STORM

Simulation TOol for Real-time Multiprocessor scheduling

Designer Guide V3.3.1 – September 2009

Richard Urunuela, Anne-Marie Déplanche, Yvon Trinquet

This work is part of the project PHERMA supported by ANR grants ANR-06-ARFU-003.

1. Introduction

STORM that stands for "Simulation TOol for Real time Multiprocessor scheduling", is a multiprocessor scheduling simulation and evaluation platform. Figure 1 gives an overview of the STORM platform architecture. For the time being, engineering efforts on STORM have concerned its simulator component since it is the retaining element of the platform. For a given "problem" i.e. a software application that has to run on a (multiprocessor) hardware architecture, this simulator is able to "play its execution" over a specified time interval while taking into account the requirements of tasks, the characteristics and functioning conditions of hardware components and the scheduling rules.

As shown in Figure 2, the specification of the architectures and scheduling policy to be simulated is done via an XML file. The result of simulation is a set of execution tracks that are either made directly observable through diagrams, or recorded into files for a subsequent computer-aided analysis. All these results allow the user to analyze the behaviour of the system (tasks, processors, timing, performances, etc.).

STORM is a free, flexible and portable tool : i) "free" because the STORM software is freeware under *Creative Commons* License ; ii) "flexible" because it offers the possibility to the user to program and add new components through well defined API(s) with the simulation kernel; iii) "portable" because it is possible to run it on various OS thanks to the Java programming language.

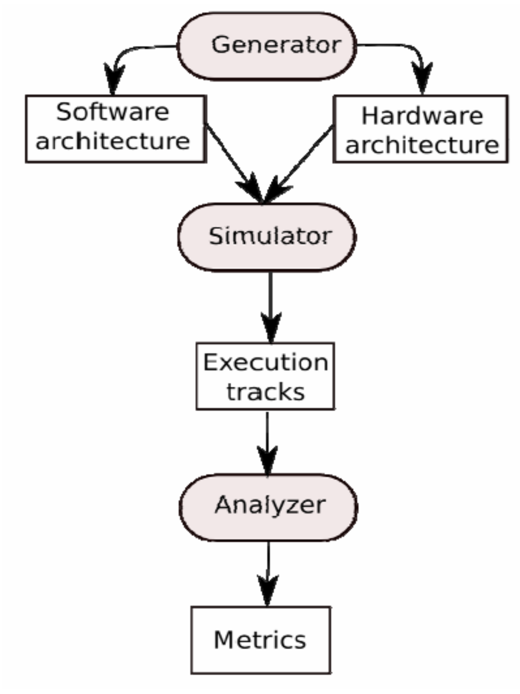

Figure 1. The STORM platform.

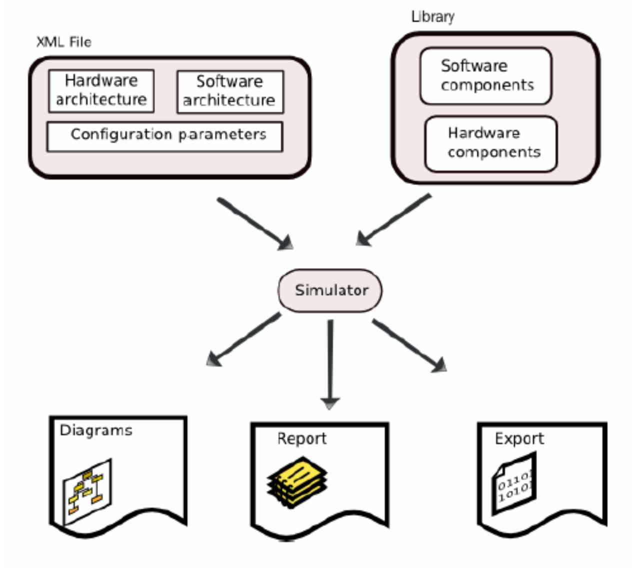

Figure 2. The STORM simulator.

The objective of this guide is particularly to give information on the way to design and add new scheduler component(s) into the STORM library.

The installation procedure and user guide of the STORM software are available on the web site: http://storm.rts-software.org/.

