Czech Technical University,
Faculty of Electrical Engineering,
Department of Control Engineering

MASTER'S THESIS

Porting of resource reservation framework to RTEMS executive

Master Programme: Space Science and Technology
Supervisor: Ing. Michal Sojka, Ph.D.

Prague, May 27, 2011

Author: Petr Beneš

Space Master is a Joint European Master in Space Science and Technology.

Erasmus Mundus helps out thousends of students to gain an international education every year.

Department of Control Engineering, Czech Technical University, Prague.

Lulea University of Technology, the main partner of the Space Master consortium.

This thesis is mainly a contribution to FRSH/FORB project.

This thesis is a contribution to RTEMS project under a supervision of its developers.

This thesis is supported by Google Summer of Code opportunity for software oriented students worldwide.

Declaration

I declare that I have written this thesis myself and that I have not used any sources or resources other than stated for its preparation. I further declare that I have clearly indicated all direct and indirect quotations.

Prague, May 27, 2011 Petr Beneš

Acknowledgements

I would like to express my thanks to Michal Sojka, my thesis supervisor, who was the main leading hand on the way to a successful thesis accomplishment even in times of his tremendous work overloads. Furthermore, I would like to thank to all the professionals who either shared their experience with me or encouraged to pass the project to the waters of better quality and general usefullness. Namely Pavel Píša, Gedare Bloom, Tommaso Cucinotta and Joel Sherrill.

Abstract

The aim of this thesis is to create a port of a FRSH/FORB project for RTEMS operating system. FRSH/FORB is a software middleware implementing a resource reservation framework for tasks running on a certain operating system, in particular real-time embedded systems. So far, the framework runs on Linux platform, which is only a general purpose operating system. Therefore, the port is to be done for RTEMS. The main steps to achieve this are redesigning a part the framework in order to become independent of the underlying operating system, preparation of the latest version of RTEMS, implementing an EDF scheduler with a sporadic server, linking the framework together with RTEMS, and finally testing with an appropriate application. The framework has been refactored and scheduler implemented and tested. Only the last part of linking was not finished because of lack of time.

Keywords: real-time, middleware, RTEMS, EDF

Objectives

1. Familiarize yourself with FRSH/FORB framework and its current state.

2. Modify FORB middleware to allow running applications in a single address space.

3. Port the FRSH framework from Linux to real-time executive RTEMS.

4. Implement some mechanism for CPU resource reservations such as constant bandwidth server and interface it with the FRSH framework.

Contents

List of Figures

List of Acronyms

API Application Programming Interface

AQuoSA Adaptive Quality of Service Architecture

BSP Board Support Package

BWI Bandwidth Inheritance

CBS Constant Bandwidth Server

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CVS Concurrent Versions System

DM Deadline Monotonic

EDF Earliest Deadline First

ELF Executable and Linkable Format

FCB FRSH Contract Broker

FORB FRSH Object Request Broker

FOSA FRSH Operating System Adaptation

FRA FRSH Resource Allocator

FRESCOR Framework for Real-time Embedded Systems based on COntRacts

FPS Fixed Priority Scheduling

FRM FRSH Resource Manager

FRSH FRESCOR Scheduler

FWP FRESCOR WLAN Protocol

GRUB GRand Unified Bootloader

GPOS General-Purposes Operation System

IDL Interface Description Language

IOP Inter-ORB Protocol

MSB Most Significant Bit

NPCS Nonpreemptive Critical Sections

OMK Ocera Make System

OS Operating System

PCP Priority Ceiling Protocol

PIP Priority Inheritance Protocol

POSIX Portable Operating System Interface for Unix

PreCP Preemption Ceiling Protocol

QoS Quality of Service

RM Rate Monotonic

RR Round Robin

RT Real-Time

RTEMS Real-Time Executive for Multiprocessor Systems

RTOS Real-Time Operating System

SMP Symmetric Multiprocessing

TCB Thread Control Block

Chapter 1

Intoduction

1.1 Motivation

Nowadays, real-time systems keep gaining significance in the field of control engineering. A lot of applications require safety-critical properties such as space technologies where an unpredicted timing violation may result into a mission fail responsible for a loss of an astronomic amount of money. Secondly, another group of applications such as media transmission considered as soft real-time put emphasis on *Quality of Service*(QoS) [Abeni et al., 2005] in which case the important measures are statistical guarantees and cost efficiency.

Development of real-time applications is typically a complex task. It is not only sufficient to write the application logic but one also needs to ensure that the application meets all temporal requirements such as deadlines. For this reason it is difficult to apply component-based development methodologies. Normally, composing the application from well tested components leads to better applications and shorter development time, but in the case of real-time applications even the most tested component can behave incorrectly if they do not have sufficient amount of resources to perform their functionality.

Resource reservation framework helps to temporally isolate the application components by reserving the resources for the individual components. By incorporating on-line schedulability tests the availability of reserved resources can be guaranteed.

An availability of such a middleware on a hard real-time operating system will yield a top quality of temporal guarantees for all kinds of simultaneously

running services while still providing a time and cost effective development conditions for application designers.

1.2 Contribution

This thesis aims at enriching FRSH/FORB resource reservation framework (later only *framework*) with hard real-time capabilities that open new area of possible utilization mentioned above. The thesis mainly refactors the current version of middleware and also contributes with new features specific for the underlying operating system RTEMS.

Hard real-time capabilities of the framework. Adapting the back-end of the framework to the RTEMS operating system makes use of its all features, especially the hard real-time timing capabilities, which is necessary for a certain scope of applications. Thereafter, the framework becomes capable of handling highly critical systems.

EDF scheduler. An *Earlies Deadline First* (EDF) scheduler for RTEMS has been designed and implemented. As a matter of fact, this is a part of overall scheduling algorithm for FRSH/FORB project, but it will also serve as an optional scheduler for a general use of RTEMS project users. The scheduler will be merged directly into RTEMS operating system, so that everybody can use it.

Hybrid priority representation. As a compromise between hard real-time deadline driven scheduling with a possibility of executing background tasks with no real-time requirements and a low overhead is a hybrid priority representation that has been designed. This concept making use of a simple mathematical trick saves a lot of effort when implementing EDF and possibly also other, more complex schedulers under certain circumstances, see Section 5.2.3.

Temporal isolation of tasks in RTEMS. A *Constant Bandwidth Server* (CBS) as a reservation-based scheduling policy atop of EDF has been designed and implemented in order to guarantee temporal isolation of tasks essential for resource reservation on the RTEMS platform.

1.3 Structure of the thesis

In the very first part, this thesis includes an introduction about the FRSH/FORB project which is being enhanced, its aim, description and use in the Chapter 2. Since the project is very modular, its components will be consequently described separately.

Further, the Chapter 3 presents necessary changes and refactoring of the FRSH/FORB framework that have been performed.

Next, a brief description of RTEMS operating system, its properties especially in terms of hard real-time behavior is provided in Chapter 4. Also the application programming interface will be discussed which is necessary to comprehend in order to figure out what options and obstacles will arise concerning the API matching between FRSH/FORB and RTEMS.

An introduction to basic concept of real-time scheduling, description of scheduling algorithms and related features, their pros and cons related to FRSH/FORB needs, and finally a design of a low-level scheduling Earliest Deadline First scheduler will be described and justified in Chapter 5.

The Chapter 6 describes how a temporal isolation of task will be achieved by a Constant Bandwidth Server implementation. The temporal isolation property of CBS is an essential property for resource reservation capability of the FRSH/FORB project. The CBS utilizes low-level Earliest Deadline First scheduler

The implementation details of RTEMS scheduling policy based on previous design is described in Chapter 7.

Validation of all parts of work in terms of tests is presented in Chapter 8.

Finally, conclusions are made in the last Chapter 9 also indicating the current stage of project and commenting on its deficiencies. Also proposals for another future work are summed up in Section 9.1. Since this thesis is actually not the end of effort, some of the points will be finished in a very close time horizon.

Details on how to get and compile both RTEMS and FRSH/FORB can be found in Appendix A.

Chapter 2

About FRSH/FORB project

FRESCOR (Framework for Real-time Embedded Systems based on COntRacts) is a finished research project intended to create an effective technology for embedded real-time distributed systems allowing users and application programmers a simple approach to develop flexible, reconfigurable and distributed architectures using the most advanced techniques along with shortest time-to-market [fre, 2008]. The project was executed by a consortium of several European partners and financially supported by the European Union. The result of this effort is FRSH/FORB resource reservation framework [frs, 2011].

