

SCHEDSIM

SCHEDuler SIMulator

(Antonio Barbalace, 2011)

This project provides a model intended to serve as a guideline to the students for the development of project assignments in the post-graduate course on Real-Time Systems (degree in Computer Engineering, University of Padova - Italy).

The aim of this project is to develop a simulator of different scheduling policies. The developed simulator is generic in the sense that it can be easily extended and modified. At present it handles periodic tasks, but not aperiodic, nor sporadic jobs.

The simulator mimics a single unit of computation (processor) with preemption enabled. The whole project follows the Object Oriented design rules and it is written in C++ (tested with GCC).

The current version of the project implements an event driven simulator and the following scheduling disciplines: EDF, LST, FIFO, LIFO, RM, DM. Preemption is always enabled and there is no pre-scheduling feasibility test (the implementation of such a test is straightforward and is left as an exercise). The trace of the scheduled tasks is formatted to be read by the Kiwi trace viewer written by Agustín Espinosa Minguet, project hosted by the University of Valencia (Spain).

This scheduler simulator can be used as an aid on a Real-Time Systems course or as a basic prototyping tool. The project is hosted on SourceForge.net (<http://schedsimula.sourceforge.net/>).

OTHER WORKs

Many similar event driven simulators can be found on internet:

- Real-Time system SIMulator (RTSIM, <http://rtsim.sssup.it>) developed at Retis Lab of the Scuola Superiore Sant'Anna (Italy);
- Cheddar (<http://beru.univ-brest.fr/~singhoff/cheddar/>) a free real time scheduling tool that supports AADL and it is prescribed by the OMG MARTE profile as a RMA checker. It is actively developed and maintained by the LISyC Team, University of Brest (France);
- SCHEDSIM (<http://schedsim.sourceforge.net>) a very basic scheduler simulator written in Java;
- LinSched (<https://lwn.net/Articles/409680/>) maintained by Google helps developers understanding what happens during the Linux scheduler activity.

The group of Jane W.S. Liu developed the famous PERTS (Prototyping Environment for Real-Time Systems) in the 90's; many articles related to such work can be found in the literature.

DEFINITIONS

A periodic task releases a job each period. A job is made up by different operations. Different jobs of the same task can be splitted in various operations. A job can be splitted in different operations if preempted. We do not account for resource sharing and contention.

We use the same conventional notational from the book of Jane W.S. Liu to identify periodic tasks (page 85). A periodic task T_i with phase ϕ_i , period p_i , execution time e_i and relative deadline D_i is represented by the 4-tuple (ϕ_i, p_i, e_i, D_i) .

Following again the book of Jane W.S. Liu an aperiodic or a sporadic task is a stream of aperiodic or sporadic jobs. The interarrival times between consecutive jobs in such tasks may vary widely. Sporadic jobs have hard real-time deadlines, while aperiodic jobs have soft deadlines or no deadlines at all.

DESIGN ISSUES

As stated before, the project supports scheduling algorithms on a single unit of computation; no issues concerning migration in a multiprocessor environment are considered.

In an Operating System the scheduler is triggered by the timer or by generic events. Such events can be the end, release, wait and yield operation of a task but also the return from an interrupt (not the timer interrupt). The scheduling activity is performed on returning from an interrupt because the interrupt service routine can have signaled/posted/unlocked a synchronization object (i.e. a semaphore). Synchronization objects are used to protect shared resources, the simulator does not handle resource control.

Whichever way an Operating System scheduler is triggered (events, system clock, or both) it has to make scheduling decisions and the operations to perform depend on the chosen triggering policy.

For instance, the RTAI kernel scheduler can be configured to work as time triggered (when you choose to use the *periodic timer*) or event driven (when you choose the *one shot timer*). In those configurations scheduling decisions are not only handled at different times (upon each timer interrupt, or on the occurrence of some event), but also decisions taken are different: an event usually triggers the execution of one task, whereas a timer interrupt can move to the ready queue many tasks depending on the time period of the system clock.