2. The architecture of the STORM simulator

2.1. The functional architecture

The architecture of the STORM simulator is composed of a set of entities built around a simulation kernel (see Figure 3). Software entities stand for the tasks and data that compose the software architecture for which the simulation is conducted. Up to now, hardware architectures are composed of processors only; that's why processor entities are the only hardware ones. At the moment, system entities are the task list manager, the scheduler and the memory manager. The scheduler entity is in charge of sharing the processor(s) between the ready tasks; its election rules depending on the scheduling strategy it implements.

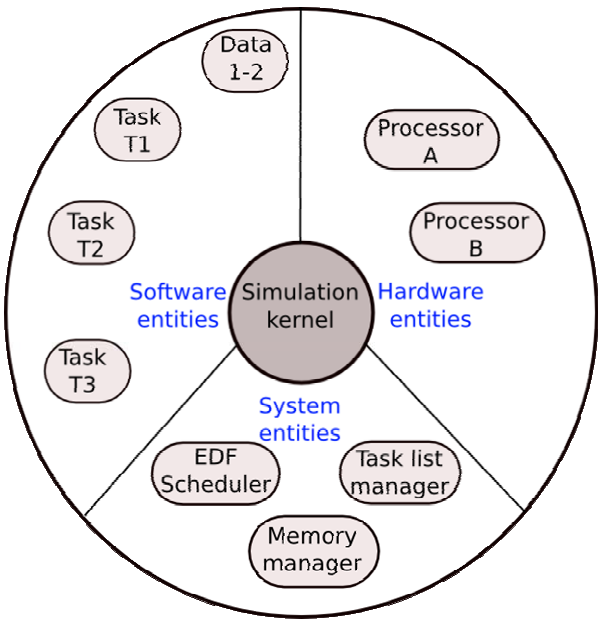

Figure 3. The simulator architecture.

2.2. *The software architecture*

The STORM simulator is written in Java programming language which makes it independent of any execution platform. Figure 4 gives a simplified view of the current UML class diagram of STORM

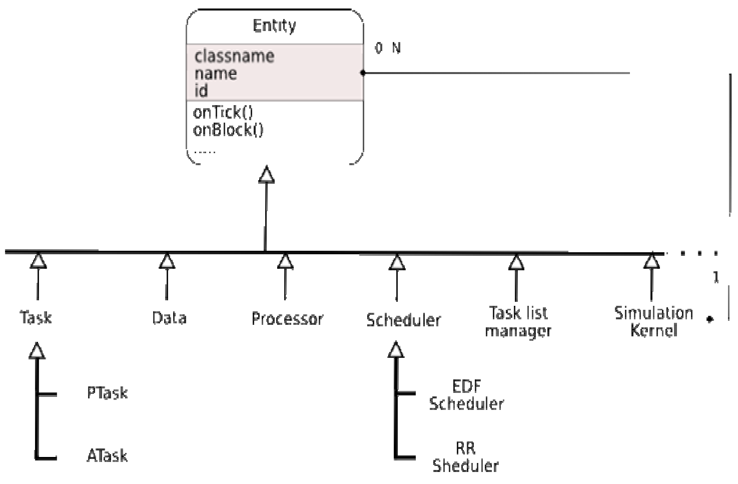

Figure 4. The UML STORM class diagram.

In accordance with Figure 3, we can find there, on the one side, the `SimulationKernel` final class that implements the kernel of the simulator, and on the other side, a set of classes (`Task`, `Data`, `Processor`, `Scheduler`, `TaskListManager`, etc.) and their subclasses that model the various simulation entities (some of them correspond to the value that has to be given to the `className` attributes in the input xml file).

All of the subclasses inherit from the superclass `Entity`. It contains the declaration of the methods involved in the implementation of the different services. Depending on the subclasses and the behaviour they must exhibit, those methods may need to be overrided.

3. The entities that interact with the scheduler component

The scheduler entity is led to interact mainly with: i) the task entities; ii) the processor entities; iii) the simulation kernel.