Distributed systems. The main emphasis of this project is put on distributed systems which means that not only a resource reservation on a single computer is provided, but also a possibility to share computational capacity among multiple computers. The framework provides to users an abstraction yielding a very convenient development environment for distributed systems (see Section 2.2.3).

2.1 Resource reservation framework

Every computer system running a set of applications provides with resources which are shared by these applications. These resources may be divided into *active* and *passive*. The active resources are the ones that can execute tasks, thus e.g. a CPU. Passive resources are additional necessary resources such as LAN or disk. All of these resources are under control of an operating system and applications utilize these resources. Every time a resource is shared by multiple applications, it is necessary to come up with some access control mechanism in order to serve all the application requesting the

particular resource. In case of *General-Purpose Operating Systems* (GPOS), there is no motivation to study any guarantees on resource assignment since the applications are not critical. Moreover, some of the resources may be considered sufficiently large to run what is necessary. In case of *real-time operating systems* (RTOS) the situation is different. Since the applications require a guarantee that no deadlines are going to be missed, the resources have to be seen as limited no matter how big they are.

Reserved resources. The idea of resource reservation is based on assumption that a task is capable of a proper execution as long as it has necessary amount of resources. The resource reservation framework in order to ensure a full functionality of each application sharing active and passive resources in a real-time manner has to take care of a proper distribution and reservation of these resources [Molnár et al., 2008]. This will be carried out using *contracts*, which provide the possibility for the applications to indicate their resource requirements, ask for them and keep them until the end of their life. Since the amount of each resource is limited, the framework has to keep information on how much resource is already in use and decide on-line whether enough of resource is available or not. This is managed by blocks handling resources described in Section 2.2.

Contracts. As the framework name already suggests, the resource reservation is accomplished by a contract based negotiation with applications. In order to reserve all necessary resources that are requested by the application, a contract has to be prepared and negotiated with the framework.

Quality of Service. Moreover, given a *spare capacity* of resources is available, currently not being used by any application, this spare capacity may be redistributed across the running tasks. This is a very useful feature for a certain area of applications such as media transmission, where no strict deterministic deadlines are wanted, but rather a *Quality of Service* is requested. Such applications are usually not optimized in terms of criticality, but rather in terms of flexibility and cost. The applications having a possibility to run in several levels of QoS [Abeni et al., 2005] tend to utilize the resources as much as possible and only a lower bound of the quality is necessary to be maintained. Therefor, the possibility of spare capacity redistribution has its place in this framework [Sojka et al., 2011].

2.2 Fremework structure

The FRSH/FORB project consists of several more or less independent software projects,

- FRSH Object Request Broker (FORB),
- FRESCOR Scheduler (FRSH),
- FRESCOR WLAN Protocol (FWP),
- FRESCOR Network Adaptation (FNA),
- Wireless sensor network protocol ITEM.

later denoted only as *components* [Molnár et al., 2008], some of which will be in more detail described in latter sections. Each of the components provide with some kind of functionality. The modularity is a very convenient design for possible adjustments. A block diagram of the middleware underlining the layers and mutual communication is depicted on Figure 2.1.

And another software projects have been integrated:

- FRSH Operating System adaptation layer (FOSA),
- Cluster-Tree wireless sensor network
- ORTE middleware (developed in EU project OCERA)
- OMK make system (developed in EU project OCERA) [omk, 2010]

2.2.1 FRSH

FRSH is the highest level part of FRSH/FORB including functionalities of a general use. There should not be any platform and resource specific parts (however, this is not exactly the reality). The FRSH API is an interface directly being used by application programmers while implementing their systems. It provides with the contract based reservation negotiation for available resources.

Resource allocators. These are responsible for handling a particular resource by means of communicating with a resource scheduler. The functions of a resource allocator (FRA) are basically assigning and removing resource fractions to and from tasks and also handling the spare capacity.

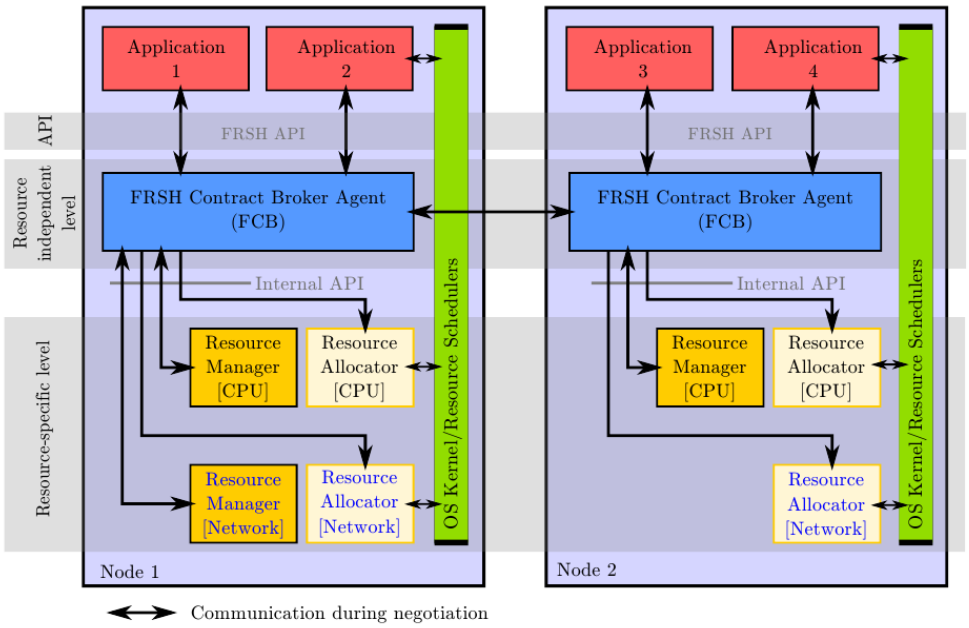


Figure 2.1: Structure of the FRSH/FORB middleware [Sojka, 2010].

Resource managers. The responsibility of calculation how much of a resource is currently in use and whether another task may obtain a fraction of a resource or has to be refused in order to maintain the reservations of the tasks already sharing the resource lies upon the resource managers (FRM). Every resource has its own resource manager deciding about possible resource allocation and schedulability (see Section 5.1.2) of the system as a whole.

In order to add a resource, a new Resource allocator and a new Resource manager have to be implemented.

Contract broker. The responsible entity distributing the contracts between application and involved resource managers is called *Contract Broker* (FCB). The FCB announces the application the result of negotiation. That means whether the contract was accepted or not.

2.2.2 FOSA

FRSH Operating System Adaptation (FOSA) layer is the back-end of the middleware providing an independence of the higher level components on the underlying operating system.

2.2.3 FORB

FRSH Object Request Broker (FORB) is an implementation of CORBA middleware. Although the CORBA protocol [cor, 2011] comprises a huge amount of features, only a small part of them has been implemented in FORB. Particularly for practical purposes and simple usability. This middleware ensures a communication among parts of software residing in the same process, another process or even another node, along with providing an abstractized interfaces among the *objects* so that there is no need to know where the components are actually located. This feature is a necessary building block for distributed and modular system development approach.

The objects are main entities in the FORB communication and own a certain set of methods. When a method of an object is invoked, it is performed in a synchronous manner, thus the calling entity waits for a return value. Moreover, the methods are not invoked directly, however, a *stub* is called instead. The communication sequence is shown on Figure 2.2.

