IST-2001 34140

Operating System Primitives – S.Ha.R.K. OS

Deliverable D-OS.2v3

Responsible: SSSA

Giuseppe Lipari, Michael Trimarchi,
Giacomo Guidi, Paolo Gai
Tomas Lenvall, Radu Dobrin

24th March 2005

Contents

1 Introduction **2**

2 The FSF interface library **2**

3 Implementation of FSF on S.Ha.R.K. **3**
3.1 Basic scheduling structure 3
3.2 Core module implementation 4
3.3 Spare Capacity module implementation 7
3.4 Shared Objects module implementation 7
3.5 Dynamic Reclaimation module implementation 9
3.6 Hierarchical Scheduling module implementation 10

A Multimedia example application **14**
A.1 Architecture Details 14
A.1.1 MPEG Player 14
A.1.2 S.Ha.R.K Version 15
A.1.3 Execution Time Estimation 16
A.1.4 S.Ha.R.K. Version with FSF 16
A.1.5 Running the player application 18
A.2 Summary 18

1 Introduction

In this document we describe the final implementation of the *First Scheduling Framework* (FSF) in the S.Ha.R.K. OS. A complete description of the framework, including the API provided to the user is reported in deliverable D-SI.1v3 and will only be summarised here.

Also, in Appendix A we report a complete example of a MPEG player developed in S.Ha.R.K. using the FSF framework.

2 The FSF interface library

The application programming interface (API) to the scheduling framework of the FIRST project consists of a software library, the FSF (FIRST Scheduling Framework) library. The library provides of a set of include files that contain the data definitions and the function prototypes to the service contract. A complete description of the FSF API is done in deliverable D-SI.1v3. In this section we briefly describe the modules that compose the FSF library, specifying which ones have been implemented in S.Ha.R.K.. The structure of the implementation is described in the next sections.

The FSF library provides many different and complex services, from simple budget control to synchronization primitives, hierarchical scheduling, reclaimation, shared objects and distribution. To simplify the structure of the library and its implementation, and to make the provided services more accessible to the final user, the FSF library has been divided into a set of modules, each one providing a specific set of services.

The *Core module* is essential for the framework because it provides the basic concept of service contract, which the entire library is built upon. The service contract is the mechanism that the application uses to dynamically specify its own set of complex and flexible execution requirements. A contract is specified through an opaque structure of type fsf_contract_parameters_t , whose parameters can be set through some functions. The contract can then be negotiated with fsf_negotiate_contract () or similar functions. If the negotiation is succesful, a *server* is created for a specified thread. The core module also include functions to obtain information from the scheduler or to synchronize the thread with the server.

The *Spare Capacity module* is optional. It is useful if we want to distribute extra capacity available in the system to the needing applications. This module adds parameters to the contract and functions to specify these additional parameters. Then the negotiation algorithm takes into account the extra capacity in assigning the budget to the servers.

The *Shared Objects module* is optional. It is used when two applications, using two different contracts, share a common data structure in memory with mutual exclusion semaphore. The module takes into account the extra budget that may be needed if the server normal budget is exhausted while the task is in a critical section. For this reason, it is necessary to specify the lenght of the critical sections in the contract specification. Also, we defined a new object type, fsf_shared_obj_id_t to identify shared objects, to specify the lenght of the associated critical sections, and to create a specific pthread_mutex_t variable for it.

The *Dynamic Reclaimation module* is optional. It adds the possibility to dynamically reclaim extra capacity available in the system due to non active servers or to threads that do not consume all the server budget. This extra capacity is distributed to the needing applications in a “best effort” way, in the sense that it is not possible to control how much extra budget an application will receive. This module has not specific function because it is not possible to set any parameter.

The *Hierarchical Scheduling module* is optional. It allows many threads belonging to one application to share the same server with a local scheduling algorithm. The corresponding API allows to specify the local scheduler and its parameters, and allows threads to be added to servers with their own scheduling parameters.

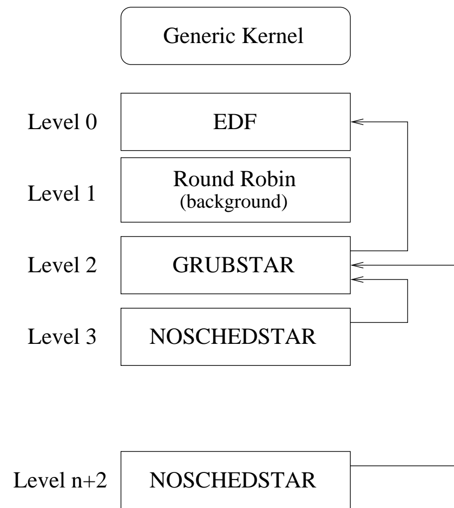

Figure 1: Organisation of the modules in S.Ha.R.K..

All the modules described so far have been implemented in the S.Ha.R.K. OS, and their implementation will be discussed in the following sections. S.Ha.R.K. does not implement the *Distributed module* and the *Distributed Spare Capacity module*.

3 Implementation of FSF on S.Ha.R.K.

In this section, we describe how the FSF library has been implemented in S.Ha.R.K..

3.1 Basic scheduling structure

As global scheduling algorithm, we selected the EDF togheter with a resource reservation algorithm, the GRUB algorithm [5]. This algorithm is very similar to the CBS algorithm of Abeni and Buttazzo [1], and in addition it automatically performs the dynamic reclaimation of the bandwidth. In S.Ha.R.K. it is possible to configure the total amount of available bandwidth for the framework. In most of the experiments, this bandwidth has been limited to 80% of the total system bandwidth. However, it is possible to change this fraction.