3.1. The task entity

There are as many task entities as specified in the xml input file. It is important to note that these entities capture the behaviour of the real components they represent only from a control viewpoint and not a functional one, i.e. no applicative programs run for the tasks. Whatever its type, the generic state diagram of a task entity is shown on Figure 5. At the very beginning, a task is unexisting. As soon as its first activation occurs, it becomes ready and falls under the control of the task list manager. Depending on the scheduling decisions, it may run (running state) and possibly be pre-empted. On its definitive completion, the task goes back into the unexisting state. On a job completion, it becomes waiting until all its execution conditions be met, i.e. only its next release in case of an independent task, but together with the availability of all the data it requires in case of a consumer task.

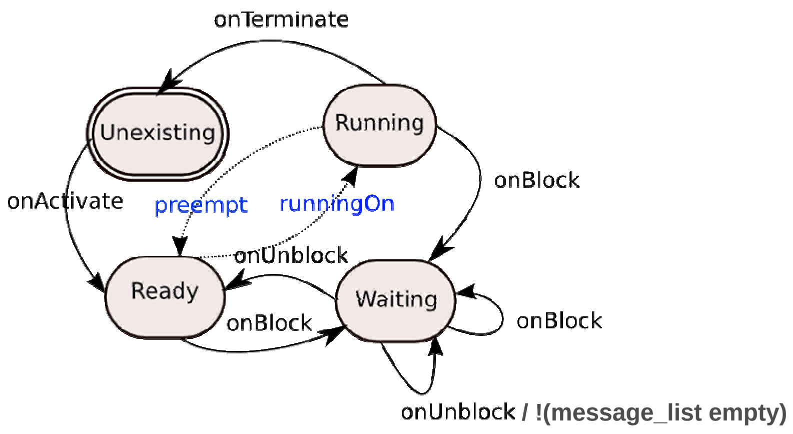

Figure 5. The task state diagram.

3.2. The processor entity

Each processor of the considered hardware architecture specified in the xml input file has its equivalent processor entity. Up to now no control is modeled in such an entity but instead it encapsulates some properties stating about its current activity (no operational, idle, busy with the running task identifier) and its current functional conditions (voltage, frequency in case of a processor with DVFS capabilities), together with the functions for updating them.

3.3. The simulation kernel

A simulation is achieved in a discrete way, i.e. the overall simulation interval is cut into a sequence of unitary slots [0,1), [1,2), …, [t,t+1), etc., and simulation moves forward at each instant 0, 1, …t, t+1, etc. Thus, at t, the next simulation state (for instant t+1) is computed from the current one (of instant t) and the pending simulation events at time t.

From a functional point of view, the kernel provides a very few basic services for managing simulation time and some interactions between entities. Only the *time service* may concern the designer of a scheduler component. It indicates the spending of time to the simulation entities. Thus, at each slot, all the simulation entities are informed that a unit of time has elapsed (of course, it can be ignored by the entity if it has no concern about time).

Due to our discrete approach for the simulation process, the behaviour of the simulation kernel is a cyclic one as shown by the Figure 6, one cycle for one slot. After a necessary initial step where all the simulation entities are created and the global time variable is initialized, the loop of cycles is entered. Any cycle is split into successive steps that come down to:
- a) manage the watchdogs, i.e. to detect those possible watchdogs (cf. the watchdog service no described here) that expire and to call the specified function of the specified entity;
- b) process all the currently pending kernel messages (it leads for the most to inform other simulation entities of the current situation; no described here);
- c) call upon the scheduler. We underline that it doesn't mean that the scheduling selection rules have to be applied at each slot: that is part of scheduling policy type and implementation choice;
- d) manage time passing, i.e. to inform all the simulation entities of a tick (cf. the aforementioned time service);
- e) ask for the task list manager to operate (no described here);
- f) increment the time variable.