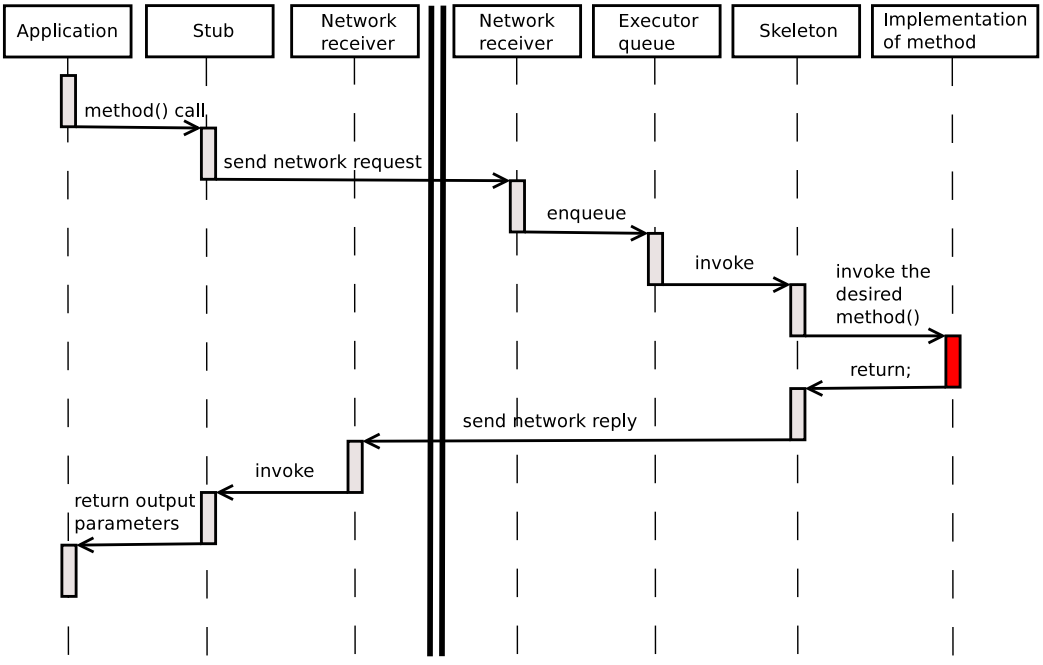


Figure 2.2: Sequence diagram of FORB inter-thread invocation of a method. On the left hand side there is a client node including the application and on the right hand side there is a server node including the implementation of called function.

Stubs. The stub is a mean of abstractization of a method according to IDL language on the side of caller (client). It is not necessary to know whether the function is local or remote. The stub figures this out and takes appropriate steps to find and invoke the function.

If an object (or method) is found remote (implementation is not available on the same peer), the *Inter-ORB protocol* (IOP) defines rules for data serialization and transmission between peers (other computers). In case of a remote call, a message is to be serialized, sent and deserialized on the destination peer, where a *skeleton* of a method deserializes the request and finally invokes the desired method (Figure 2.2).

Skeletons. Skeleton has the same function as stub but on the side of called object (server part). It invokes the desired method and sends a reply with output paramenters back to the caller in a way dependent on whether the caller is remote or local.

Possible invocation procedures in FORB were so far:

- Direct invocation (A method is called directly when the implementation is available),
- Inter-process invocation (This is taken into account only in case of OS having multiple processes, such as Linux. UNIX sockets were used),
- Inter-node invocation (A serialized message has to be sent via network).

Executor and executor queue. Each object includes a thread that is responsible for executing all incoming requests. This thread is called *Executor*. The incoming requests are stored in an *executor queue* of an object and waiting for being processed.

IDL language and compiler

The IDL (*Interface description language*) is a language describing an arbitrary interface regardless of which language is used for the interface implementation or what machine (endian) it is to be run on. This makes different languages and computers communicate with each other. The IDL compiler translates the IDL language into already a particular language such as C. The result is that the interfaces (stubs and skeletons) are created automatically and the user cares only about the IDL definition [Molnár et al., 2008].

Chapter 3

Changes and implementation in FRSH/FORB

This chapter introduces the first part of work, implementation details encountered during refactoring the framework into a single address space, this will be presented in Section 3.1, and then, issues related to matching and porting FRSH/FORB to RTEMS are presented in Section 3.2.

3.1 FORB running in a single address space

Threads and processes are two possible processing units in a multitasking operating system. Unlike thread, processes are not only an execution of a program code, but has also assigned a part of computer memory space called *address space*. Each thread is associated with a process and each process is associated with a program. However, not all operating systems, especially the ones for embedded purposes, which tend to be small in size and do not need all features provided by GPOS, implement processes and all threads run in one address space as a single process or program. The feature of multitasking is simply provided by threads.

The initial design of FRSH/FORB middleware was performed for sake of Linux platform which provides with processes. Therefor, the active components of the middleware described in Section 2.2 are implemented as separate processes. The real-time operating system RTEMS, however, executes applications as a single process. As we need to port the complete middleware on RTEMS, this property is a major obstacle and we are forced to refactor the middleware in a way all components run as a single process, thus they share

the entire memory. Moreover, it is desirable that the refactored version will run in Linux the same way as before and maintain a high degree of backward compatibility so that no change is notable from the user's point of view.

In the following sections I present necessary steps and problems that arose during this procedure, along with means to validate its functionality on Linux.

Validation of these steps is presented in Chapter 8.

3.1.1 Current executor and thread specific data

The FORB layer as described in Section 2.2.3 is responsible for communication between remote objects according to Inter-ORB protocol, so that a distributed system can be easily established without any need for applications to care about low-level communication details. In case a method belonging to an object is to be invoked, the layer is supposed to determine where the object owning the called function is implemented. The actual implementation may be invokable from the place where it is called from in which case a *direct invocation* can occur. In the opposite case when the object is either in a different process or on another node in the distributed network, a *remote invocation* occurs.

In case of a single address space it may happen that an object previously implemented in a different process, thus invoked remotely, is available in the same address space. Since the methods of objects are not reentrant, multiple threads are not allowed to access the data. Therefor, it is not possible to invoke the method directly, but an invocation request has to be passed to a thread executing all requests on the object. This thread is called *executor*.

Thus, it is essential to distinguish this case in order to avoid concurrent access. Let us call this *inter-thread invocation* in detail described in Section 3.1.3. It is a blend of local and remote invocation procedures, it serializes the request and inserts into the executor queue of requests without a need of sending via network. This is depicted on the Figure 3.1.

First of all, in order to be capable of deciding which kind of local invocation we require, we have to figure out which thread we are in before invocation. For this purpose a function `forb_get_current_executor()` was added. The function uses POSIX thread specific data pointing to an assigned executor. The thread specific data are initialized as an executor is started

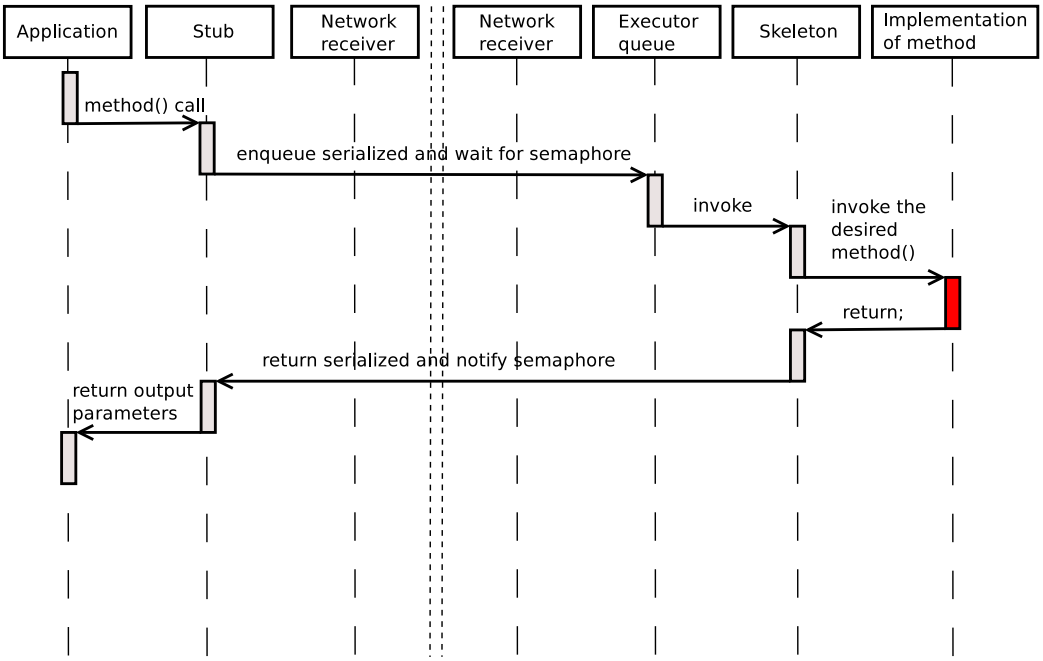


Figure 3.1: Sequence diagram of FORB inter-thread invocation a method. On the left hand side there is a client application and on the right hand side there is a server object including the implementation of called function. Both entities share a common address space.

(`forb_executor_run()`). If the thread does not belong to any executor but to an application a `NULL` pointer is returned. This uniquely separates local and inter-thread invocation.

3.1.2 Invocation procedure decision

Since we are now able to figure out the relation between a calling and destination entity, we can make a decision on how to start a communication. This is a responsibility of stub, which is called first in any case. The possible invocation procedures are now:

- Direct,
- Inter-thread,
- Remote (inter-node).