S.Ha.R.K. permits to easily modify the scheduler and to mix different scheduling policies. The basic mechanisms to implement a new scheduling policy is the *scheduling module*. A scheduling module resemble an object in a object oriented language: it has internal data structures, a set of "private" functions and a set of functions that implement the interface with the S.Ha.R.K. generic scheduling mechanism.

Therefore, to implement the basic structure of the FSF in S.Ha.R.K. we implemented a set of scheduling modules. Particular attention has been devoted to the hierarchical scheduling structure designed in the FSF.

In Figure 1 we show the basic structure of the S.Ha.R.K. scheduling modules used in FIRST. The *Generic Kernel* performs generic operations like the dispatching and suspension of a task. It also implements the interface for all system calls. The actual scheduling is done in the modules that are organized in levels. Modules in lower levels have higher priority.

Tasks are assigned to scheduling modules. When a system call is invoked by a task, the Generic Kernel identifies which module the task belongs to, and invokes the appropriate operation on that module.

In the structure we designed for the FSF, the module in the lower level is an EDF scheduler. Only if the EDF has no task to schedule, the module in level 1 (a simple Round Robin scheduler) is asked for something to be executed. This module contains the dummy task and the main() function. Thus the main and the dummy run in background.

Modules can "insert" tasks in other modules. This is the mechanism used to implement the server algorithm. Module GRUBSTAR handles the descriptors of all servers in the system, and the corresponding tasks. In Figure 1, the arrow from the GRUBSTAR module to the EDF module means that the GRUBSTAR inserts one task per each server in the EDF module using the deadline of the server.

Finally, each task is assigned a different module. In Figure 1we present the structure when no hierarchical server is involved: therefore, each task is assigned different NOSCHEDSTAR module (the name comes from the fact that this module does nothing becuase it has no specific local scheduler and can hanlde only one task).

Now we describe the actual sequence of function calls that is performed when a task becomes active. The sequence of messages is shown in Figure 2 as a UML sequence diagram.

When a task corresponding to a server becomes active, it invokes the activate () function of the Generic Kernel. This, in turns, calls the insert function of the NOSCHEDSTAR module, which in turns simply redirects it to the insert function of GRUBSTAR module. The module computes the budget and the deadline for the task, and "inserts" the task in the EDF module. If the task is the earliest deadline task, this insertion triggers a "re-schedule" in the Generic Kernel, which invokes a dispatch on the involved modules. As a consequence, module GRUBSTAR activates a timer to expire at the budget expiration of the task. The situation described above is depicted in Figure 2.

Similar situations happen for other scheduling decisions, like preemption or task suspension. For brevity, we do not report here the complete description of the S.Ha.R.K.ś internal scheduling mechanism for FSF. Please refer to the reference manuals of S.Ha.R.K. available for download on the S.Ha.R.K. web site (http://shark.sssup.it).

3.2 Core module implementation

Given the structure briefly described in the previous section, the implementation of the Core module is almost straightforward. The core module supports a negotiation mechanism for admission control of newly created servers. Since the creation of a new server may take some time (due both to the complexity of the admission control, and of the particular algorithm for distributing the spare capacity implemented in the spare capacity module) we decided to implement the negotiation mechanism as a separate thread, called *service thread*. This thread is handled by a dedicated server which is activated at system initialization. It is possible to change the pamaters of this servers with function

```
int fsf_set_service_thread_data (
    const struct timespec * budget,
    const struct timespec * period,
    bool * accepted )
```

The service thread communicates with the rest of the system with a *client/server* model, i.e. through message passing. The structure is shown in Figure 3. In particular, we created a message queue (by using the PORT mechanisms of S.Ha.R.K.) in which the clients insert the negotiation requests. The clients can then synchronize with the results of the negotiation by using private message queues. This mechanism is

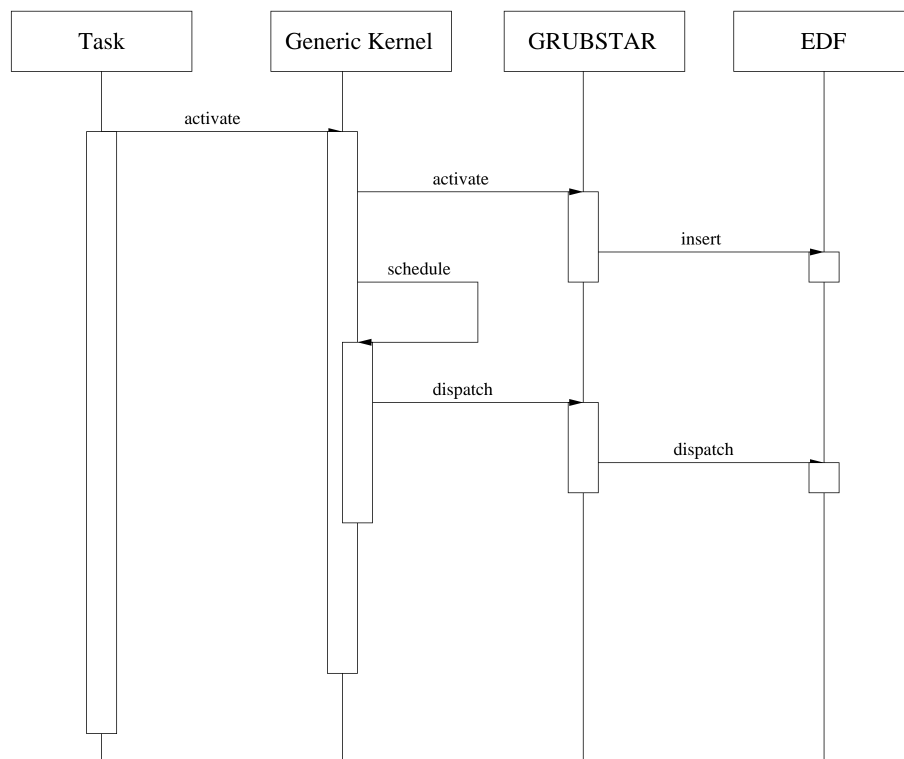

Figure 2: Sequence of messages exchanged between the S.Ha.R.K.ś scheduling modules when a task is activated.