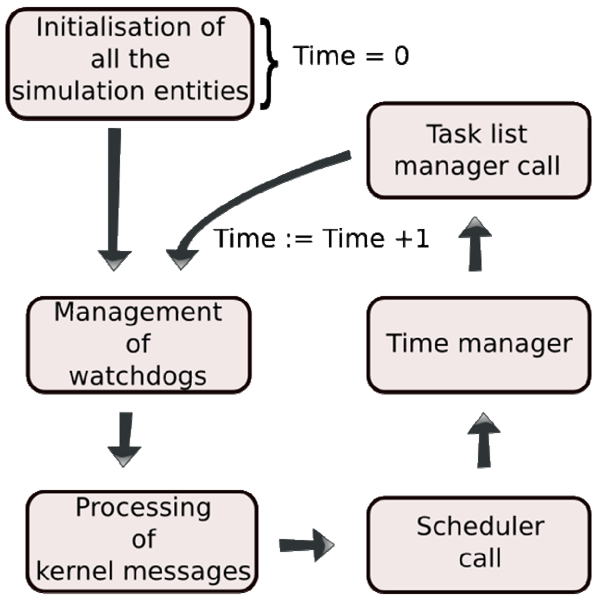

Figure 6. The simulation process.

4. The available APIs for the scheduler component

In this section, we give a description of the (part of the) interfaces of those classes that a designer may be concerned with while building a new scheduler component.

4.1. The `Task` interface

Table 1 gives the list of the public methods of the `Task` class that may be called on any `Task` instance.

Method	*Description*
`int getId()`	Returns the value of the `Id` attribute given to the task in the input xml file.
`int getActivationDate()`	Returns the value of the `activationDate` attribute of the task. It represents the release date (in slots) of the first job of the task.
`int getWCET()`	Returns the value of the `WCET` attribute of the task. It represents the worst case execution time (in slots) of the task.
`int getBCET()`	Returns the value of the `BCET` attribute of the task. It represents the best case execution time (in slots) of the task.
`int getAET()`	Returns the value of the `AET` attribute of the task. It represents the actual execution time (in slots) of the current job of the task.
`int getPeriod()`	Returns the value of the `period` attribute of the task. It represents the period (in slots) of the task.
`int getPriority()`	Returns the value of the `priority` attribute of the task. It represents the priority of the task.
`int getDeadline()`	Returns the value of the `deadline` attribute of the task. It represents the critical delay (in slots) of the task.
`void runningOn(Processor P)`	Assigns the task to the processor the reference of which is `P` and moves the task state from `Ready` to `Running` (cf. Figure 5).
`void preempt()`	Moves the task state from `Running` to `Ready` (cf. Figure 5). The processor on which the task was previously running becomes free.

Table 1. The `Task` interface.

4.2. The Processor API

Table 2 gives the list of the public methods of the Processor class that may be called on any Processor instance.

Method	*Description*
`int getId()`	Returns the value of the Id attribute given to the processor in the input xml file.
`boolean isRunning()`	Returns true if the processor is presently busy, otherwise false.
`Task getrunning()`	Returns the reference of the task instance that is presently running on the processor if the processor is busy, otherwise null.
`int getLoad()`	Returns the value of the processor load i.e. the number of slots (since the beginning of the simulation) where the processor has been running.
`void setRunning(Task T)`	Assigns the task the reference of which is T to the processor and moves the task state from Ready state to Running (cf. Figure 5).

Table 2. The Processor interface.

4.3. User-defined dynamic attributes

The Entity class provides additional methods that enable the use of dynamical attributes and map them on hashtables. By dynamical attributes, we mean attributes that are not explicitly declared as members in the Java code of a class, but that can be created, initialized and then updated thanks to specific function calls inside the methods of this class.

Method	*Description*
`void setOwnFieldIntValue(String name, int i)`	Creates (or updates if yet created) the integer dynamic attribute name with the value i.
`int getOwnFieldIntValue(String name)`	Returns the current value of the integer dynamic attribute name.
`void setOwnFieldStringValue(String name, String s)`	Creates (or updates if yet created) the string dynamic attribute name with the value s.
`int getOwnFieldStringValue(String name)`	Returns the current value of the string dynamic attribute name.

Table 3. Dynamic attribute methods.

4.4. The `SimulationKernel` interface

Table 4 gives the list of the public methods of the `SimulationKernel` class that may be called on the `SimulationKernel` instance on which STORM relies.

Method	*Description*
`int getDuration()`	Returns the value of the `duration` attribute (in slots) given to the simulation in the input xml file.
`TasksListeManager getTasksListeManager()`	Returns the reference on the object that implements the task list manager.