As there are two scheduling techniques, two simulation techniques exist: event driven and clock driven (discrete time).

In an event driven simulation the simulator maintains a global queue of events, it loops on such events but other global queues of events can exists.

In a clock driven simulation, first of all it is necessary to choose the time quanta, then the simulator loops on the time quanta until a maximum value. In a discrete time simulation, decisions are taken every time quanta; the time quanta is usually selected to divide each task's time-related quantity (phase, period, execution time, deadline). By the way such choice requires much more

computational work than the one required by an event driven simulation to run. Moreover a round robin scheduling policy (like any other time triggered scheduling policy) is much easier to implement adopting a clock driven simulation approach.

The project implements an event driven scheduler simulator that reacts only to two kind of events: job releases and job terminations (termination due to the end of the execution time). Reacting to those two events is enough to enable the simulator scheduling the following policies (following the categorization of Jane W.S. Liu):

- fixed priority
 - DM
 - RM
 - LETF
 - SETF
- dynamic priority (task level)
 - EDF
 - LRT (with pre and post elaboration)
 - FIFO
 - LIFO
- dynamic priority (job level)
 - LST non-strict
 - LST strict
 - RR

Without handling time events the round robin algorithm cannot be implemented. The LRT can not be implemented in a event driven simulator without a pre elaboration (that inverts all time-values and then schedules the task set using EDF) because actions do not take place on events but are referred to deadlines. As a result the current version of the project does not include the round robin (RR), nor the LRT policies. In addition LETF, SETF, and LST strict are not implemented.

SIMULATOR DESIGN

The simulator has been designed considering the interaction with a human user. The user should be allowed to visualize the scheduling trace in a friendly and intuitive way: the view of a scheduling trace using a GUI is mandatory. Among the many graphical trace viewers available on internet, we have chosen Kiwi (<http://rtportal.upv.es/apps/kiwi/>).

Using a 3rd party trace viewer permits us to implement the simulator as a command-line application with the following activity flow:

- check the command line arguments (input file name, output file name, scheduling policy and runlength of the schedule);
- read the tasks parameters from a file;
- determine how many decimal figures the task's parameters have;
- calculate the hyperperiod;

- perform the schedule;
- generate the Kiwi output;
- write the output.

All those steps are implemented in the source file *sched_sim.cpp*.

DATA MODEL and NOTES ON Standard Template Library (STL)

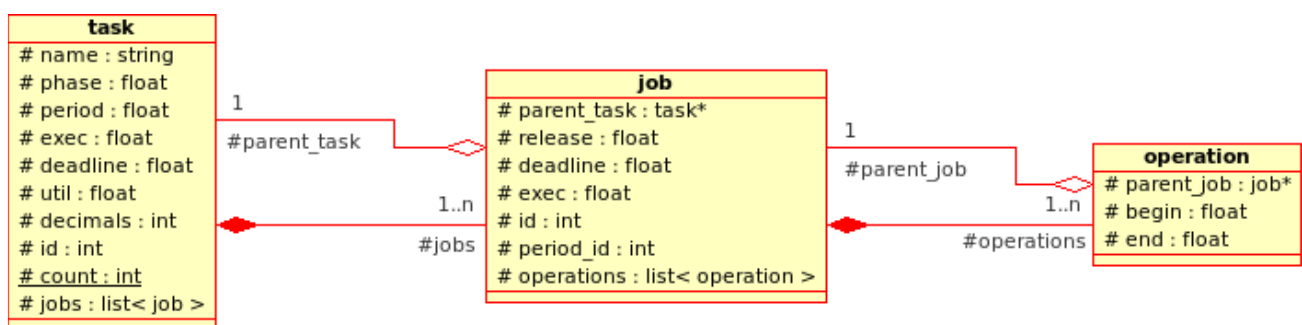
In this application we deal with different data structures of tasks, jobs and operations. Instead of developing our custom data structures from scratch we choose to use the Standard Template Library (STL) containers. STL implements the following base data structures:

- vector (dynamic growing array);
- deque (double ended queue, i.e. a vector that grows in both directions);
- list.

