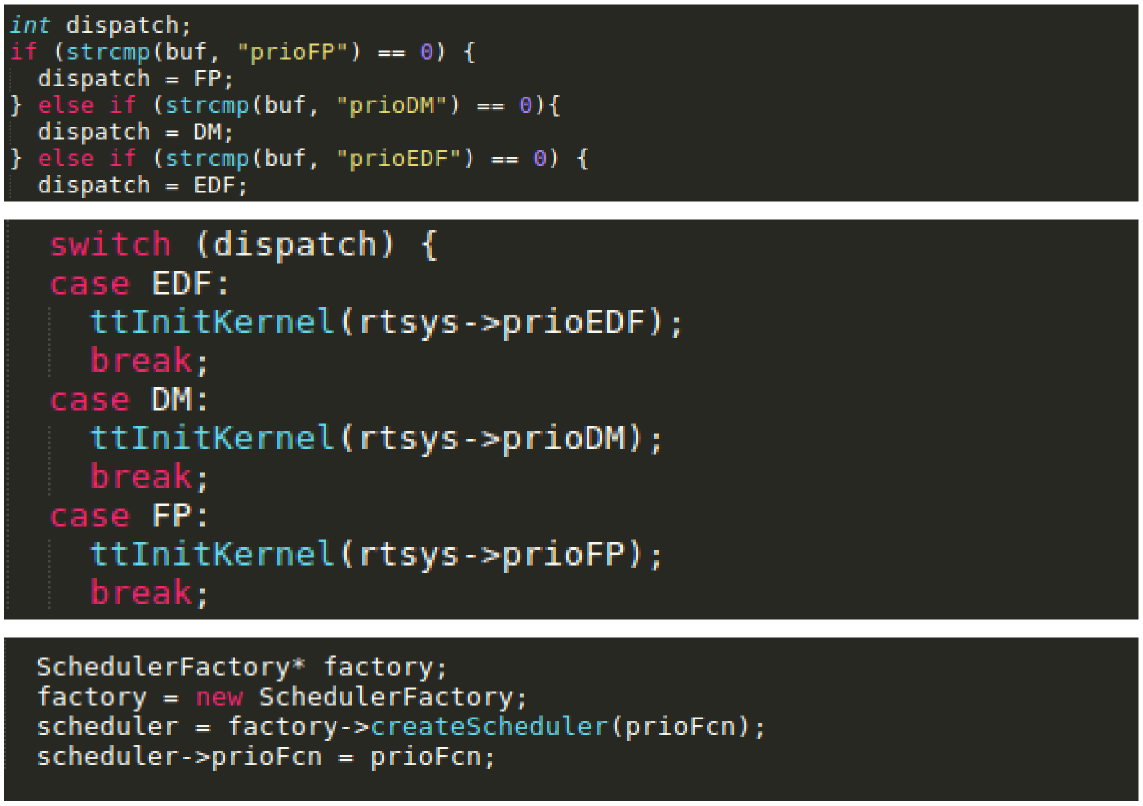
After wrapping all the needed instructions in the kernel code, it is clear what operations are done by the scheduler, since all the scheduling decisions and actions are methods of the scheduler object.



```
//---------------SCHEDULER----------------------------------------------------------------

    task = scheduler->getFirstTaskFromTimeQueue();

//---------------FINE SCHEDULER-----------------------------------------------------------
    while (task != NULL) {
      if ((task->wakeupTime() - rtsys->time) >= TT_TIME_RESOLUTION) {
        break; // timeQ is sorted by time, no use to go further
      }

      // Task to be released
      temp = task;
      task = (Task*) task->getNext();
```

Figure 49: TrueTime, example of the wrapping done by the scheduler methods

In order to make this all work, the correct scheduler object must be instantiated accordingly to the scheduler chosen by the user: since the TrueTime kernel block code passes the priority function to the kernel code, this is used to determine which scheduler must be created.