As for the implementation, the condition

```
forb_object_is_local(obj) &&
forb_get_current_executor() == forb_object_get_executor(obj)
```

where `obj` is the target object determines the local invocation case. Consequently, the rest of cases are handled the same because the procedure is to a certain extent very similar. The procedure of request preparation in stub continues with no change as a remote invocation case. The consequent distinction emerges only in implementation of the request and response handling functions.

Thus, the stubs were reimplemented in a corresponding way. Respective their template only because they are automatically generated.

3.1.3 Inter-thread invocation implementation

The newly introduced inter-executor invocation procedure requires a couple of changes in the FORB structure.

In case of a remote invocation a complete serialized message represented by (`struct request`) is to be made up and sent via network. In the other node it is deserialized and enqueued as a request (`struct exec_request`) for an executor.

Let us begin with the same scenario just omitting a few steps. The request consists of serialized parameters, request header (specifying what object, method is being requested, ...) and a message header specifying other information. Since the requested object and executor is in the same address space, instead of a full serialization of the message with *request header* and *message header*, it is sufficient to keep only the request header which mainly determines which function is requested. The message header is a lower level indication of a message type when sending via network and is not at all of our interest.

Now, as we decided to pack up the request and enqueue it for the executor of destination object, the previous implementation gave a little complication. The representation of request on the side of caller `forb_request` is different to representation on the side of executor `forb_exec_req` because also the data to be kept there are different. On both sides C structure components are filled out.

The request represented by structure `forb_exec_req`, once prepared, is a

member of the executor queue and thus, it would be desirable to fill out only that one. However, some necessary information such as function name and its input parameters are still in a serialized form (`FORB_CDR_Codec codec`). But as the remote invocation procedure (in stub) already serializes these data into `forb_request`, we will keep this structure as well. Moreover, there is no need for the `forb_exec_req` to keep any output parameters or return value because after the called function finishes the result in case of remote invocation just has to be sent out. As for the inter-executor invocation, the output data (buffer) is directly saved to the corresponding `forb_request`. Which one is the corresponding one is determined by the newly added pointer `forb_exec_req.input_request`.

The request represented on stub (caller) side (`/src/forb/src/request.h`) [1]:

```
struct forb_request {
        CORBA_unsigned_long request_id;
        FORB_CDR_Codec cdr_request;
        FORB_CDR_Codec *cdr_reply;
        gavl_node_t node;
        forb_object obj;
        unsigned method_ind;
        char *interface;
        struct forb_env *env;
        forb_syncobj_t reply_ready;
        forb_syncobj_t *reply_processed;
        + unsigned end_of_header_index;
};
```

The `end_of_header_index` is added for sake of the changed serialization method which omits message header.

The request represented on skeleton side (`/src/forb/src/exec_req.h`):

```
typedef struct forb_exec_req {
        unsigned request_id;
        forb_server_id source;
        forb_object obj;
        unsigned method_index;
        FORB_CDR_Codec codec;
        ul_list_node_t node;
        + enum forb_exec_req_type request_type;
```

[1] plus sign denotes newly added components

```
        + forb_request_t *input_request;
        + char *interface;
} forb_exec_req_t;
```

For a distinction whether a remote or inter-executor invocation occurs, the `forb_exec_req.request_type` indicator has been added.

```
enum forb_exec_req_type {
        FORB_EXEC_REQ_LOCAL,
        FORB_EXEC_REQ_REMOTE
};
```

3.1.4 Forbrun, framework initialization

Previous usage of the middleware in Linux was based on existence of separate programs. These programs (processes) were executed form a command line. In order to run a user application utilizing the FRSH/FORB middleware, it was necessary to start

- FRSH Contract Broker,
- required FRSH Resource Managers,
- the user application itself.

Now, as we have just a single executable, we have to ensure a proper initialization of all these components which are to be compiled as shared libraries (in Linux). Each shared library has a `forb_main()` function as the entry point.

A simple program called *forbrun* was created, which is given a set of command line parameters. According to these parameters `forb_main()` functions with appropriate parameters belonging to them are executed. Each shared library as well as the forbrun itself may have specific command line parameters. Thus, if an user application is to be run it is necessary to execute the forbrun with parameters of the shared libraries' names along with parameters belonging to them. This makes the execution a little messy, however, it is still possible to run the components separately. In that case you execute forbrun several times, each times with only one shared library as a parameter. It leads to the same effect.

The syntax for forbrun program is

```
forbrun [ options ] -- <forb-server>.so [ options for forb_main() ]
    [-- ...]
```

3.1.5 Backward compatibility scripts

The previous change of execution procedure (Section 3.1.4) may cause troubles to applications already designed with FRSH/FORB in Linux before. Therefor it was necessary to come up with some kind of backward compatibility strategy. For this reason, each of the components is assigned a shell script having the same name as the component. The script is just a replacement for the previous executable component and its content is only a forbrun call with a parameter of the corresponding component shared library plus its own parameters. Verification whether the changes did or did not harm functionality in Linux is presented in Chapter 8.

3.1.6 Conclusion

Since the middleware is composed of several layers, it was necessary to alter only the FORB layer and the rest stays untouched. In general, the necessary steps included recreating the components from processes into threads having a different name of the main function since there may be only one such function. Consequently, communication means have to be established and implemented for the inter-thread execution and finally all of this still has to be executable in Linux in the same manner as before, preferably maintaining a backward compatibility with the previous invocation strategies.

3.2 FRSH/FORB to RTEMS adaptation

3.2.1 FOSA for RTEMS

As the FOSA (Section 2.2.2) layer is an operating system abstraction, it is necessary to create one for RTEMS. However, it is not necessary to do so from scratch because the RTEMS operating system already inherits a significant portion of POSIX standard (Section 4.2). Therefor the FOSA sources for Linux and RTEMS do not differ too much but some features are not included in RTEMS. For example `siglongjump` is not implemented properly. As a matter of fact, just a proper testing will figure out how compliant with POSIX RTEMS actually is.

A main difference between RTEMS and Linux is a task identification. While in Linux two identifiers are necessary, for threads `pid_t linux_tid` and for processes `pid_t linux_pid`, in case of RTEMS only a task is to be identified `rtems_id rtems_tid`.

3.2.2 Cross compilation, libraries creation

Since the RTEMS support of shared libraries is not very mature, we were forced to make up a different way of compilation of the middleware components. The easiest way is to compile all the libraries as static instead of dynamic. The precompiled RTEMS core is actually not a final executable either, but just a set of built object files waiting for user application to be compiled and linked together with in order to yield a final ELF file which is already capable of executing on a specific target or in an emulator.

The platform dependent compilation of FRSH/FORB components as libraries is carried out by defining a specific macro `forb_share_LIBRARIES` either as `lib_LIBRARIES` for RTEMS or `shared_LIBRARIES` for Linux, which is in both cases already an OMK [omk, 2010] variable.

3.2.3 Platform dependent changes in FRSH/FORB

This section finishes the enumeration of all necessary adjustments on the way of obtaining a running framework under RTEMS in as much as possible similar way as in Linux. Mostly the changes were accomplished using a conditional compilation where the definition of `RTEMS` macro introduces a RTEMS specific part. Another, more specific macros are used in cases where the condition for compilation is not just an assumption of RTEMS platform but rather a particular library.

Select vs. Epoll. A very significant issue concerning multiplexing network sockets on a server has been identified in FORB (`proto_inet.c`). For an asynchronous servicing of multiple clients the library *Epoll* [epo, 2010] is being used. Although this library is one of the best of this kind, it is platform dependent and definitely not supported in RTEMS. Therefor, a much more common *Select* [sel, 2011] library has to be used for RTEMS, which is implemented. At this point I have to say that the platform dependency is not very convenient as the FORB layer should not be platform dependent and any conditional compilation using a preprocessor makes the code very hard to read.

Main functions. The above described issue of static compilation and linking of framework component libraries (Section 3.2.2) yields another problem. Not only the address spaces of previously separate components have been merged, but also name spaces became one. As each component required a `forb_main()` function, as an entry point to the dynamically loadable library, in case of a single name space, the functions have to be called differently. This is accomplished by introducing a generalizing macro `FORB_MAIN()`[2] that renames the `forb_main()` functions by concatenating a component specific suffix e.g. `forb_main_fcb()`.