STL containers come along with sorting algorithms, in this sense we have not just containers but also mechanisms to use them. STL implements the sorting algorithms you can find in the literature; since this application is a simulator, i.e. not a real-time scheduler where performance counts, we do not make performance considerations in this document.

One important fact about STL is that containers handle objects by copy. Considering that in our application every object can be potentially on more than one data structure (vector, deque, list) at a time, managing objects by copy is not a good solution. Instead of queueing objects we choose to queue pointers to objects.

Application's objects are tasks, jobs and operations. Every task is responsible of all of its jobs and every job is responsible of all of its operations. This situation is depicted in the following UML diagram.



A task contains many jobs and the STL list named jobs in task contains job objects. When a task is deleted, the field jobs is deleted calling the STL list's destructor that in turn calls all the destructor of its jobs. The same happens with operations field in job class.

In a running simulator we expect only one instance of any task and all of its objects so the classes task and job are the only two classes holding STL containers **of objects**, all other classes hold STL containers **of pointers**.

We choose to use the list template to contain objects because the list moves list node's pointers instead of copying objects during sorting. As a pointer container we choose to use the STL double ended queue (deque) container even if a vector could be used instead. Both use indexed access to elements but in a deque it is also possible to choose enqueue and dequeue side (that varies in different scheduling algorithms).

EVENT DRIVEN ALGORITHM DESIGN and RELATED ISSUES

The main data structures required by the event driven scheduler simulator are:

- global events queue (`event_queue`, queue of all jobs in release time order);
- ready queue (`ready_queue`, ordered queue of jobs, the order depends on the scheduling policy);
- scheduled events queue (`term_queue`, queue of all terminated operations in happened time order).

Other variables tied to the scheduler are:

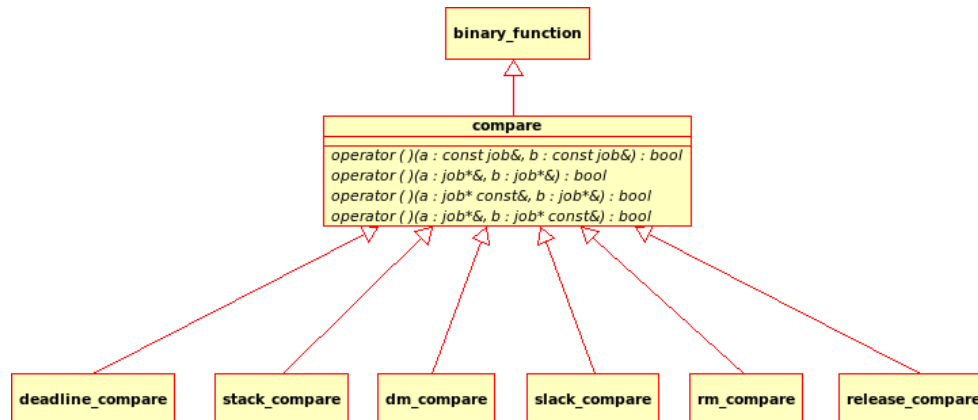
- a running pointer (`running`, pointer to the running job object);
- the current time (`current_time`).

The aforementioned variables are fields of the `scheduler` class. To define a scheduler you should select the scheduling policy. Such a policy is defined when a sorter is associated to the ready queue (in our implementation it is also required to associate a sorter to the global events queue, this feature was embedded to ease the development of the LRT algorithm).