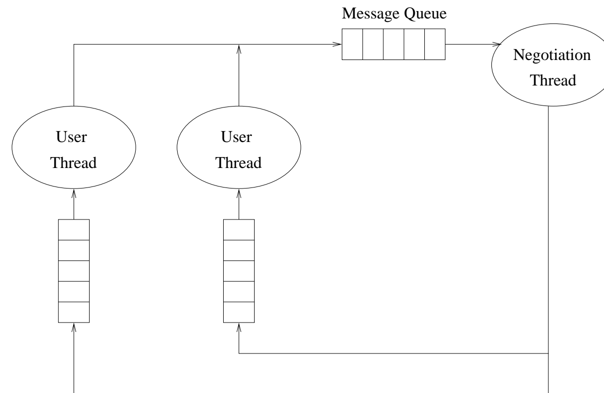

Figure 3: Client-server structure of the negotiation.

completely hidden to the FSF programmer, through the various negotiation functions available in the API (see the reference manual in the appendix of D-SI1.v3).

Another useful mechanism provided by the core module is the set of functions to synchronize the threads and the servers. The following functions:

```
int fsf_schedule_timed_job(const struct timespec *abs_time,
        struct timespec *next_budget,
        struct timespec *next_period,
        bool *was_deadline_missed,
        bool *was_budget_overran
    );

int fsf_schedule_triggered_job(fsf_synch_obj_handle_t synch_handle,
        struct timespec *next_budget,
        struct timespec *next_period,
        bool *was_deadline_missed,
        bool *was_budget_overran
    );

int fsf_timed_schedule_triggered_job(fsf_synch_obj_handle_t synch_handle,
        const struct timespec *abs_timeout,
        bool *timed_out,
        struct timespec *next_budget,
        struct timespec *next_period,
        bool *was_deadline_missed,
        bool *was_budget_overran
    );
```

interact with the GRUBSTAR module to get the needed information, like the assigned budget and periods.

The synchronisation object is mapped onto a S.Ha.R.K. semaphore. When a sporadic task finish its execution it can block on the semaphore with the fsf_schedule_triggered_job () primitive. When the corresponding event is triggered (with the fsf_signal_synch_obj ()), the semaphore is signaled and one of the

sporadic tasks blocked on the semaphore is unblocked.

The fsf_schedule_timed_job () allows the implementation of bounded workload threads (for example periodic tasks) and is mapped on the task_endcycle () of the S.Ha.R.K. specific API.

3.3 Spare Capacity module implementation

The spare capacity module modifies the standard negotiation algorithm adding the possibility of re-distributing the extra capacity in the system between the needing servers. For this reason, additional parameters are added to the contract, like minimum and maximum capacity, minumum and maximum period, the granularity that can be continuous or discrete, and in this second case, a list of possible utlization values, the importance (an integer number between 1 and 5) and the quality (a relative share, expressed as an integer number greater than or equal to 0).

S.Ha.R.K. implements the Elastic Task model [2] as an algorithm for distributing the spare capacity. In the Elastic Task model, each task is assigned a minumum and maximum utilization, and an *elastic parameter*. The system is seen as a set of *springs*, each one with its own elastic parameter, and the task utilization represent the spring lenght. When a new task is activated in the system the length of each task is adjusted to make "space" for the new task.

This model has been adapted and generalized for the FSF. Each server is seen as s spring and the quality parameter is the elastic parameter, while $U_{min} = \frac{C_{min}}{T_{max}}$ and $U_{max} = \frac{C_{max}}{T_{min}}$. Moreover, the algorithm is applied at every importance level. First, all the spare capacity is distributed at the highest importance level. After this, if there is still some spare capacity left, the algorithm is run at the second importance level, and so on. The pseudo-code for the algorithm is shown in Figure 4. In the figure, S(I) denotes the set of servers with importance level equal to I, while Quality(s) is the quality of server s. Finally, IsNextTargetImportance is a boolean value that is true if there is some task to be processed in the lower importance levels.

Currently, the implementation of this algorithm has one limitation. In fact, the original elastic task model was conceived only for tasks with deadlines equal to periods. Therefore, our spare capacity distribution algorithm currently only works when the server deadline is equal to the server period. In case the user specifies for some contract the parameter b_equals_t equal to false , the algorithm is automatically disabled.

3.4 Shared Objects module implementation

This module allows different servers to share objects thought mutex semaphores. One problem that arises when two aperiodic servers share a shared object through critical sections is that the server budget could be exhausted while one of the server is inside the critical section. This may lead to very large priority inversions and blocking times.