Forbrun in RTEMS. All required components have to be started within the framework initialization in RTEMS. For this purpose a `forbrun()` function has been added where subsequent initialization of components is realized.

Error handling. Although a small, yet an important difference or an incompatibility was discovered in error handling. While in Linux the header file `error.h` has to be included and the actual error messages are being passed by calling `error()` function, in RTEMS the include is `rtems/error.h` and `rtems_error()` is to be called. This difference in function names was masked as `ERR()`[3], which is independent on underlying platform.

[2] forb.h
[3] forb.h

Chapter 4

RTEMS Operating System

This chapter just briefly summarizes why RTEMS operating systems has been chosen for use of FRSH/FORB. The Section 4.1 shows its advantages. Comments on application programming interface of RTEMS are made in Section 4.2 and description of internals with main focus on scheduling in Section 4.3.

4.1 RTEMS advantages

The RTEMS Operating system [rte, 2010] is a very good candidate to became an underlying platform for the FRSH/FORB project. It is a *hard real-time* operating system for embedded purposes which overlaps the same area of interest as of FRSH/FORB, and it is open-source. The open property gives us the opportunity to make minor changes in the internals of the operating system or at least to study them and thus to make a better fit for the middleware.

Portability. A very strong side of RTEMS may be considered a very good portability across hardware platforms and the isolation of RTEMS core from hardware specific packages. This plays a very important role in terms of possibly being accepted by a wide audience in the real-time and embedded area.

The operating system is written in two languages Ada and C. We will deal with C only all the time.

4.2 API

RTEMS basically includes two application programming interfaces. First one is called Classic API which better corresponds to feature set of the system, and the second one, POSIX API which is a latter effort of RTEMS towards having a common API with other operating systems in order to support portability of applications. However, implementation of the POSIX standards is still under development.

4.3 Internals

The internal structure of RTEMS includes basically all necessary features for mature real-time applications. The implementation is even quite readable and easy to use. The Figure 4.1 displays and overview of RTEMS core features.

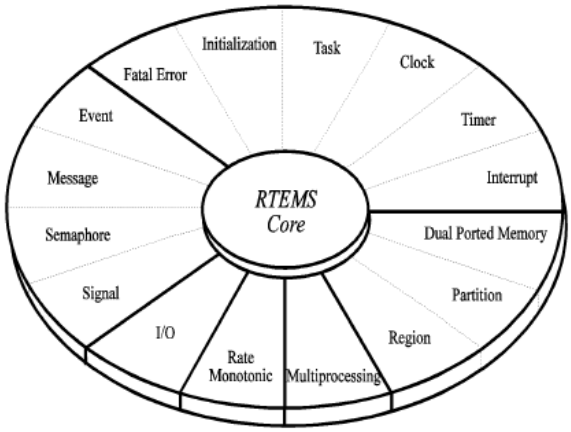


Figure 4.1: Structure of RTEMS internals [OAR, 2009].

4.3.1 RTEMS Pluggable Scheduler infrastructure

The implicit and only scheduling policy implemented in RTEMS is a priority scheduling for tasks with priorities between 0 to 255. However, beginning the version 4.11 of RTEMS, there is a brand-new feature called Pluggable Scheduler and as the name already insists the feature enables to pull a random scheduler out of the RTEMS kernel which enables application programmers to arbitrarily choose and implement a scheduling policy they need for their applications. Moreover, it is very suitable for experiments with scheduling

which is of our major interest as well.

Implementation of a pluggable scheduler does not mean a complete implementation of thread handling, but rather just a couple of callback functions along with scheduler specific memory requirements have to be defined whereas the rest of general thread handling is still managed inside of the core. However, in order to be capable of implementing a pluggable scheduler, an user has to be aware and closely acquainted with the RTEMS internals. How far the pluggable scheduler infrastructure is useful for various types of schedulers is not clear and the RTEMS developers are open to any feedback and experiences with issues encountered while using the pluggable scheduler infrastructure in order to adapt the interface so that it can be used for as wide a span of scheduling algorithms as possible.

Chapter 5

Earliest Deadline First scheduler

Every system that runs multiple tasks on a limited number of processors is required to have a certain set of rules deciding how the tasks are assigned processor in order to fulfill some criteria which may differ depending on the purpose. This set of rules is called *scheduling policy* of *scheduling algorithm*.

The reason why I deal with scheduling in this thesis is the need to include the RTEMS CPU as a resource. The CPU resource is of the main importance because every application utilizes CPU.

This chapter first introduces basic concepts of real-time theory, scheduling algorithms and issues related to scheduling in Section 5.1.

Later, in Section 5.2 the basic concepts are commented form the point of view of FRSH/FORB and a design of low-level scheduler along with some specific features for our purposes will be performed.

However, for sake of use in FRSH/FORB this scheduling policy is not sufficient and a higher-level scheduler that provides the possibility to RTEMS CPU act as a reservable resource will be introduced in Chapter 6.

5.1 General concepts

5.1.1 Mathematical model of a real-time system

Every task τ_i according to the real-time theory [Liu, 2000] has several parameters such as

- Release time r_i - time when the task becomes ready
- Start time s_i - time when the task starts its execution
- Computation time C_i - period necessary for the task execution (without any interruption)
- Deadline d_i - time by which the task has to be finished
- Finishing time f_i - time when the task finishes its execution

as depicted on the Figure 5.1.

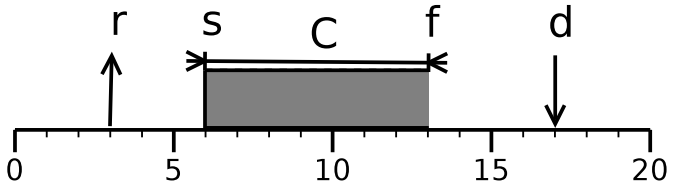


Figure 5.1: General model of a real-time task.

Task states. Each task has a certain state. The main states usually considered in RTOS are *running*, the task is currently being executed on a processor, *ready*, the task is ready to run and waiting in a *ready queue* for its execution. If a task is said to be *suspended*, it means the task is removed from the ready queue and is waiting for some event (mutex, timer, ...) to become ready.

Ready queue. Ready queue is a representation of an entity containing set of ready tasks in an order corresponding to their *priority* How to assign task's priorities is a matter of used scheduling algorithm.

We can separate the task into two groups *periodic* and *aperiodic*. Periodic tasks are of the main concern in the real-time applications, thus a special attention has to be payed to them. A periodic task τ_i is a sequence

of infinite number of its instances called *jobs* (J_{ij}) with the above mentioned properties. The index i denotes a task belonging to and j denotes a number of task instance τ_i.

Moreover, a periodic task is characterized by inter-arrival time also called *period* (T_i) which is a period between two subsequent activations of task $T_i = r_{ij} - r_{ij-1}$. For sake of simplicity the periods are equal to deadlines.

Schedule. If we consider a set of n tasks $\tau_1 \dots \tau_n$ competing for the processor time, the order of their job executions is called a *schedule*.

Scheduling algorithm. *Scheduling algorithm* is an algorithm that decides which task of a ready queue can gain processor and orders the ready tasks into a ready queue.

5.1.2 Schedulability analysis

A *schedulability analysis* refers to the procedure which determines whether a *feasible* schedule can be found. Feasible schedule is such a schedule that makes all tasks meet their deadlines. Calculation complexity of such analysis depends on a lot of things, number of tasks, complexity of their mathematical model as well as additional given constraints such as access to mutually exclusive resources.

Depending on complexity and application the analysis may be performed either

- on-line (while the tasks already run)
- or off-line (given all the parameters and constraints are available a-priory).

On-line schedulability analysis. The on-line case of schedulability analysis is needed in case we do not know the set of task or exact requirements before the application starts. That is a case of modular applications. Since the complexity of schedulability analysis grows rapidly with respect to complexity of assumed task model, it is necessary to keep the model as simple as possible in order to be capable of applying the schedulability analysis on-line.

Utilization. The schedulability analysis may be based on various information. However, the easiest option is called *utilization-based* schedulability analysis. For each periodic task τ_i consider utilization of

$$U_i = C_i/T_i$$

corresponding to a fraction of processor utilized by τ_i. And for set of n tasks *total utilization* of processor

$$U = \sum_{i=0}^{n} U_i.$$

can be calculated. Regardless of what scheduling algorithm we use, a necessary condition is in any case expressed by the following theorem.