STL comes with two sorter template functions (`sort` and `stable_sort`), templated on the compare function/object; in fact it is possible to pass as a comparator either a function or a `Compare` object with the `bool operator()` redefined. The declaration of the two templated functions follows; it is necessary to include the STL's *algorithm* header file to use them.

```
template <class RandomAccessIterator>
    void sort ( RandomAccessIterator first, RandomAccessIterator last );
template <class RandomAccessIterator, class Compare>
    void sort ( RandomAccessIterator first, RandomAccessIterator last,
               Compare comp );
template <class RandomAccessIterator>
    void stable_sort ( RandomAccessIterator first, RandomAccessIterator last );
template <class RandomAccessIterator, class Compare>
    void stable_sort ( RandomAccessIterator first, RandomAccessIterator last,
                     Compare comp );
```

Having a sorter that can be setup with different comparators is a starting point to develop a single scheduling algorithm that can be specialized to any scheduling policy simply by selecting the sorting comparator. In C++ this can be done via object inheritance, i.e. every comparator will be a subclass of a `compare` class that has to be passed as the last argument to the sorting function (see declarations above). The following figure shows the UML diagram of the discussed implementation.

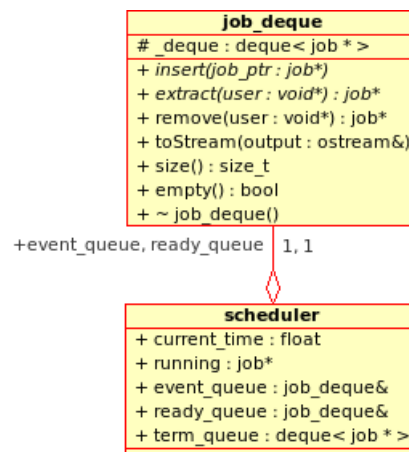


Unfortunately a template function in C++ has to know its exact template typename at compile time (before the linking stage); as a consequence on such template typename inheritance and virtual methods do not work.

Considering this fact and that we do not want to write our sorting algorithm from scratch, but still wish to use the one from STL, we move generalization a bit further. Since the sorting algorithm plus the comparator are used to keep a job queue in order, instead of generalizing on the comparator we generalize on the job queue.

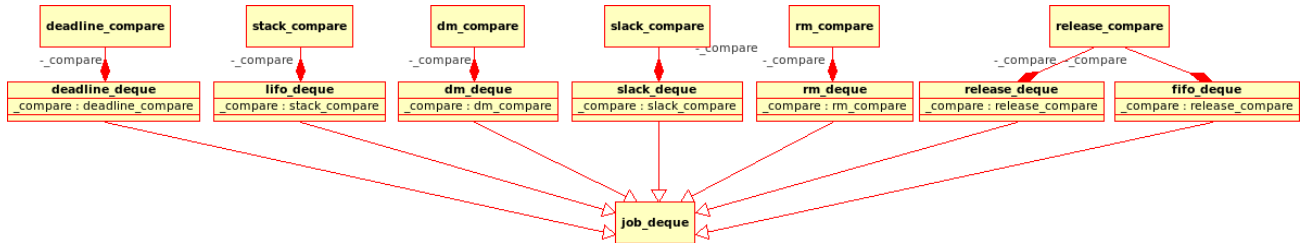
Rather than having a `scheduler` class initialized by a different `compare` object depending on the scheduling discipline, the developed `scheduler` class requires a `job_deque` object subclass in the constructor.

The UML diagram shows the class dependency:



The class `job_deque` is an extension of the STL's class `deque`, templated on the type `job*`. Such a class, similarly to the STL's `priority_queue` data structure, guarantees a sorted access to container elements. The access to such elements occurs with methods `job_deque::extract(...)` and `job_deque::remove(...)`. The sorting is done in any subclasses of `job_deque`, that in fact is an interface class (all methods are pure virtual). We choose to use the `stable_sort` STL's algorithm to keep in the same order items with the same "key". Such a choice

avoids random behaviors on the ready queue and avoids problems when the FIFO policy has to deal with jobs having the same release time.