In FSF we decided to solve the problem by letting the server execute for some more (and its budget become negative) is insied a critical section when the budget was exhausted. However, this must be taken into account in the acceptance test. For this reason,

- a data type fsf_shared_obj_id_t to identify shared objects;
- a data type fsf_shared_obj_handle_t to identifies the *handle* to the shared object for each server (this will allow implementation of this mechanism even when the applications run in separate address spaces);
- a data type fsf_critical_section_data_t to identify the *list of critical sections* in the contract specification.

```
function recalculate_contract(bandwidth_t U)
{
  bandwidth_t current_bandwidth;

  /* The current bandwidth is the min bandwidth */
  current_bandwidth=SERVER_return_bandwidth(fsf_server_level);

  do {
    isok=1;

    while(s in S(I)) {
      TotalQuality+=Quality(s);
    }

    while(s in S(I)) {
      temp_U=U(s);
      delta_U=1-U;
      delta_U=delta_U*Quality(s)/TotalQuality;
      temp_U=temp_U+U(s)
      if (temp_U<=Umin(s))  U(s)=Umin(s);
      else if (temp_U>Umax(s)) {
        U(s)=Umax(s);
        isok=0;
      }
      else U(s)=temp_U;
    }
  }
  while (!isok || IsNextTargetImportance);
}
```

Figure 4: Pseudo-code of the algorithm for the negotiation (elastic task algorithm).

A run-time mechanism for mutual exclusion is not provided in FSF for two important reasons. One of them is upward compatibility of previous code using regular primitives such as mutexes or protected objects (in Ada); this is a key issue if we want to persuade application developers to switch their systems to the FSF environment. The second reason is that enforcing worst case execution time for critical sections is expensive. The number of critical sections in real pieces of code may be very high, in the tens or in the hundreds per task, and monitoring all of them would require a large amount of system resources.

Therefore, normal pthread_mutex_t variables are used, and the corresponding standard functions. To obtain the mutex corresponding to the shared object we use the following function:

```
int fsf_get_shared_object_mutex(
      fsf_shared_obj_handle_t obj_handle,
      pthread_mutex_t ** mutex);
```

The FSF API does not specify any particular synchronization protocol. In S.Ha.R.K. we decided to use Bandwidth Inheritance BWI [4]. The protocol is similar to the Priority Inheritance Protocol [6], but the pair (budget,deadline) is inherited instead of the deadline only. This protocol maintains isolation between non-interacting servers even in the case of misbehaviours in the use of critical sections.

The protocol was easily implemented in S.Ha.R.K. using an internal mechanism already implemented in the OS, called *shadow pointer*. In short, when a task blocks on a critical section, instead of removing it from the ready queue and putting in the blocked queue, we simply update its *shadow exec* pointer to the task that inherits the priority of the blocked task (e.g. the task executing the critical section). When the Generic Kernel looks for the first task in the ready queue to be executed, it checks its shadow exec pointer and follows the chain of pointer until it finds the task to be executed.

The overhead of this mechanism is limited and can be easily extended to implement BWI: the budget to be decremented is the one of the task at the head of the ready queue, independently of which taks is executing.

It remains to check that, if the budget is exhausted while in a critical section, the server is not suspended. This was done by simply modifing the rule for budget exhaustion in the GRUBSTAR module.

Notice that the BWI is completely transparent to the user: as a matter of fact, no specific API is present in the FSF for handling this server mechanism.

3.5 Dynamic Reclaimation module implementation

If an application uses less than it has been reserved, it is useful to reclaim this spare capacity and give it to the needing applications.

Many reclamation algorithms have been presented in the real-time system literature. In the context of EDF scheduling, we wish to mention the CASH algorithm by Caccamo, Buttazzo and Sha [3] and the GRUB algorithm by Lipari and Baruah [5]. The first one assumes that each server is a bounded workload server. Hence, as soon as a thread terminates execution, its remaining capacity is immediately reclaimed for the other tasks.

The second one assumes unbounded workload servers and can be used for any kind of system. However, it works in a *greedy* way: the reclaimed capacity can only be given to the currently executing task. This algorithm is implemented in S.Ha.R.K. in the GRUBSTAR module.

The idea behind the algorithm is the following. While a server S_i executes in the system, its budget is decremented accordingly. Suppose the server executed for a small amount of time Δt, during which there is no change in the system. If no reclaimation is present, then the budget is updated simply as:

$$B_i = B_i - \Delta t$$

If the GRUB mechanism is active, then the budget can be decremented as:

$$B_i = B_i - \Delta t + U_{free}\Delta t = B_i - (1 - U_{free})\Delta t.$$

U_{free} is the amount of bandwidth that can be reclaimed in the system. To measure this bandwidth, we distinguish between *active* and *inactive* servers. Active servers are severs contending for the processor, or servers that have already completed their workload but their bandwidth cannot yet be reclaimed. If a server completed its workload at time t, and it has $B_i(t)$ units of budget left, it will remain "active" until time $d_i - \frac{B_i}{U_i}$. After this intant, the server becomes inactive. The free bandwidth U_{free} is:

$$U_{free} = U_{max} - \sum_{S_i \in active} U_i$$

where U_{max} is the bandwidth assigned to FSF.

This mechanism does not jeopardize the schedulability of the system. Therefore, hard real-time tasks that are served by servers whose period is equal to the task's period and whose budget is greater than the task's WCET are guaranteed to complete before their deadline. A proof for this property can be found in [5].

The overhead of the mechanism is very low: in addition to the code needed for the normal server operations, we need to add one more timer event to handle the change of the server status from active to inactive; also, budget accounting introduces one more multiplication. The overhead and the performance of this mechanisms have been measured through synthetic experiments, reported in deliverable D-SI2.v3.