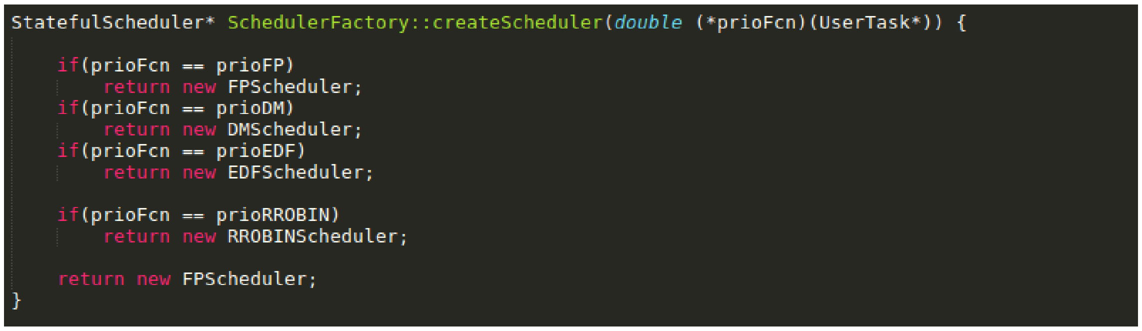
```
int dispatch;
if (strcmp(buf, "prioFP") == 0) {
  dispatch = FP;
} else if (strcmp(buf, "prioDM") == 0){
  dispatch = DM;
} else if (strcmp(buf, "prioEDF") == 0) {
  dispatch = EDF;
```

```
  switch (dispatch) {
  case EDF:
    ttInitKernel(rtsys->prioEDF);
    break;
  case DM:
    ttInitKernel(rtsys->prioDM);
    break;
  case FP:
    ttInitKernel(rtsys->prioFP);
    break;
```

```
  SchedulerFactory* factory;
  factory = new SchedulerFactory;
  scheduler = factory->createScheduler(prioFcn);
  scheduler->prioFcn = prioFcn;
```

Figure 50: TrueTime, creation of the right scheduler using the SchedulerFactory class

SchedulerFactory is a class used for instantiating the correct scheduler depending on the priority function passed by the block kernel code. The method *createScheduler* is defined as follows:

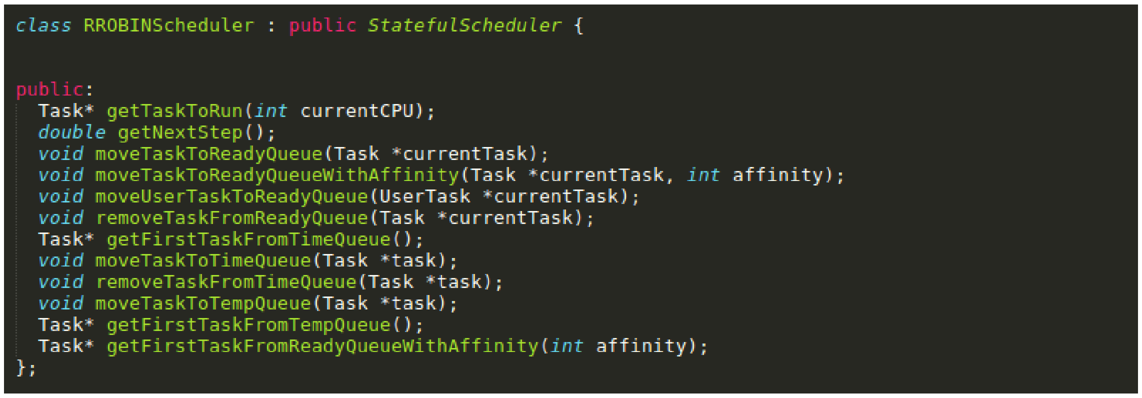


```
StatefulScheduler* SchedulerFactory::createScheduler(double (*prioFcn)(UserTask*)) {

    if(prioFcn == prioFP)
        return new FPScheduler;
    if(prioFcn == prioDM)
        return new DMScheduler;
    if(prioFcn == prioEDF)
        return new EDFScheduler;

    if(prioFcn == prioRROBIN)
        return new RROBINScheduler;

    return new FPScheduler;
}
```