The above UML diagram shows that for each scheduling discipline two classes must be provided: one subclass of `job_deque`, the job queue, and one subclass of `compare`, i.e. the comparator. Classes `deadline_deque`, `slack_deque`, `dm_deque`, `rm_deque`, `lifo_deque` and `fifo_deque` sort the ready queue. The class `release_deque` sorts the event queue. Class `fifo_deque` and `release_deque` use in fact the same `release_compare` comparator but for performance issues the `fifo_deque` class sorts the queue before extracting an elements but the `release_deque` class after inserting.

As an example we describe here the steps we followed to add the LIFO scheduling discipline to the simulator. First of all the class `stack_compare` was created as an header file `stack_compare.h`. The first few lines of the C++ source code follow.

```

...
struct stack_compare : public compare
{
    inline bool operator() (const job& a, const job& b)
    {
        return (a.getRelease() > b.getRelease());
    }
}
...

```

The first line declares the `stack_compare` class as a subclass of the `compare` class. The last line implements the LIFO comparator. The remain of the file is similar.

Then files `lifo_deque.h`, `lifo_deque.cpp` were added to the project as an implementation of the `lifo_deque` class. The following piece of code is what we have customized in the header file: renamed the class as `lifo_deque` and changed the `_compare` type to `stack_compare`.

```

...
class lifo_deque : public job_deque
{
    stack_compare _compare;
}
...

```

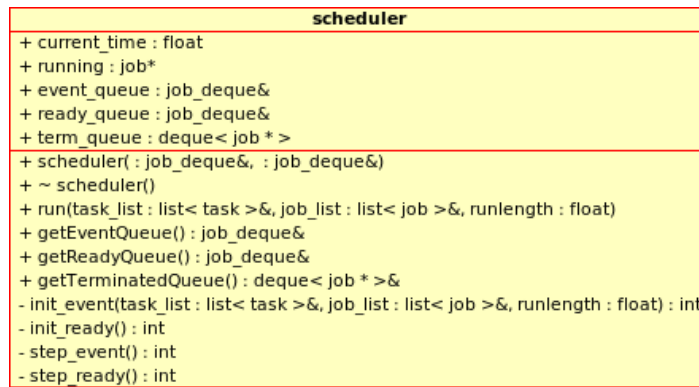
In the `*.cpp` file we have just changed some include directive but no other work is required to add a new scheduling policy.

ALGORITHMIC STEPS

The whole simulation algorithm is divided in the following steps:

- jobs generation; for each periodic task we generate jobs until runlength. Runlength can be user defined, or two hyperperiods (this procedure is implemented in `scheduler::init_events(...)`);
- all ready jobs at the initial instant are moved to the ready queue and the most prioritary one is selected to be run (`scheduler::init_ready(...)`);
- event driven scheduling loop (events are: job releases and job terminations) triggered by events on the events queue (`scheduler::step_event(...)`);
- flushing of the remaining jobs on the ready queue (`scheduler::step_ready(...)`).

All of this computational stages are reflected by methods in the scheduler class as the following UML shows.



THE ALGORITHM

The event driven algorithm loop is coded in the method

`scheduler::step_event(...)` .

On every loop it checks for a new task release or a task termination. A task release can preempt the current running task or simply move a task from the event to the ready queue. A task termination moves the current running task to the termination queue and picks up a task from the ready or event queue.

The algorithm in pseudo code follows.

ROUTINE `step_event`

INPUT: *event_queue*, *ready_queue*, *term_queue*, *running*

VARs: *finish_time*, *event_time*, *current_time*, *job*

WHILE *event_queue* NOT empty DO

job = extract from *event_queue*

finish_time = *current_time* + slack time of *running*

event_time = release time of *job*

 IF (*finish_time* >= *event_time*) DO

 insert *job* in *ready_queue* and remove it from *event_queue*

 IF (*finish_time* != *event_time*) DO

 insert *running* in *ready_queue*


```

ENDIF
job = extract and remove the most prioritary from the ready_queue
IF ( job NOT running ) DO
    add an operation to running and decrement its execution time**
    IF ( finish_time == event_time )
        add running to term_queue
    ENDIF
    current_time = event_time
    running = job
ENDIF
ELSE
    add an operation to running and decrement its execution time**
    add running to term_queue
    current_time = finish_time
    IF ( ready_queue NOT empty ) DO
        running = extract from ready_queue
    ELSE
        remove job from event_queue
        running = job
        current_time = event_time
    ENDIF
ENDIF
ENDWHILE

```

When all jobs from the event queue are processed we have to flush the ready queue to be sure there are no more jobs waiting to terminate execution. This step is done by `scheduler::step_ready(...)`.