This mechanism has one limitation: it works only when all servers deadlines are equal to the periods. Therefore, the mechanism is automatically disabled when the user specifies for at least one server a deadline different from the server period.

3.6 Hierarchical Scheduling module implementation

S.Ha.R.K. modules must be organised in a certain way to support the hierarchical scheduling architecture. The organisation of the modules in S.Ha.R.K. is better explained by an example. Consider a system consisting of four applications, each one of them is supported by a dedicated server and a local scheduler. Application A1 consists of two threads T1 and T2 that are scheduled by an EDF local scheduler. Application A2 consists of two threads T3 and T4 that are scheduled by a RM local scheduler. Application A3 consists of 3 threads that are scheduled by a Table Driven local scheduler. Application A4 consists of 2 non real-time threads scheduled by a Round Robin local scheduler. The situation is depicted in Figure 5.

The modules to be installed in the S.Ha.R.K. OS are depicted in Figure 6. They implement the following functionalities (from the top).

Generic kernel Implements the standard generic interface of the S.Ha.R.K. OS, and delegates the scheduling decision to the scheduling modules.

EDF It is the global scheduler, it is used to schedule the servers. Servers are orders by the absolute server deadline. Each server will handle one or more threads of one application.

Round Robin It is used to schedule non-real-time threads. It supports a POSIX-compatible API. When a thread is created is initially assigned to this module. If no server is active (the queue of Level 0 is empty), then the kernel selects one task of this level for execution. Threads can be moved to other levels with the function.

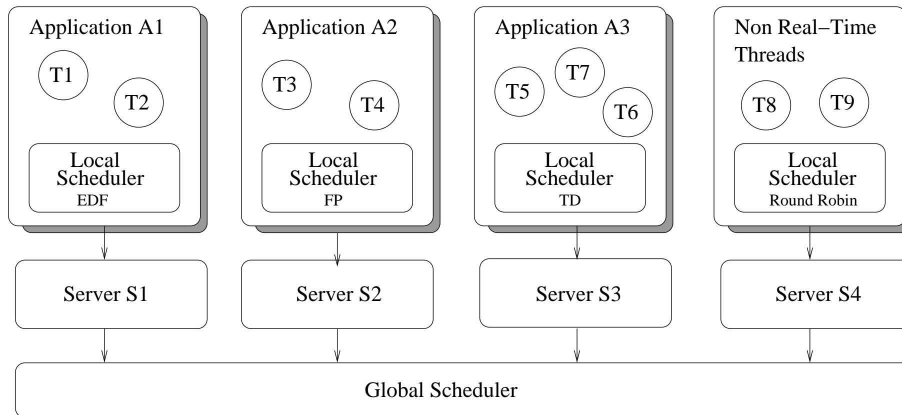

Figure 5: Example of hierarchical system with 3 applications.

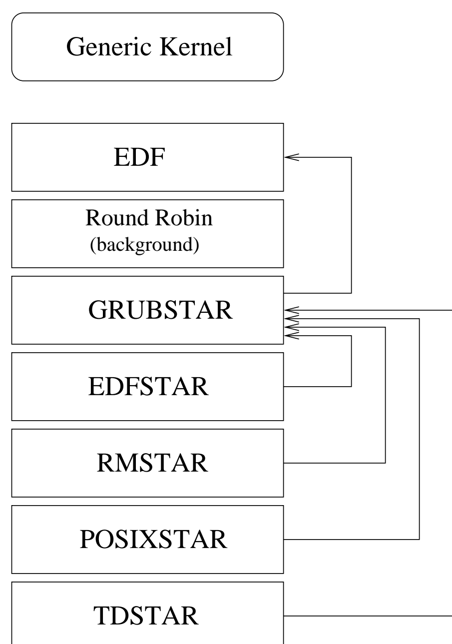

Figure 6: Organisation of the modules in S.Ha.R.K..

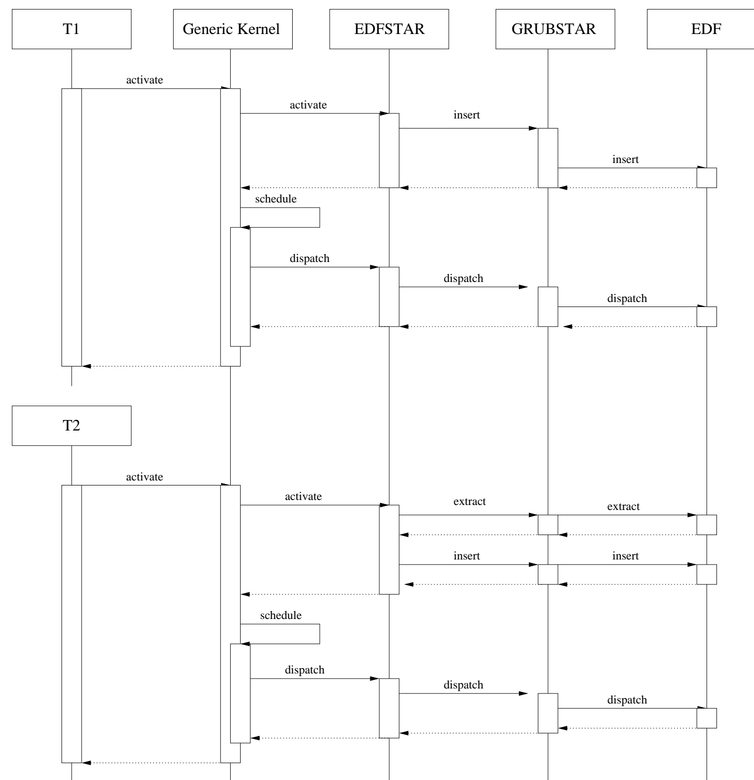

Figure 7: Sequence of messages in case of local preemption: task T2 is activated after T1 and preempts it.