Figure 51: TrueTime, createScheduler method implementation

2.4.3 Implementing the Round Robin scheduler

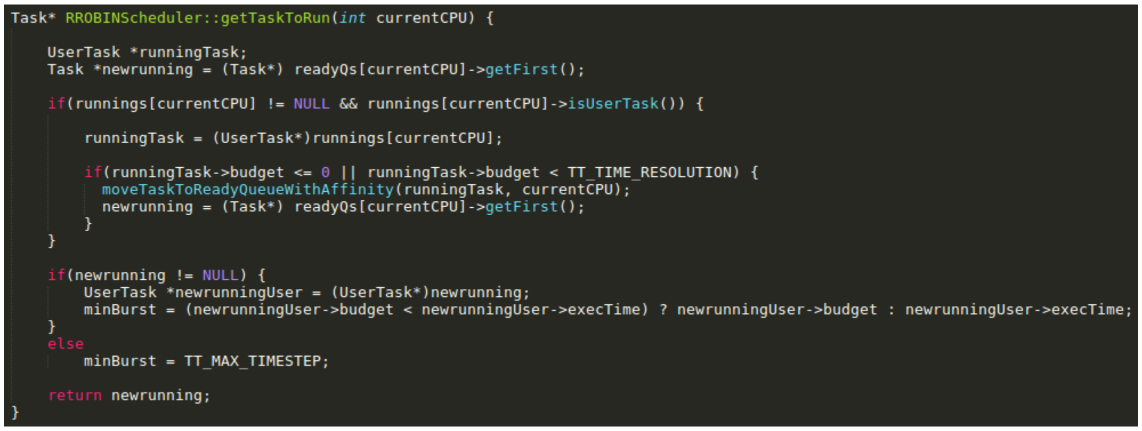
In order to implement the Round Robin scheduler in TrueTime, a new class for this type of scheduler must be created. The class structure is essentially the same as the previously implemented scheduler.



```
class RROBINScheduler : public StatefulScheduler {


public:
  Task* getTaskToRun(int currentCPU);
  double getNextStep();
  void moveTaskToReadyQueue(Task *currentTask);
  void moveTaskToReadyQueueWithAffinity(Task *currentTask, int affinity);
  void moveUserTaskToReadyQueue(UserTask *currentTask);
  void removeTaskFromReadyQueue(Task *currentTask);
  Task* getFirstTaskFromTimeQueue();
  void moveTaskToTimeQueue(Task *task);
  void removeTaskFromTimeQueue(Task *task);
  void moveTaskToTempQueue(Task *task);
  Task* getFirstTaskFromTempQueue();
  Task* getFirstTaskFromReadyQueueWithAffinity(int affinity);
};
```

Figure 52: TrueTime, RROBINScheduler class definition

While also most of the methods implementation is the same, the big difference is the *getTaskToRun* method: this method returns to the kernel the task that must be in running time at that particular moment. For the other schedulers this method is just a wrapper for the method *getFirst* of the queue class that returns the first task in the queue (because as explained earlier the first task in the ready queue is the task that must be running), but in the case of the Round Robin scheduler is a little more complex. The code is as shown in the picture below.



```
Task* RROBINScheduler::getTaskToRun(int currentCPU) {

    UserTask *runningTask;
    Task *newrunning = (Task*) readyQs[currentCPU]->getFirst();

    if(runnings[currentCPU] != NULL && runnings[currentCPU]->isUserTask()) {

        runningTask = (UserTask*)runnings[currentCPU];

        if(runningTask->budget <= 0 || runningTask->budget < TT_TIME_RESOLUTION) {
            moveTaskToReadyQueueWithAffinity(runningTask, currentCPU);
            newrunning = (Task*) readyQs[currentCPU]->getFirst();
        }
    }

    if(newrunning != NULL) {
        UserTask *newrunningUser = (UserTask*)newrunning;
        minBurst = (newrunningUser->budget < newrunningUser->execTime) ? newrunningUser->budget : newrunningUser->execTime;
    }
    else
        minBurst = TT_MAX_TIMESTEP;

    return newrunning;
}
```

Figure 53: TrueTime, getTaskToRun method implementation

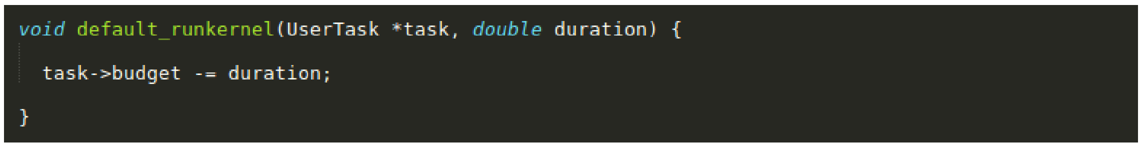
The *getTaskToRun* method, in order to return the correct task to run, does the following:

- in order to implement the preemption when a task finishes its burst, if there is currently a running task and that task is a *UserTask* (meaning that is not a timer) and that task has finished its burst, the function moves the task back to the ready queue. In this way the task is placed in the ready queue, which is then automatically ordered accordingly to the priority function. Since the priority function for the Round Robin scheduler returns the priority of the task, the queue is ordered accordingly to the tasks priorities. In the case of the Round Robin scheduler, the priorities are set to their default value and cannot be changed, in this way, since the priorities are all equal, the tasks order in the ready queue is based on each task's arrival time. This behavior guarantees the cycling of the tasks typical of the Round Robin scheduler.

- After the preempted task is placed back in the ready queue, the method uses the *getFirst* function in order to take the task that must be run (since the queue is now correctly ordered as explained before).

- If the task that must be run is not NULL (meaning that the cpu is not in idle state but has to run a task), then the next burst for the task (here called *minBurst*) is computed. Its value will be the minimun between the task budget (which in TrueTime has the same value of the WCET) and the remaining execution time (here called *execTime*). This is done in order to guarantee that when the remaining execution time for a task is less that the assigned budget, the task will be preempted.