Table 4. The `SimulationKernel` interface.

4.5. The `TaskListeManager` interface

Table 5 gives the list of the public methods of the `TaskListeManager` class that may be called on the `TaskListeManager` instance returns by the `getTasksListeManager()` previously listed method.

Method	*Description*
`ArrayList getProcessors()`	Returns the reference on the `ArrayList` object that records all the processors of the hardware architecture.
`ArrayList getTasks()`	Returns the reference on the `ArrayList` object that records all the tasks that are presently either in the `Ready` state or the `Running` one (cf. Figure 5).
`ArrayList getAllTasks()`	Returns the reference on the `ArrayList` object that records all the tasks of the software architecture.

Table 5. The `TaskListeManager` interface.

5. Definition of a new scheduler

5.1. The `NewScheduler` class

Defining a new scheduler simply consists in creating a new Java class (we call it `NewScheduler` hereafter) that inherits from the abstract `Scheduler` class and :

- overriding (if necessary) the methods inherited from the `Entity` class that handle the task state transitions given in Figure 5 (except the `preempt` and `runningOn` ones);

- defining the abstract methods of the `Scheduler` class.

Table 6 gives the list of those methods inherited from the `Entity` class that may be defined in the `NewScheduler` class.

Method	*Description*
`void onBlock(EvtContext c)`	Handler called each time a task enters the `Waiting` state (cf. Figure 5). The reference on the `Task` object may be known by calling `c.getCible()`.
`void onUnBlock(EvtContext c)`	Handler called each time a task enters the `Ready` state (cf. Figure 5). The reference on the `Task` object may be known by calling `c.getSource()`.
`void onActivate(EvtContext c)`	Handler called each time a task leaves the `Unexisting` state (cf. Figure 5). The reference on the `Task` object may be known by calling `c.getCible()`.
`void onTerminated(EvtContext c)`	Handler called each time a task enters the `Unexisting` state (cf. Figure 5). The reference on the `Task` object may be known by calling `c.getCible()`.
`void onTick()`	Method called at each simulation cycle meaning that a simulation slot has elapsed (cf. `Time manager` on Figure 6). Be careful that it is called after the `sched()` method call.
`void init()`	Method called at the scheduler initialization.

Table 6. The methods to possibly override (inherited from the `Entity` class).

Table 7 gives the list of those abstract methods of the `Scheduler` class that need to be defined in the `NewScheduler` class.

Method	*Description*
`void select()`	Method that should implement the scheduling decision algorithm and be called judiciously by the `sched()` method.
`void sched()`	Method called at each simulation cycle (cf. `Scheduler call` on Figure 6). Be careful that it is called before the `onTick()` method call.

Table 7. The methods to define (inherited from the `Scheduler` class).

When a scheduling decision leads to:

- a task preemption, the method `preempt()` has to be called on the concerned `Task` instance;
- a task release, the method `runningOn()` has to be called on the concerned `Task` instance.

Figure 7 gives a simplified view of those various Java classes that are concerned together with the interactions between them. The circled numbers make reference to the number of the corresponding table of this document.

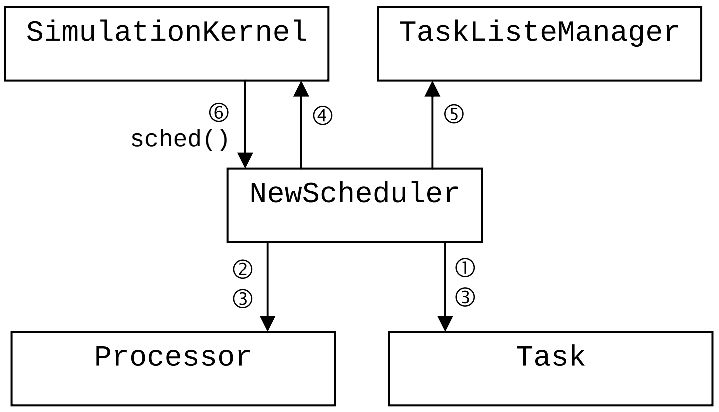

Figure 7. The Java classes and their interactions.