GRUBSTAR . It is the module that manages the servers. As said, this module implements the GRUB algorithm.

EDFSTAR This module implements a local EDF scheduler. It manages threads and then pass them to the corresponding server of the GRUBSTAR module. In the example of Figure 5 it contains threads T1 and T2.

FPSTAR This module implements a local fixed priority scheduler. It manages threads and then pass them to the corresponding server of the GRUBSTAR module. In the example of Figure 5 it contains threads T3 and T4.

POSIXSTAR . This module handles the threads that are scheduled according to the Round Robin scheduler. In the example of Figure 5, it handles threads T8 and T9.

TDSTAR . This module handles the threads that are scheduled according to the Table Driven scheduler. In the example of Figure 5, it handles threads T5, T6 and T7.

Now, let's see how this scheduling structure works. First, we analyze the situation in which task T1 is activated on Server 1, and after a while, task T2 arrives with a shorter deadline and preempts task T1 (local preemption). The sequence of messages is shown in Figure 7.

The upper part of the Figure is similar to Figure 2. The only difference is that task T1 belongs to module EDFSTAR. Let us analyze what happens when T2 is activated.

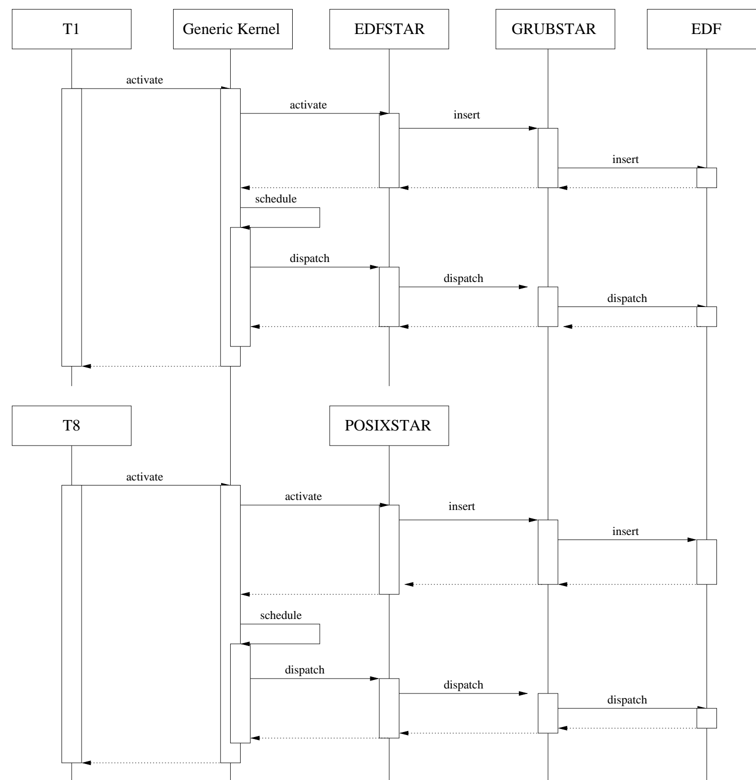

Figure 8: Sequence of messages in case of local preemption: task T2 is activated after T1 and preempts it.

As before, the activate function of the Generic Kernel is invoked, which calls the activate function of the EDFSTAR module. This module recognizes that a local preemption must be done by comparing the deadline of T2 with the deadline of the current local executing task T1. The first thing to say is that, for simplicity, only one task from every module can be present in module GRUBSTAR at the same time. This means that one one task from module EDFSTAR can be inside module GRUBSTAR at some point. Therefore, to perform the local preemption, module EDFSTAR has to remove task T1 by invoking the extract function of the GRUBSTAR module. In turns, the extract function of the EDF global scheduler is invoked, and task T1 is removed form all global queues. After this, module EDFSTAR can insert the newly executing task T2 in module GRUBSTAR, which in turns inserts the task in module EDF. Notice that, when inserted in the global EDF queue, T2 is assigned the deadline of the server, which is the same as the deadline it was assigned to T1.

After this "swap" between T1 and T2, the Generic Kernel recognizes that a context switch must be performed and invokes the dispatch function on the modules.

A slightly different situation is when a "global preeemption" must be performed. Suppose that, while T1 is executing, a task from Server 4 (Round Robin) is activated, and that Server 4 has a shorter absolute deadline with respect to Server 1. Therefore, a global preemption must be performed. The sequence of messages is shown in Figure 8.

Again, the upper part is identical to Figure 7. Suppose then that task T8 is activated. The Generic Kernel then invokes the activate of the POSIXSTAR module. Suppose that the module was idle. Therefore, it must insert its task in module GRUBSTAR. Remember that only one task from each module can be present in

module GRUBSTAR. In this case, since the POSIXSTAR module was idle, it has nothing to extract. Also, the executing task T1 is not removed, since it belongs to another module.

The insert function of GRUBSTAR in turn invokes the insert function on the EDF global scheduler. So, now, at the same time, task T1 and task T8 are both present in the global EDF queue, T1 with the deadline of Server 1, and T8 with the deadline of Server 4. If T8's deadline is the earliest one, then a context switch must be performed. Generic Kernel is notified of this change and invokes a dispatch function of module POSIXSTAR which is propagated to the other levels.