**operation will not be added if its timespan is zero.

KIWI OUTPUT

This version of the project output the execution trace in Kiwi format only. During the generation of the trace file every job end time is checked against its deadline and a message is showed on the shell in case of a miss. In the Kiwi trace a deadline miss is marked with a yellow down arrow on the deadline line bar.

Kiwi is written in the **tcl/tk** scripting language and the whole source is coded in one file only. The package comes with a documentation *directory kiwi-1.0.1/docs/* where you can find the *user-guide.html* file and a directory containing some rich examples *kiwi-1.0.1/examples/*. It is really important to either a look at the examples and at the user guide before starting using this simple tool.

Kiwi was developed to trace feasible schedules of tasks where two consecutive release-deadline timespans never overlap. In our implementation, where we have chosen to mark and warn about deadline miss but not to stop the job that exceed its deadline; as a result consecutive release-deadline timespans can overlap; Kiwi is not able to correctly visualize them (so, take care when analyzing traces).

Such a trace visualizer requires that events appear in time-stamp order in the source file and events from the same job must be as close as possible to one another. In the scheduler simulator we have first generated all events in time-stamp order for each task separately and then reordered all events using a stable sorter.

INCLUDED EXAMPLES

As written in the previous pages the simulator requires a list of periodic tasks as input file. Such input file must contain a task description per line and the description must adhere to the following format:

```
<task_name> <phase> <period> <execution_time> <relative_deadline>\n
```

that matches the following `scanf` format string:

```
%s %f %f %f %f\n
```

Except for the first character string parameter, all others are float numbers.

The following task sets are included in the software package. The task sets are taken from the book of Jane W.S. Liu. (*tasks6.2a* refers to the task set from *Figure 6.2a* of the cited book, and so on).

- tasks6.2a
- tasks6.2b
- tasks6.3
- tasks6.4
- tasks6.5a
- tasks6.5b
- tasks6.5c
- tasks6.11
- tasks_p121a (task set defined in lines 19 and 20 of page 121)
- tasks_p121b (task set defined in the last line of page 121)

COMPILING THE CODE

Download the code from sourceforge by anonymous login using for example the CVS service (you can optionally download it via SVN):

```
$ cvs -  
d:pserver:anonymous@schedsimula.cvs.sourceforge.net:/cvsroot/schedsim  
ula login  
$ cvs -z3 -  
d:pserver:anonymous@schedsimula.cvs.sourceforge.net:/cvsroot/schedsim  
ula co -P .
```

Move to *schedsimula* directory and then simply type *make* at the shell and you are done with compilation.

Before running the application you have to download and install the Kiwi trace viewer in your computer to be able to view the trace.

