FIRST
Flexible Integrated Real-Time Systems

IST-2001 34140

Hierarchical scheduling in the S.Ha.R.K. Kernel

Deliverable D-OS2-v1

Responsible:
Paolo Gai
ReTiS Lab
Scuola Superiore S. Anna, Pisa
pj@sssup.it

29th August 2002

Contents

This document has been written using LaTeX and LyX.

1 Introduction

This document describes the implementation of a hierarchical scheduling architecture on the S.Ha.R.K. Kernel.

The proposed scheduling architecture allows multiple applications to be scheduled on the same system, each application having its separate scheduling algorithm.

The system is capable to mix the scheduling of the various applications using a reservation approach based on the CBS algorithm [1]. Basically, each application's scheduler is linked to a CBS server that schedules its tasks guaranteeing that the application will receive at least the bandwidth it has requested to the Kernel.

2 Scheduling architecture

The system is composed by a set of *CBS servers* and by a set of *independent* applications.

Each CBS server β_i contains a set of tasks and is assigned two parameters: a period P_i and a budget Q_i. The algorithm assigns a proper deadline to each β_i to guarantee that each server will receive at least Q_i units every P_i units of time. Each server has an internal FIFO queue to handle its tasks.

Each application is composed by a set of tasks T_j, and by a *local* scheduling algorithm α_j. At each instant the α_j algorithm is able to select a task to schedule from T_j.

The architecture proposed in this document is a multi level scheduling solution:

- At the application level we consider the local scheduling algorithm α_j of each application.
- Each local scheduling algorithm inserts the selected task into a CBS server β_i. That server assigns a proper deadline to the first task in its queue, and then inserts it into the global queue.
- At the upper level, the global queue is scheduled using an EDF [2] scheduler.

Figure 1 describes a simple example composed by three applications. Application 1 and 2 share the same CBS server with parameters (Q_1, P_1), whereas Application 3 uses a dedicated CBS server with parameters (Q_2, P_2). The three application's local schedulers insert their first task into the CBS server linked to them. Each CBS server inserts its first task into the global EDF queue. The global EDF queue contains also a system task t_s (i.e., a device driver) scheduled together with the other CBS servers.

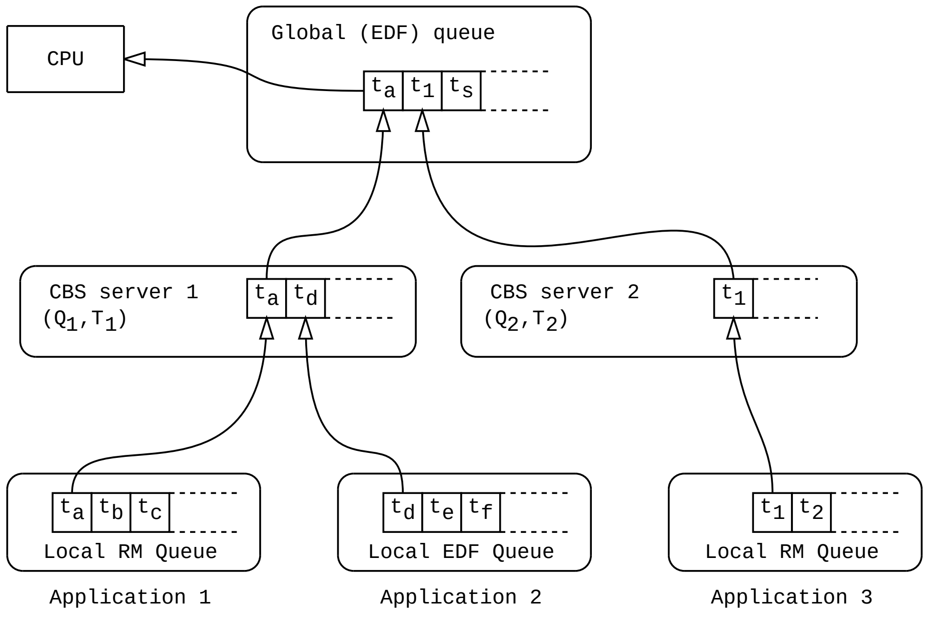

Figure 1: An example.

3 Implementation details

The proposed implementation is composed by three S.Ha.R.K. scheduling modules:

`CBSSTAR` This module implements the CBS servers that are used to schedule the tasks inserted by the local schedulers.

`EDFSTAR` This module implements a local EDF scheduler that handles hard periodic tasks inserting them into a CBS server.

`RMSTAR` This module implement a local RM scheduler. This module is similar to the `EDFSTAR` module.

Moreover, an abstract data type called `IQUEUE` is proposed to efficiently handle task queues into the `EDFSTAR` and the `RMSTAR` modules.