Theorem 1. *[Abeni and Buttazzo, 1998] A system of independent, preemptable periodic tasks with relative deadlines longer than their periods can be feasibly scheduled on a processor as long as the total utilization is equal to or less than 1.*

5.1.3 Classification of scheduling algorithms

There are several classes of scheduling algorithms which might be divided according to the Figure 5.2.

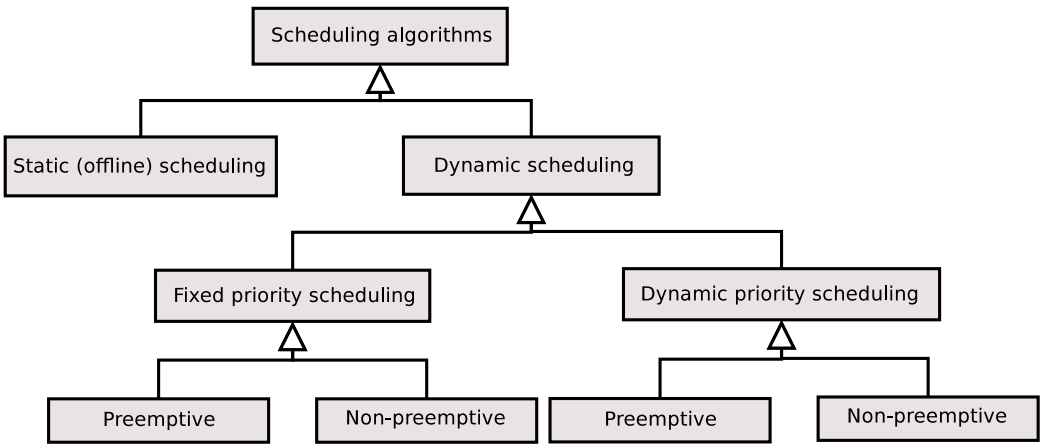


Figure 5.2: General division of scheduling algorithms.

The static scheduling algorithms are fully calculated off-line whereas the dynamic scheduling algorithms decide about the order of tasks on-line. The

off-line scheduling algorithms are not useful for dynamic applications. The dynamic schedule algorithms will be studied in more detail.

Preemptive algorithms unlike non-preemptive ones are allowed to interrupt a running task before its execution finishes.

Widely used scheduling algorithms

In real-time systems there are a few widely used scheduling algorithms such as *Rate-Monotonic* (RM), *Deadline-Monotonic* (DM) and *Earliest Deadline First* (EDF) [Liu, 2000].

The Rate-Monotonic algorithm belongs to the group of Fixed Priority Scheduling (FPS) algorithms. The priorities are assigned corresponding to rates based on assumption that a task with higher rate (shorter period) deserves higher priority. The Deadline-Monotonic algorithm also belongs to the FPS and assigns priorities according to relative deadlines. The shorter deadline the higher priority.

The Earliest Deadline First algorithm belongs to the group of dynamic priority algorithms where the priorities correspond to absolute deadlines of the tasks and thus the priorities have to be updated with each release of a job. Unlike FPS it leads to an unlimited number of priority levels.

5.1.4 Shared resources

The above mentioned model of real-time tasks assumes that the tasks are independent. however, several tasks may share common resources with exclusive access which makes them not independent. These resources have to be accessed only by one thread at a time in order to maintain consistency of the resource data. Therefor, it can happen that a task (thread) has to wait for the resource to become available. The means to ensure an exclusive access are called *mutexes* commonly implemented in RTOS which are used in order to lock or unlock the resource for its use. Consequently, another task demanding to use the same locked resource is blocked until the resource is available again.

Priority inversion. The problem arising as a consequence of the task blocking due to waiting for a shared resource may result in an effect called *Priority Inversion* in case a task is blocked by another task with a lower priority. A sample of Priority Inversion causing a deadline miss is shown on the Figure 5.3.

Deadlock. In case of tasks blocking each other, another problem might emerge. If a situation where tasks are waiting for each other in a loop occurs, a deadlock may cause halt of the set of task.

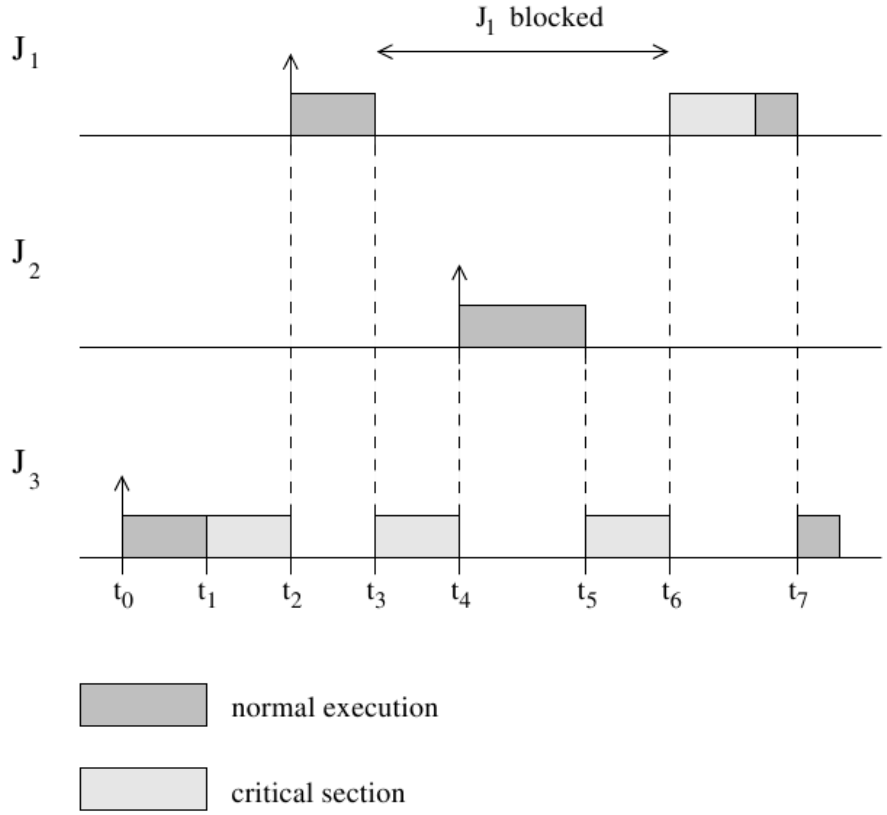


Figure 5.3: A simple example of priority inversion (picture taken from [Molnár, 2006]) Job J_1 is blocked by a job J_3 with lower priority.

In order to solve this problem various rules (protocols) have been proposed. Some of them such as *Priority Inheritance Protocol* (PIP) and *Priority Ceiling Protocol* (PCP) are designed for sake of fixed priority systems and some other ones e.g. *Preemption Ceiling Protocol* (PreCP) for dynamic

priority systems. Another possibility to avoid long blocking is a protocol called *Nonpreemptive Critical Sections* (NPCS). There is a list with a brief description of various protocols that may be used with the EDF algorithm along with both their advantages and disadvantages.

Priority Inheritance Protocol

This is a simple protocol that works with any priority-driven algorithms (thus even EDF). It does not prevent deadlocks unless another protocol doing it is employed simultaneously. Each job is assigned a priority by a scheduling algorithm. This priority may be raised to a higher priority of blocked job. The priority is *inherited* and the job continues its execution with this inherited priority as long as it causes the blocking. The PIP is defined by set of rules [Liu, 2000]:

- 1. Scheduling Rule: Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities. At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3.

- 2. Allocation Rule: When a job J requests a resource R at time t,
  (a) if R is free, R is allocated to J until J releases the resource, and
  (b) if R is not free, the request is denied and J is blocked.

- 3. Priority-Inheritance Rule: When the requesting job J becomes blocked, the job J_l which blocks J_h inherits the current priority $\pi(t)$ of J_h . The job J_l executes at its inherited priority $\pi(t)$ until it releases R; at that time, the priority of J_l returns to its priority $\pi_l(t)$ at the time t when it acquires the resource R.

Priority Ceiling Protocol

As well as the PIP, this protocol is based on the idea that a job J_{low} blocking another job J_{high} of a higher priority is supposed to inherit its priority in order to decrease the blocking period of J_{high}. Moreover, unlike the PIP, PCP is capable of avoiding deadlocks because a priority ceiling is set. The drawback of this protocol is that it is pretty difficult to implement in a system with dynamical priorities of tasks. As a matter of fact, it is possible to use this protocol in case of Job-Level Static-Priority systems such as EDF where the the jobs in a ready queue do not change their priorities with respect to each other but the implementation requires updating the priority ceilings upon release of every new job.