A Multimedia example application

In this section we present the extended example application developed to further evaluate the FIRST Scheduling Framework (FSF) implementation on the S.Ha.R.K. operating system.

The example application consists of a MPEG-2 video player and simple load generator. The player reads a MPEG-2 video stream from a file which it then decodes and displays according to the frame and display rate of the video. In order to perceive a good quality of the displayed video, it is important to keep the display rate at all times.

Load can then be generated to simulate possible disturbance caused by tasks from other applications competing for CPU time.

By using FSF we can isolate the player part of the application by inserting it into a server with a guaranteed bandwidth (enough for the decoder to run). The disturbance caused by the other tasks are "outside" of the server and thus cause no degradation of the decoder output quality. This disturbance is clearly visible as picture artifacts when running the same video decoder without FSF, where it lacks isolation.

A.1 Architecture Details

In this section we detail the MPEG decoder used in the extended example. Furthermore we describe the architecture of the player we implemented on S.Ha.R.K, both with and without FSF.

A.1.1 MPEG Player

The base software used for the extended example is the Berkeley MPEG decoder [?], a purely software based MPEG decoder written in C and developed for the UNIX/Linux platform.

Originally the decoder is a monolithic software lacking consideration for frame and display rate, instead it reads MPEG-2 data and outputs decoded video frames as fast as possible.

In order to achieve any real-time behaviour of the player we had to change the original architecture (monolithic) into an architecture that easily enables timely behaviour when decoding and displaying of frames. In the new architecture we propose, the player consists of three separate parts: input, decode, and display as can be seen in figure 9 that communicates using buffers.

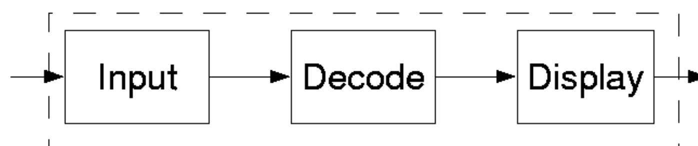

Figure 9: New player architecture, consisting of three separate parts.

The *input* is responsible for inputting data into the decoder from a source, a file or from the network (streaming). The *decoder* decodes the data from frames into pictures, and finally the *display* displays these pictures on the screen.

Within these different parts we have only modified small parts of the code, related to the input and output of data, otherwise the code is identical to the original code.

The new player architecture enables us to have a more fine grain control over the different parts of the player, i.e. the display and frame rate can be different, thus the decoding and display parts requires different timing behavior.

A.1.2 S.Ha.R.K Version

To port the player to S.Ha.R.K. we have to match the three different parts of the new player architecture with the existing task models. S.Ha.R.K provides a wide selection of different task models, from non real-time to hard real-time.

We match each of the three parts of the player into three different tasks: an input task, a decoder task, and a display task and the communication between the tasks uses ports [1].

We had to introduce task communication into the player code by replacing the read(), function that originally read data from a file, with our own read() function that reads data from a S.Ha.R.K. port instead. We also modified the display part of the player to receive its input from a port. All ports where configure to internally have a queue that can hold three messages.

Figure 10 show a more detailed view of the player architecture with the ports.

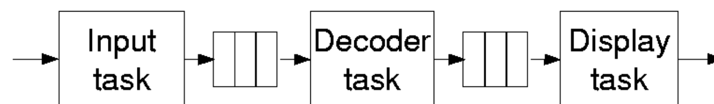

Figure 10: Architecture for S.Ha.R.K, with tasks and ports.

Input Task Ideally we would like the input task to directly read from the video file, but due to limitations in the DOS file system implementation of S.Ha.R.K we have to store the complete file in a buffer in the initialization phase of the S.Ha.R.K startup.

The input is modeled as a Non Real-Time (NRT) task that periodically reads data from the buffer, preloaded with the complete video file, and sends this data to a port shared with the decoder task. Data is read and stored in messages, chunks of size 2048 bytes. If the shared port queue is full the input task will be blocked until there is space available in the buffer.

Decoder Task The decoder task is also modeled as a NRT task. It reads input data from the port shared with the input task and outputs decoded frames to a port shared with the display task. The decoded frames are actually stored in a dynamically allocated memory, and the pointer to this memory is sent through the port to the display task (which deallocates the memory).

The decoder task is non periodic in its nature, reading input data when its needed in the decoding process, continuously decoding frames. Since a video frame can be bigger than the data chunks, of 2048 bytes read

[1] A S.Ha.R.K. mechanism to exchange messages between tasks.

from the port shared with the input task, more chunks of data needs to be read before the complete frame can be decoded and sent to the display task.

The decoder task is also blocking on the port shared with the display task, i.e. if the decoding is faster than displaying the decoder will block.

Display Task In our player architecture we consider the display task to be the most important task. It is responsible for keeping a as constant display rate as possible. Constant display rates are perceived by the human eye as a better quality than fluctuating display rates.

The display task is therefore considered to be a hard real-time task, with a period set according to the display rate.

The display task receives a pointer to a decoded frame through the port shared with the decoder task. After displaying the picture the memory allocated to the picture must be deallocated (it was allocated by the decoder task).

A.1.3 Execution Time Estimation

We measured the worst case execution time of the display task by repeating the decoding and displaying of a movie during 48 hours. As a result we got a wcet of 18 milliseconds. The execution time of the display task could be decreased if we used the possibility of writing directly into the frame buffer of the graphics card (currently only supported for some Matrix cards). The period of the display task is set to 40 milliseconds, which corresponds to a display rate of 25 pictures per second.