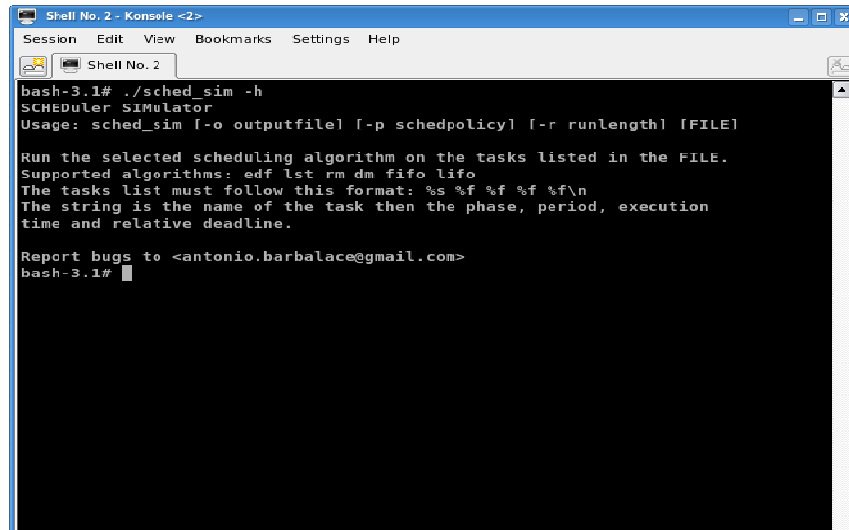
RUNNING THE CODE

After running *make* with success you will find in the current directory an

executable called *sched_sim*: this is the scheduler simulator.

The application can run also without any command line arguments using the default built-in configuration that searches for an input file named *task.list*, outputs the trace in *sched.ktr* and performs an EDF schedule.

To have a complete list of the currently supported scheduling algorithms by the application run the simulator with the *-h* option, below there is a screen shot of the output.



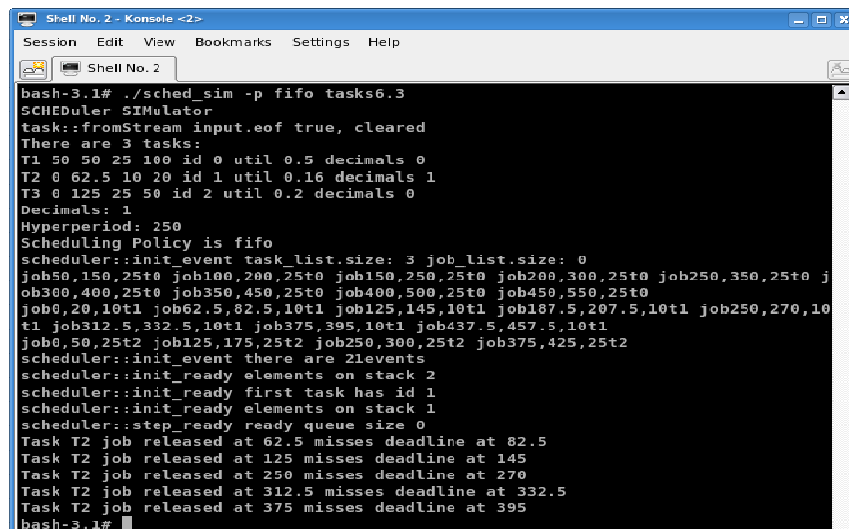
```

Shell No. 2 - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell No. 2
bash-3.1# ./sched_sim -h
SCHEDULER SIMULATOR
Usage: sched_sim [-o outputfile] [-p schedpolicy] [-r runlength] [FILE]

Run the selected scheduling algorithm on the tasks listed in the FILE.
Supported algorithms: edf lst rm dm fifo lifo
The tasks list must follow this format: %s %f %f %f %f\n
The string is the name of the task then the phase, period, execution
time and relative deadline.

Report bugs to <antonio.barbalace@gmail.com>
bash-3.1#
  
```

In the following we go through the few steps required to create and to view a schedule of a set of tasks. We have chosen to schedule by the FIFO algorithm the task set of Figure 6.3 of the aforementioned book. Look at the picture below that shows the command line argument.