Nonpreemptive Critical Sections

This is a very simple protocol avoiding deadlocks because no preemption during critical sections can occur. This protocol is very useful especially when the critical sections are short. However, of course, not allowing to preempt longer critical sections may result in long waiting periods of the higher priority jobs.

5.1.5 Multiprocessing

So far any *Symmetrical Multiprocessing* (SMP) support is not considered, multiprocessor and multi-core systems [Brandenburg et al., 2000] exceed the scope of this thesis. However, a practical use of SMP would be definitely found and thus, possibly implemented in the future.

5.2 Design of a scheduler for the resource reservation framework

This section describes how the low-level scheduler for RTEMS has been designed and what features it includes.

Before we dive into any details, let me introduce the crucial expectations of the scheduler we select and what are the constraints in order to keep this in mind during reading of following sections. The main reasons why I have to make decisions about how the scheduler for FRSH/FORB will look like are

- possibility of CPU time reservation (in the ideal case 100% of it),
- independence of tasks,
- implementation for RTEMS, in particular use of the Pluggable scheduler infrastructure (see Section 4.3.1).

5.2.1 Comparison and selection

A very important parameter playing a major role in the scheduling algorithm selection is *schedulable utilization* U_{alg} which indicates the maximum possible total utilization that a particular algorithm can schedule. Therefor, the condition $U < U_{alg}$ guarantees the algorithm to create a feasible schedule.

As for the RM algorithm, the schedulable utilization is

$$U_{RM}(n) = n(2^{1/n} - 1) \text{ converging to } \lim_{n \to +\infty} \simeq 0.69$$

and it is said to be optimal among all FPS.

Since the EDF schedule has the possibility to dynamically change the deadlines it necessary has to yield a better performance.

Theorem 2. *[Liu, 2000] The schedulable utilization $U_{EDF}(n)$ of the EDF algorithm for n independent, preemptable periodic tasks with relative deadlines equal to or larger than their periods is equal to 1.*

As a matter of fact, the EDF algorithm does not have only the advantage over FPS in terms of schedulable utilization and thus possibility to employ 100% of processor time which is very useful for the resource reservation as stated above, but also the switching overhead is lower in terms of context switch counts [Abeni and Buttazzo, 1998, Molnár, 2006]. Therefor, the EDF scheduler is the best choice as the low-level scheduler for reservation-based scheduling (see Chapter 6).

5.2.2 Priority inversion handling

However, all this considerations have been made only upon the assumption of independent tasks executing without blocking periods. It is necessary to decide how to handle the priority inversion in order to avoid unnecessarily long waiting periods. The PIP was selected for the EDF scheduler because it was found very easy to use on RTEMS, where the it is already implemented for static priority scheduling. The protocol works almost the same way in case of dynamic and static priorities. On the other hand, PCP would be a significant challenge and thus has not been chosen. As a matter of fact, we can not avoid deadlocks properly at the moment, therefor the applications should be cautious in this matter.

In case we wanted to schedule complicated sets of mutually dependent tasks that tend to deadlock, the NPCS protocol might be easily implemented. However, it is not considered as of much importance now.

5.2.3 Background tasks inclusion

So far in this chapter we were examining only the situation of deadline-driven tasks with real-time properties and a discussion has been made which algo-

rithms fit the most to the previously stated requirements of the framework. However, in some cases it is desirable to run tasks without real-time properties on the same processor. Since such tasks do not have any requirements on deadlines, it is sufficient when they are allowed to enter processor only in the otherwise idle periods of processor. Tasks having no deadlines barely fit to the concept of dynamic priority scheduling. The initial idea would be to assign a background task and infinite or a very long deadline. This is sufficient in case of FPS scheduling algorithms such as RM where the the background tasks would be simply assigned a low priority. In case of dynamic priority scheduling the problem is that priorities move in the course of time and increase their values. It is required to have an explicit priority comparison in case of deadlines since normal algebraic comparison does not work for a finite-bit variable priority representation where the priorities overflow.

There are a few possible ways how to incorporate the background task into the deadline-based dynamic priority scheduling. Now, an analysis on possible approaches will be introduced.

Single ready queue

The background jobs are assigned a very far deadline d_f, which is higher than all deadlines $d_{realmax}$ of real-time tasks, while shifting the d_f forward in time to maintain the condition $d_f > d_{realmax}$ which is not practical. Moreover, note that this is actually a case of job-level dynamic-priority scheduling and any conclusions coming out of the EDF theory have to be reconsidered. Especially PIP does not work here.

Double ready queue

It is possible to use additional (lower-priority) ready queue. Thus there are two parallel ready queues, first one containing finite priority (deadline) jobs and second one containing infinite priority jobs scheduled for example in a RR manner. The practical drawback is that an infinite deadline has to be represented by either a finite number or a flag (separate information in another variable). For a better distinction of the two situations we have to highlight that the essential difference is whether to indicate that the task belongs to the set of background ones inside or outside of the priority variable.

- **Additional variable denoting the background property.** In this case you may make up several ways how to represent the background property of a task even with some additional priority levels. However, a common drawback is that the basic PIP is not applicable and you

have to implement special rules saying how to handle the additional properties. As a result, the inheritance protocol would have to be also called differently because the tasks do not inherit only priorities but also the additional properties of higher-priority task.

- **Hybrid priority space - background property determined by a priority level.** This approach might be called a *Hybrid* priority representation. To indicate that a set of tasks belongs to a specific scheduling policy, we reserve a part of the *priority space* (full set of priority levels) for one algorithm and a part for another algorithm. By specifying rules for priority comparison we declare priorities of that regions which is all we have to do about it. Remember that a specific priority comparison have to be created in any case. The PIP works fine because the job-level priorities are static and properly comparable, moreover, the only property of jobs to be inherited is the priority. Thus the jobs may arbitrarily skip across all the priority regions and back as they are blocking and unblocking other jobs.

Obviously, the hybrid approach has been selected for the RTEMS EDF scheduler.

5.2.4 Design of a ready queue

This section introduces a step by step design procedure of the EDF scheduler ready queue specifically focused on the priority interpretation allowing to schedule deadline-driven and background tasks simultaneously.

Imagine a priority space represented by an n-bit variable. The Figure 5.4 shows how the priority representation is being incrementally transformed (T stands for current time, D1 and D2 stand for deadlines of deadline-driven tasks, P1 represents a priority driven task). Since the priorities represent deadlines (time instants), they are increasing as well as current time and overflowing after the highest possible value of the variable. So both the current time and deadlines rotate within the variable.

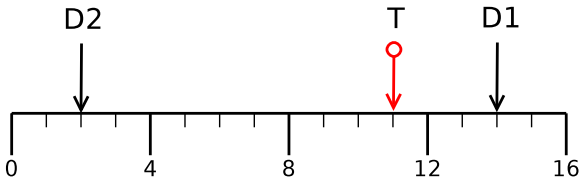
Now, imagine the same situation but with having the most significant n-th bit (MSB) equal to zero. This we can reach by looking at the variable through a mask (this was inspired by [Kim and Shin, 1997]). The current time along with deadlines are limited only to the region where MSB is equal to zero which is the lower half of variable. The upper part is never reached by deadlines and the current time, the time circulates only in the lower part

and the upper part stays forever untouched.

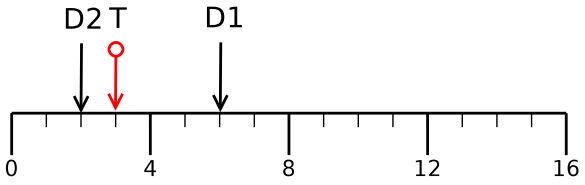
The upper part of priority space will be used by background tasks. Each background task is assigned a priority in the same fashion as in case of normal priority-based scheduling and these priorities are mapped into the upper part of priority space. Now we can see that we reached the desired effect where the deadline-driven priorities never cross the background priorities as they are perfectly separated.

Consequently, in this set-up a proper priority comparison method is to transform the priority space so that a simple numeric comparison can be performed in order to figure out which tasks have higher priority than other ones. This is rather simple, since only one step is needed. All tasks in the lower part of priority space are to be shifted to the left by a value corresponding to current time. Underflowing deadlines show up in the middle of the priority space. After this procedure we end up with the priority space perfectly sorted in the order they are allowed to become scheduled.