5.2. *An example*

Now let's see how a FIFO-queue scheduler can be added into STORM. We call its equivalent class: `FIFO_Scheduler`. It is a non-preemptive strategy and its scheduling rule is: at any time, the (m at most) oldest ready tasks run on the (m) processors.

The `FIFO_Scheduler` class has to declare and instance an attribute of Java `LinkedList` type for the ready task queue. Each time a task enters the `Ready` state (cf. its methods `onActivate()` and `onUnBlock()`), it has to be added to the end of this list (by calling its `addLast()` method). Each time a task leaves the `Running` state (cf. its methods `onTerminated()` and `onBlock()`), it has to be removed from this list (by calling its `remove()` method). The method `sched()` of the `FIFO_Scheduler` (that will be automatically called by the simulation kernel at each slot) has to be overloaded so as to detect which processor(s) is (are) idle and, if possible, to assign to it (them) the first non running task(s) of the ready task queue. The state change from `Ready` to `Running` for such a task simply requires to call the `onRunning` method of its equivalent object.

The Java program of the corresponding `FIFO_Scheduler` class is given hereafter:

```
package storm.Schedulers;
import java.util.*;
import storm.*;
import storm.Processors.*;
import storm.Tasks.*;

public class FIFO_Scheduler extends Scheduler {

    private LinkedList list_ready;

    public void init() {
        list_ready = new LinkedList();
    }

    public void onActivate(EvtContext c) {
        list_ready.addLast(c.getCible());
    }

    public void onUnBlock(EvtContext c){
        list_ready.addLast(c.getSource());
    }

    public void onBlock(EvtContext c){
        list_ready.remove(c.getCible());
    }

    public void onTerminated(EvtContext c){
        list_ready.remove(c.getCible());
    }

    public void sched(){
        select();
    }

    public void select() {
        ArrayList CPUS = this.Kernel.getTasksListeManager().getProcessors();
        Iterator CPU = CPUS.iterator();
        while (CPU.hasNext()) {
            Processor p = (Processor) CPU.next();
            if (!p.isRunning()) {
                Iterator task= list_ready.iterator();
                while (task.hasNext()) {
                    Task t=(Task) task.next();
                    if (!t.isIsrunning()) {
                        t.runningOn(p);
                        break;
                    }
                }
            }
        }
    }
}
```

5.3. An other example[1]

In this section, the case of a preemptive Earliest Deadline First scheduler is considered. We call its equivalent class: `EDF_P_Scheduler`. It is a strategy based on dynamic priorities for the tasks: at any time, the (m at most) highest priority ready tasks run on the (m)

[1] The Java code of the various scheduler classes presently implemented in STORM is available via the "`Getting Started`" page at http://storm.rts-software.org.

processors. The priority of a task depends on its dynamic critical delay, i.e. the remaining time until its next deadline: the smaller its dynamic critical delay is, the higher its priority is.

Since the current deadline of a task is not an intrinsic attribute (directly accessible via a getter method), it has to be defined thanks to the user-defined dynamic attribute facility. It is called `next_deadline` and is computed at the task activation and its subsequent job releases (see `T.setOwnFieldIntValue("next_deadline", this.Kernel.getTime() + T.getDeadline())`). The ready task queue needs to be sorted according to increasing `next_deadline` values. That's why the `EDF_P_Scheduler` class declares an inner class `LReady` that inherits the standard `ArrayList` class and implements the `compare` method based on the `next_deadline` of tasks. The ready task queue which is an instance of `LReady` (the reference of which is `list_ready`) can be easily sorted by calling the static method `Collections.sort(list_ready, list_ready)`.

Each time a task enters the `Ready` state (cf. its methods `onActivate()` and `onUnBlock()`), it has to be added to the `list_ready` list. Each time a task leaves the `Running` state (cf. its methods `onTerminated()` and `onBlock()`), it has to be removed from this list. The method `sched()` of the `EDF_P_Scheduler` (that will be automatically called by the simulation kernel at each slot) is overloaded by a simple `select` method call that implements the scheduling rule. So as to make the scheduler program more efficient, the new running tasks need to be computed only after a task state change (and not at each simulation slot). This is controlled thanks to the boolean variable `todo`. At first, the ready task queue has to be sorted. Then, since highest priority tasks are the m (at most) first ones, the other possibly running tasks have to be preempted. The state change from `Running` to `Ready` for such a task simply requires to call the `preempt` method of its equivalent object. At last, each non running task (among the m at most first ones) is assigned to an idle processor.