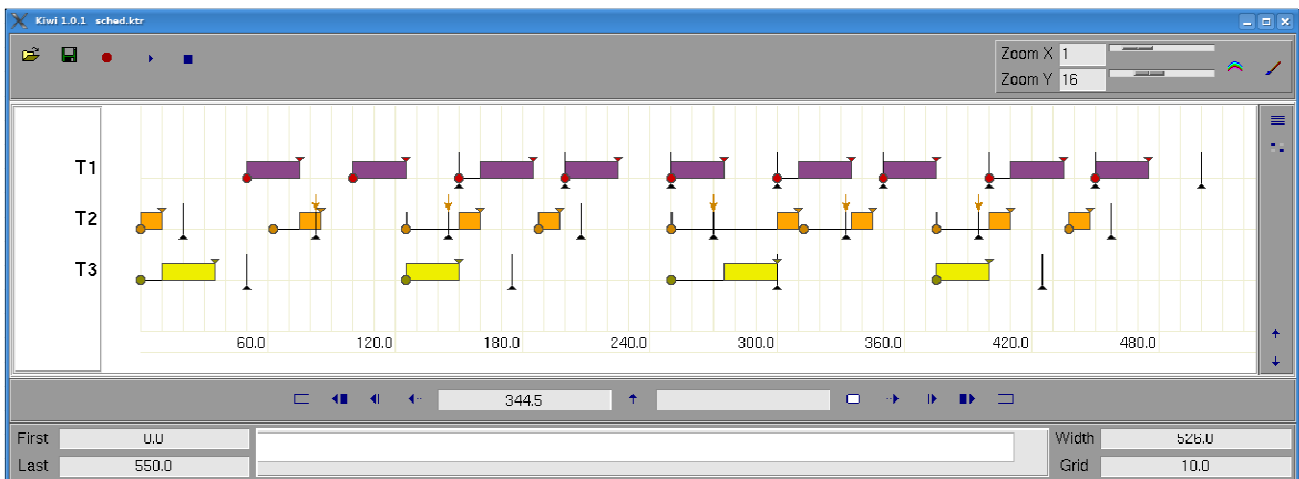


```

Shell No. 2 - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell No. 2
bash-3.1# ./sched_sim -p fifo tasks6.3
SCHEDULER SIMULATOR
task::fromStream input.eof true, cleared
There are 3 tasks:
T1 50 50 25 100 id 0 util 0.5 decimals 0
T2 0 62.5 10 20 id 1 util 0.16 decimals 1
T3 0 125 25 50 id 2 util 0.2 decimals 0
Decimals: 1
Hyperperiod: 250
Scheduling Policy is fifo
scheduler::init_event task_list.size: 3 job_list.size: 0
job50,150,25t0 job100,200,25t0 job150,250,25t0 job200,300,25t0 job250,350,25t0 j
ob300,400,25t0 job350,450,25t0 job400,500,25t0 job450,550,25t0
job0,20,10t1 job62.5,82.5,10t1 job125,145,10t1 job187.5,207.5,10t1 job250,270,10
t1 job312.5,332.5,10t1 job375,395,10t1 job437.5,457.5,10t1
job0,50,25t2 job125,175,25t2 job250,300,25t2 job375,425,25t2
scheduler::init_event there are 21events
scheduler::init_ready elements on stack 2
scheduler::init_ready first task has id 1
scheduler::init_ready elements on stack 1
scheduler::step_ready ready queue size 0
Task T2 job released at 62.5 misses deadline at 82.5
Task T2 job released at 125 misses deadline at 145
Task T2 job released at 250 misses deadline at 270
Task T2 job released at 312.5 misses deadline at 332.5
Task T2 job released at 375 misses deadline at 395
bash-3.1#
  
```

The first five lines give a feedback on the read input file, then the selected number of significant decimal digits and the hyperperiod are specified. The

selected scheduling policy is reported and all the jobs in the timespan between zero and the runlength are generated for each task. The five lines starting with `scheduler::` give you a feedback that the five steps of the algorithm (discussed in text above) are done. All other lines on the prompt log any deadline miss. The whole trace can be only viewed using Kiwi, the screen shot below in fact shows the obtained trace. To visualize a trace you have first to open the file (using the yellow folder icon) and then click on the blu play icon, use the *Zoom X* slider to adjust the view (click on the brush icon to repaint the view).



A very interesting feature of Kiwi is the one shown in the next picture: clicking on the playlist icon (the one on top of the right middle bar) a list of all events will appear. You can then navigate and deep analyze what happened in the scheduling activity.