We could not measure the execution time of the decoder task as it has an unbounded workload behaviour, it reads data at various points in the code and outputs the picture att different places depending on frame type.

A.1.4 S.Ha.R.K. Version with FSF

Here we describe the various contracts used by the extended example application for requesting FSF servers. Each of the parts of the application requests its own FSF server and thus we have to specify 4 contracts (input, decoder, display, and load generation), details for each of the contracts are shown below. The extended application only requires that the *core* part of FSF is used.
More details of the FSF contract can be found in deliverable [?].

The resulting configuration of FSF servers and tasks are shown in figure 11.

Input task contract The input task requests a FSF server with the following contract:

Contract parameters	Value
Minimum budget	3ms
Maximum period	30ms
Deadline	None
Workload	Indeterminate
Deadline miss notification	None
Budget overrun notification	None

The scheduler used in this server is the FSF_RR scheduler (Round Robin) and because we don't know the exact execution pattern of the task (it can be blocked) we use an indeterminate workload for the server.

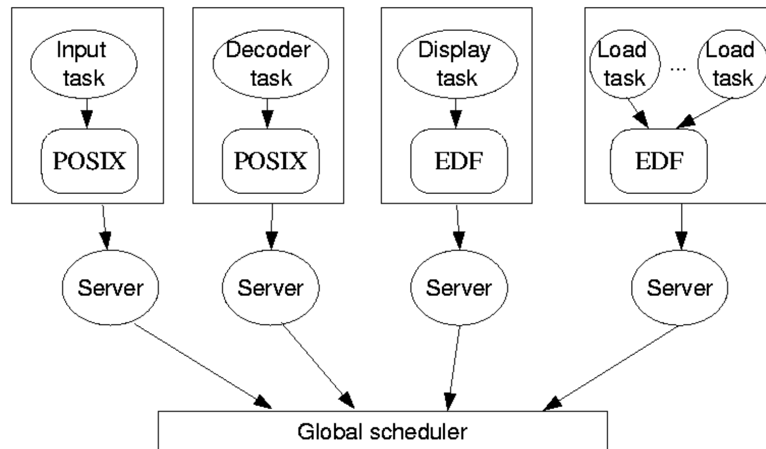

Figure 11: Architecture using FSF on S.Ha.R.K.

Decoder task contract The decoder task requests a FSF server with the following contract (same as for the input task):

Contract parameters	Value
Minimum budget	3ms
Maximum period	30ms
Deadline	None
Workload	Indeterminate
Deadline miss notification	None
Budget overrun notification	None

The scheduler used in this server is also the FSF_POSIX scheduler and again because of the unknown execution pattern we use an indeterminate workload server.

Display task contract To ensure that the display task can execute without any disturbance we create a contract for the task to run in a server by itself. The contract requests a FSF server with same parameters as the display task itself (a wcet of 18ms and a period of 40ms) using the following contract:

Contract parameters	Value
Minimum budget	18ms
Maximum period	40ms
Deadline	None
Workload	Bounded
Deadline miss notification	None
Budget overrun notification	None

We can use a bounded workload since we know the exact execution pattern of the display task. The scheduler used in this server is the FSF_EDF algorithm.

Load generator contract The load generator has a contract requesting a server using the remaining capacity of the system. Since we isolate the load in a FSF server there is no possibility that the load can disturb any other part of the system, it can just use the capacity of the server it is running in.

Contract parameters	Value
Minimum budget	1ms
Maximum period	10ms
Deadline	None
Workload	Bounded
Deadline miss notification	None
Budget overrun notification	None

The load generator used the FSF_EDF algorithm to schedule all its tasks.

A.1.5 Running the player application

To run the demo application there must be a video file present in the same directory as the executable. After initialization and startup, the application is started by pressing '1' key. Load generation tasks are created by pressing the "2" key, where each task demands a utilization of about 0.1.

A.2 Summary

In this document we describe the architecture of a real-time MPEG-2 video player running on the S.Ha.R.K operating system using the First Scheduling Framework (FSF) to achieve isolation from possible disturbances caused by other applications.

The base decoder used for this implementation is the Berkeley MPEG decoder from [?] which is a software only decoder written in C.

To introduce real-time timing behavior in the decoder we modified the architecture from monolithic (original Berkeley model) into a multi-task architecture. In our new architecture the decoder has been split into three separate parts. The first part is the *input task*, which is responsible for reading input data (an MPEG video stream) from a source file and storing it in a buffer. Secondly, the *decoder task* uses the data from the input to decode the actual video frame into a picture, which is stored in buffer shared with the display task. The final part is the *display task*, which takes the pictures output by the decoder task and displays them on the screen according to the display rate.

In the proposed architecture the display task is considered to be the most important one. This is because a constant display rate is perceived as good quality by the human eye.

FSF introduces the possibility to isolate the player application, and more importantly the display task from any possible disturbances by creating a single server for it. Other applications runs in other servers and can therefore not use more budget than what is allocated to that server. FSF ensures that servers cannot disturb each other, thus the server where the display task is running is guaranteed its budget which allows the display rate to be constant.

References

[1] L. Abeni and G.Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, december 1998. IEEE.

[2] Giorgio C. Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

[3] Marco Caccamo, GIorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE, December 2000.

[4] Gerardo Lamastra, Giuseppe Lipari, and Luca Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the Real-Time Systems Symposium*, London, Dec 2001. IEEE.

[5] Giuseppe Lipari and Sanjoy Baruah. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transaction on computers*, 39(9), September 1990.