The last step that must be done in order to make the Round Robin scheduler work is to decrement the budget of the task while it runs. This is done using what in TrueTime is called hooks: hooks are pieces of code that are executed in predetermined moments of the simulation. For example there is a *runkernel* hook that is executed when the kernel runs, a *release* hook that is executed when a task is released, and so on. In our case, the runkernel hook is used to decrement the budget for the running task.



```
void default_runkernel(UserTask *task, double duration) {
  task->budget -= duration;
}
```

Figure 54: TrueTime, task budget decrease

3 Conclusions

In the course of this paper, some tools for real-time schedulers (and in general systems) have been described and analyzed.

Cheddar is a very powerful tool regarding the scheduling simulation, because it allows the user to customize nearly every aspect of the system that will be simulated, it also let the user define custom schedulers. The main Cheddar downside is that, despite the high level of customization, the results don't deliver much information to the user since they mainly focus on the feasibility of the schedule.

RTSim is a much simpler tool than Cheddar, if the user is able to write C++ code, he/she can create a simulation very quickly. This obviously comes with a downside, that is the fact that RTSim does not offer the level of customization that Cheddar does. The results given by RTSim are similar to the ones given by Cheddar: they primarly focus on scheduling feasibility and the scheduling decisions, but don't go further than that.

Litmus is very different from the scheduling simulation tools just described: it doesn't focus on giving the user a scheduling scheme in which the scheduling decision are shown, but, since it is a real kernel and not a simulation tool, it focuses on letting the user test the task behavior on a real operating system. The obvious advantage is that the results that the user can obtain from using Litmus are much more reliable that the ones obtained from a simulation tool. The main Litmus disadvantages is its inability to provide complete support for a real scheduler customization; it's true that it let the user define new scheduler plugins but, as shows in the Litmus section, there are many aspect that the user cannot customize and that he/she cannot control.

TrueTime is like the tradeoff between the elements that we have been talking about in this section: it provides the means for the user to test the feasibility of a schedule, while customizing the tasks that have to run in the schedule and deciding which scheduler to use. TrueTime also allows the user to customize the schedulers and adding new ones, not only that, the user can customize the entire kernel code. The main downside is that the kernel code is a bit convoluted and it can be hard to fully understand what each piece of code does. With our contribution in separating the kernel and scheduler code in TrueTime, we hope to have made the code more readable for future developers and users who want to further customize the TrueTime kernel and scheduler. Also, with the addition of the Round Robin scheduler, we hope to have made the TreuTime system even more complete that it already was.

One final note worth of notice is that, while all the tools for real-time scheduling simulation provides the means to obtain scheduling statistics and a graphical representation of the scheduling decisions, only a few allow the developer to really define and customize new schedulers. Even the tools that have such feature (like Litmus), it is not really supported and the customization options are not complete. Our hope is that, in future, many more tools for real-time scheduling simulation will provide this feature, in order to make the definition of new schedulers and scheduler policies easier.

References

[1] F. Singhoff, J. Legrand, L. Nana, L. Marcé. *Cheddar : a Flexible Real Time Scheduling Framework*, 2004.

[2] Cheddar project main page. http://beru.univ-brest.fr/~singhoff/cheddar/

[3] RTSim home page. http://rtsim.sssup.it/

[4] Litmus$^{\mathrm{RT}}$ main page. https://www.litmus-rt.org/

[5] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson: *LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers*, 2006.

[6] B. Brandenburg: *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*, 2011.

[7] Litmus$^{\mathrm{RT}}$ manual. https://www.litmus-rt.org/tutorial/manual.html

[8] Writing a new Litmus$^{\mathrm{RT}}$ scheduler plugin. https://www.litmus-rt.org/create_plugin/create_plugin.html

[9] TrueTime main page. http://www.control.lth.se/truetime/

[10] Anton Cervin, Dan Henriksson, Martin Ohlin: *TrueTime 2.0 – Reference Manual*, 2016.

[11] Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, Karl-Erik Årzén: *How Does Control Timing Affect Performance? Analysis and Simulation of Timing Using Jitterbug and True-Time*. IEEE Control Systems Magazine, 23:3, pp. 16–30, June 2003.

[12] M. Maggio, F. Terraneo, A. Leva: *Task scheduling: a control-theoretical viewpoint for a general and flexible solution*. ACM Transactions on Embedded Computing Systems (TECS), Volume 13, Issue 4, pp. 1-22, Feburary 2014.

[13] Arezou Mohammadi, Selim G. Akl: *Scheduling Algorithms for Real-Time Systems*, 2005.