The Java program of the corresponding `EDF_P_Scheduler` class is given hereafter:

```
import storm.Schedulers.Scheduler;
import storm.*;
import storm.Processors.*;
import storm.Tasks.*;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;


public class EDF_P_Scheduler extends Scheduler {

    class LReady extends ArrayList implements Comparator {
        public int compare(Object arg0, Object arg1) {
            Task T0 = (Task) arg0;
            Task T1 = (Task) arg1;
            int d0 = T0.getOwnFieldIntValue("next_deadline");
            int d1 = T1.getOwnFieldIntValue("next_deadline");
            if (d1 > d0) return -1;
            else if (d1 == d0) return 0;
                else return 1;
        }
    }
    private LReady list_ready;
    private boolean todo = false;
    public void init() {
        list_ready = new LReady();
    }

    public void onActivate(EvtContext c) {
        Task T = (Task)c.getCible();
        T.setOwnFieldIntValue("next_deadline", this.Kernel.getTime()+T.getDeadline());
        list_ready.add(T);
        todo = true;
    }

    public void onUnBlock(EvtContext c){
        Task T = (Task)c.getSource();
        if (T.isBegin()) {
            T.setOwnFieldIntValue("next_deadline", this.Kernel.getTime()+T.getDeadline());
        }
        list_ready.add(T);
        todo = true;
    }

    public void onBlock(EvtContext c){
        list_ready.remove(c.getCible());
        todo = true;
    }

    public void onTerminated(EvtContext c){
        list_ready.remove(c.getCible());
        todo = true;
    }

    public void sched(){
        if (todo) {
            select();
            todo = false;
        }
    }

    public void select() {
        Collections.sort(list_ready,list_ready);

        ArrayList CPUS =this.Kernel.getTasksListeManager().getProcessors();
        int m = CPUS.size();

        for (int i=m; i<list_ready.size(); i++){
            Task T = (Task) list_ready.get(i);
            if (T.isIsrunning()) T.preempt();
        }

        int j = 0;
        for (int i=0; (i<m) && (i<list_ready.size()); i++){
            Task T = (Task) list_ready.get(i);
            if (!T.isIsrunning()) {
                Processor P = null;
                for (; j<m; j++){
                    P = (Processor) CPUS.get(j);
                    if (!P.isRunning()) {
                        j++;
                        break;
                    }
                }
                T.runningOn(P);
            }
        }
    }
}
```

6. How to add a new scheduler in STORM

We assume that the STORM program (`storm-x-x.jar`) has been downloaded beforehand. We now give the simple steps that have to be followed so as to insert a new scheduler in the STORM tool and test it:

1. When the new scheduler has been written as a Java program (for example `NewScheduler` class), it has to be saved in a file (for example `NewScheduler.java`) in the `storm-x-x.jar` directory.
2. The compilation of this new program needs to be achieved thanks to the following command in the `storm-x-x.jar` directory:
   ```
   javac -cp ./storm-x-x.jar  NewScheduler.java
   ```
   If errors are detected, the `NewScheduler.java` program must be modified until the compilation succeeds.
3. An xml file specifying the system to be simulated (for example `MyExample.xml`) must be available in the `storm-x-x.jar` directory. Do not forget that the `SCHEDULER` tag in this file must reference the name of the new scheduler (thus for the example, `classname="NewScheduler"`).
4. STORM is launched by using the command in the `storm-x-x.jar` directory:
   ```
   java -cp ./storm-x-x.jar  programme.programme
   ```
   (`programme.programme` is the Java STORM main program name)
5. From the STORM console, the simulation is achieved with the command:
   ```
   exec ./MyExample.xml
   ```
   All the STORM commands are given and explained in the available STORM user guide at: http://storm.rts-software.org/.