The following subsections contains a small description of the three scheduling modules and of the `IQUEUE` data structure.

3.1 The `CBSSTAR` Scheduling Module

This module is coded into the files `cbsstar.h` and `cbsstar.c` and it implements a set of CBS servers. Each CBS server β_i contains a FIFO queue that is used to enqueue the tasks inserted in it, and two parameters (Q_i, P_i).

The module can be registered at system startup calling the following function:

```
LEVEL CBSSTAR_register_level(int n, LEVEL master);
```

where:

`n` is the maximum number of CBS servers that the module can handle;

`master` is the level number of the EDF module that implements the global EDF queue[1].

The function returns the level number at which the module has been registered.

Once the module has been registered, the user can define a CBS server calling the following function:

```
int CBSSTAR_setbudget(LEVEL l, TIME Q, TIME T);
```

where:

`l` is the level number at which the `CBSSTAR` module has been previously registered;

`Q, P` are the parameters that characterize the CBS server.

The function returns an integer that can assume these values:

[1]The EDF Module is one of the standard scheduling modules provided with the S.Ha.R.K. Kernel.

`0...n-1` This is the identifier of the CBS server. That identifier has to be used every time the (just created) CBS server has to be referenced. Once created, a CBS server can not be deleted. All the other values are error conditions;

`-1` The maximum number of CBS servers has been reached;

`-2` $\sum_i \frac{Q_i}{P_i} + \frac{Q}{P} > 1$, that is the bandwidth used by the currently allocated CBS servers plus the new server is greater than 1;

`-3` The `l` parameter is wrong.

A module m can insert its tasks into a CBS server as *guest* tasks. To insert a task `p` into a CBS server with index `id`, the module m have to call the `guest_create` guest call of the `CBSSTAR` module containing the CBS server `id` passing a `BUDGET_TASK_MODEL`. The `BUDGET_TASK_MODEL` task model, that requires as parameter only the `id`, is provided into `cbsstar.h`. Once inserted, `p` will be scheduled by the `CBSSTAR` Module.

In particular, the `CBSSTAR` module inserts the first task of each server FIFO queue into its master module using a `JOB_TASK_MODEL`. The `JOB_TASK_MODEL` is initialized with the deadline of the CBS server, which will be postponed as long as the task executes. A deadline postponement is implemented removing the task from the master module and reinserting it using the modified deadline.

The module m have to call also the following guest calls:

`guest_end` This guest call is called when the module m needs to remove `p` from the CBS server. This happens for example when `p` terminates, when it is preempted or when it is suspended for some reason (i.e., for synchronization).

`guest_dispatch` This guest call must be called by the module m when `p` is dispatched (that is, when the module m `task_dispatch(p)` is called).

`guest_epilogue` This guest call must be called by the module m when `p` is preempted (that is, when the module m `task_epilogue(p)` is called).

3.2 EDFSTAR Scheduling Module

This module is coded into the files `edfstar.h` and `edfstar.c` and it implements a local EDF scheduler. The module has its own ready queue that contains all the application's ready task.

The module can be registered at system startup calling the following function:

```
LEVEL EDFSTAR_register_level(int budget, int master);
```

where:

`budget` is the identifier of the CBS server where the module inserts its guest tasks;

`master` is the level number of the `CBSSTAR` module that implements the CBS server.

The function returns the level number at which the module has been registered.

Once the module has been registered, the user can create a periodic task using the following task model (with its minimal initialization values):

```
HARD_TASK_MODEL m;
hard_task_default_model(m);
hard_task_def_wcet(m, MYWCET);
hard_task_def_level(m, ELEVEL);
hard_task_def_periodic(m);
hard_task_def_mit(m1,MYMIT);
```

where `MYWCET` is the worst case execution time (WCET) of the task, `ELEVEL` is the value returned by `EDFSTAR_register_level`, and `MYMIT` is the period of the task

The module does not perform any kind of acceptance test on the tasks, and the tasks will be scheduled also if their WCET is greater than the declared WCET.

The module also provides the following functions:

```
int EDFSTAR_get_dline_miss(PID p);
int EDFSTAR_get_wcet_miss(PID p);
int EDFSTAR_get_nact(PID p);
int EDFSTAR_reset_dline_miss(PID p);
int EDFSTAR_reset_wcet_miss(PID p);
```

These functions receive as parameter the identifier of a task handled by an `EDFSTAR` module (otherwise, they return `-1`). In case of success, they respectively returns the number of deadline misses, the number of WCET overflows, or the number of pending activations. The last two functions simply reset the internal deadline miss or wcet overflow counters.

The internal implementation uses the guest calls provided by the `CBSSTAR` module to insert the earliest deadline ready task into the CBS server. Moreover, the ready queue is implemented using an `IQUEUE` data type, allowing the insertion of a task into the `CBSSTAR` module without removing it from the ready queue.

3.3 `RMSTAR` Scheduling Module

`RMSTAR` is similar to the `EDFSTAR` scheduling module, except that the ready queue is ordered by period instead of ordering it by deadline.

3.4 IQUEUE data structure

The S.Ha.R.K. kernel provides a set of functions that help the implementation of task queues into the scheduling modules. These functions use two data types called `QUEUE` and `QQUEUE`, and four fields of the task descriptor used to store the `prev`/`next` pointers and the priorities (an integer and a `struct timespec`) for each task. Since these functions share the same fields in the task descriptors, and it is not possible for a task to stay at the same time in more than one task queue. This fact is not a big issue in the implementation of most scheduling algorithms, but may be a disadvantage when hierarchical scheduling have to be supported.

The `IQUEUE` abstract data type tries to solve that problem. Basically, an `IQUEUE` has an "I"nternal prev/next structure, that may be shared with other `IQUEUE`s. Of course, the user MUST guarantee that the same task will not be inserted in two `IQUEUE`s that share the same `prev`/`next` pointers.

The `IQUEUE` data structure has been used in the implementation of the `EDFSTAR` and of the `RMSTAR` scheduling modules. In these modules, the local scheduling algorithm maintains a task queue where the first task in the queue is the task that is also inserted in the master module. The fact that the highest priority task remains into the ready queue makes the implementation simpler and more easy to understand.

We are currently investigating the possibility of extending the use of `IQUEUE`s to the whole S.Ha.R.K. Kernel. The `IQUEUE` source code is contained into the `iqueue.h` and `iqueue.c` source files.

4 Tests

This Section briefly presents the test cases used during the debugging of the EDFSTAR, CBSSTAR and RMSTAR scheduling modules.

4.1 test1.c

This test initializes a single CBS server and a single EDFSTAR module. The test is composed by 4 hard tasks with different periods scheduled by the EDFSTAR module, and another hard task scheduled by the global EDF scheduler. The other tasks in the system are the dummy task, the main task and the keyboard driver.

The demo shows that the EDFSTAR module enqueues its task following the EDF rules, but that the scheduling parameters (the deadline) passed in the global EDF queue are those of the CBS server.

If the file edfstar.c is compiled with the symbols EDFSTAR_DEBUG, and edfstar_printf3 active, a couple (dline, curtime) is printed on the console (in ms). If cbsstar.c is compiled with cbsstar_printf3 active, the budget replenishments are also printed.

4.2 test2.c

This test involves two EDFSTAR modules linked respectively to two CBS server. The test is composed by 2 hard tasks with different periods linked to each EDFSTAR module. The other tasks in the system are the dummy task, the main task and the keyboard driver.

The demo is used to show that in case of periodic tasks with a transient overload the two CBS servers behaves correctly sharing the total bandwidth available.

4.3 test3.c

This test involves two EDFSTAR modules linked respectively to two CBS server. The test is composed by a never ending (only one instance) hard task linked to each EDFSTAR module.

The other tasks in the system are the dummy task, the main task and the keyboard driver.

The demo shows that in case of full, stable processor utilization the two CBS servers behaves correctly sharing the total bandwidth available.

4.4 test4.c

This test involves two EDFSTAR modules linked to a CBS server, plus another EDFSTAR module linked to another CBS server. The test is composed by a periodic “long” hard task linked to each EDFSTAR module.

The other tasks in the system are the dummy task, the main task and the keyboard driver.

The demo shows that in case of full, stable processor utilization the two CBS servers behaves correctly sharing the total bandwidth available, also if one CBS server has more than one `EDFSTAR` modules linked to it.

4.5 `test5.c`

This test involves two `EDFSTAR` modules linked to a CBS server, plus another `EDFSTAR` module linked to another CBS server. The test is composed by a periodic "long" hard task linked to each `EDFSTAR` module, plus one soft task linked to one of the `EDFSTAR` modules.

The other tasks in the system are the dummy task, the main task and the keyboard driver.

The demo tests the use of a `JOB_TASK_MODEL` task model with the `EDFSTAR` module.

4.6 `test6.c`

This test is similar to `test1.c`, except that it uses a `RMSTAR` scheduling module instead of an `EDFSTAR` scheduling module.

4.7 `testiq.c`

This test tries to make a simple comparison between the execution times of the queue insertion functions for the `IQUEUE` and for the `QUEUE` data structure.

5 Availability

All the source code of this deliverable will be also available on the S.Ha.R.K. web page (`http://shark.sssup.it`) starting from September 15th, 2002.

References

[1] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real Time Systems Symposium*, Madrid, Spain, December 1998.

[2] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.