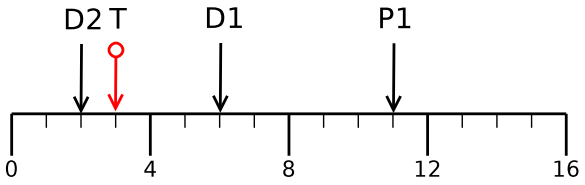
The Figure 5.4 illustrates the procedure of priority comparison for the deadline-driven and background tasks residing in a single ready queue.



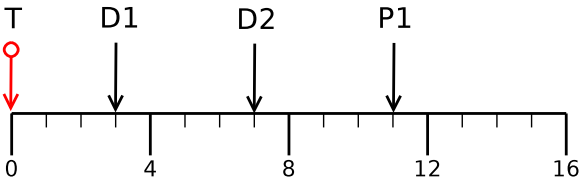
(a) Only deadline-driven tasks are included in the ready queue first.



(b) In order to use only the left hand half of the priority space, the most significant bit of current time and all deadlines is set to zero. This results in empty right hand side of the priority space.



(c) Background task (low priority) added to the right hand side of the priority space, which means that the priority is lower than priority of all deadline-driven tasks.



(d) Left hand half of the priority space shifted (rotated) to the left by the value of current time in order to set the correct order of priority levels (D1, D2, P1 as last). Now, the priority comparison is performed by a simple numeric comparison.

Figure 5.4: Background tasks' inclusion into a deadline-driven ready queue.

Chapter 6

RTEMS Constant Bandwidth Server for resource reservation

Ordinary scheduling algorithms as shown in Chapter 5 are not sufficient in terms of providing a reserved resource. This is guaranteed only using *Reservation-based schedulers* [Palopoli et al., 2008]. The major phenomenon in the are of reservation-based algorithms [Abeni et al., 2005] is temporal isolation.

This chapter basically deals with a strategy how to reach the temporal isolation of tasks. This is in more detail described in Section 6.1. Result of this chapter is a high-level scheduler making use of the low-level EDF as presented in Chapter 5.

Basic building blocks for temporal isolation are time servers. A brief overview of them as well as selection of the most appropriate of them is introduced Section 6.2.

There is, however, a set of specific rules that the time server has to adopt in order to become fully capable of temporal isolation of tasks. It has been searched for these rules in related projects and an overall design of a scheduling policy that temporally isolates tasks has been made up.

Scheduling parameters. Let us consider a very simplified model of periodic task behavior, which is defined by period P equal to deadline and by computation time Q or *budget* defined per one period of a periodic task. These attributes are denoted *scheduling parameters*.

Hard and Soft reservations. In theory, there is more options how to create a reservation on a resource. First option is a *hard reservation* which means the budget negotiated by a task is guaranteed but can not be exceeded even if there is a spare resource available and it might happen that the processor goes idle even in case of ready tasks. On the other hand a *soft reservation* strategy refers only to minimal guarantees. The tasks may be given more budget in case they do not oppress other tasks.

It was decided that the hard reservation strategy is sufficient and the following sections will deal only with the case of hard reservations.

6.1 Temporal isolation property achievement

In order to ensure desired and proper behavior of a task, resources necessary for the task execution have to be provided as mentioned in Section 2.1. Concerning active resources, the meaning of the actual resource is *execution time*. The tasks compete for the execution time on a processor (or multiple processors). The processor has a limited computation power, thus an algorithm distributing the time is required. The maximum computation power is dependent on a given algorithm (as shown in Section 5.2.1, EDF is optimal in this matter). However, not only a sufficient amount of resources for all tasks have to be guaranteed, but also their proper distribution which basically means that a task is not allowed to take more resources than a certain amount. In case of hard reservation it is the budget Q.

As soon as the CPU time is properly distributed, it is crucial to maintain the tasks within the assigned bounds. In case one task would try to spend more time than agreed, it would affect either a quality of other tasks or even cause a deadline miss of another task. Thereafter a definition of temporal isolation arises as

Definition 1. *A task is said to be temporally isolated when its quality of execution in terms of meeting all its deadlines and having a certain budget each period is independent of any other tasks.*

In ideal case we can imagine the temporally isolated tasks as if they were running on their own slower dedicated processor [Palopoli et al., 2008]. Practically it means that we have to count on all possible cases of task's execution which might negatively affect another ones.

6.2 Time servers

There are various scheduling approaches as describe in Chapter 5 that are responsible for selecting tasks gaining processor. There are also extensions of these algorithms that have a property of bandwidth limitation. These algorithms are called *time servers* and provide with various approaches of budget handling features being of our great interest.

The time servers may be used for various purposes. Some servers may be designed in such a way that one server include multiple tasks and handles their budgets together. This is not our case because the need for a time server comes from a limitation of a single task.

Thereby, the task and its server may be considered a single entity since the scheduling parameters of a task and a its time server are merged together and we can consider it one entity. This is a simplifying assumption which may be extended into a group of tasks being facilitated by a single server, but we abandon this direction as it is not necessary to have implemented at the moment.

Bandwidth limitation. *Bandwidth* (or an utilization) of a periodic task refers to average budget per period. As we agreed, each task is required to be assigned to a server in order to have a limited bandwidth. The limited bandwidth is a necessary but not sufficient condition to provide the temporal isolation. Other conditions will be presented later (Section 6.4). However, of a practical use may be also a presence of non-real time task that do not claim any deadlines. These tasks may be considered *background tasks* (see Section 5.2.3) and are assigned a very low priority (or infinite deadline in case of deadline-driven scheduling). Because of the low priority they do not occupy the processor unless it becomes idle and they can not interfere with any other tasks, therefor, it is assumed that they do not require any server.

There exists a numerous span of time servers [Liu, 2000]. We are interested in a group of servers called *bandwidth preserving* servers. These servers are characterized by consumption and replenishment rules that specify how the budget is handled in the course of time. The consumption rule specifies when and how the budget is consumed as related tasks are executed. The replenishment rule says when and how the budget is refilled. According to [Abeni and Buttazzo, 1998], the best server having the temporal isolation property is *Constant Bandwidth Server* (CBS). In case of the AQuoSA

project (Section 6.3) a CBS has been selected.

The consumption and replenishment rules of CBS are very simple.

- C: The budget is consumed only when the task is executed.
- R: The budget is replenished to the maximum (negotiated) value in every beginning of a new period.

In the example of [Abeni and Buttazzo, 1998] there is a situation of coexistence of hard and soft tasks proposed where only the soft tasks are served on the CBS basis under assumption that the hard tasks can not violate their properties and thus do not have any assigned budget. However, in order to really ensure the temporal isolation of all tasks in case they are all considered hard, it is necessary to assign a time server to each of them. In general a server may embed multiple tasks, in which case the temporal isolation is not provided between tasks sharing a common server. This might be useful in some applications but can also be remodeled into the form of single task servers.

In the Section 2.1 it was described how tasks can negotiate and obtain resources in FRSH/FORB. FRSH handles processor time as one of shared resources provided by underlying system. Now, the resource is considered processor time and each task (thread) can ask for a specific budget of time. A negotiation procedure between the application and FCB by means of contract is then performed, the exact negotiation procedure is described in [Sojka, 2010]. If the negotiation ends up successfully, the respective task is guaranteed to have the possibility to run with a given bandwidth as long as it wants. From the perspective of FRSH it is necessary to establish appropriate scheduling and time reservation method which is the aim of this chapter.

As the problem of temporal isolation was already researched, we can make use of some hints and steps. The related research was performed under name of AQuoSA project (see Section 6.3) [Palopoli et al., 2008].

6.3 Approach of AQuoSA project

Since this work concerns porting onto RTEMS, similar issue has already been solved and researched before. The Linux platform uses AQuoSA (*Adaptive Quality of Service Architecture*) project [aqu, 2008] as the mean of time distribution in Linux kernel. This architecture embeds a reservation based

process scheduler for soft real-time tasks which is capable of dynamical CPU time allocation for QoS aware applications such as media streams [Abeni et al., 2005].

So, this project stands as example how the RTEMS version of CPU time distribution might have been provided. However, this project is too extensive in order to be fully brought to RTEMS yet. It might be done in the future. On the other hand, the basic design concept was taken over for RTEMS CBS.

6.4 Rules ensuring a temporal isolation of tasks

This section introduces the set of main and essential features that a CBS scheduler has to include so that the condition of temporal isolation of all tasks is provided. Therefor, the Constant Bandwidth Server including these features yields a complete design of the resulting scheduler

6.4.1 Budget overrun

In case a task exceeds its reserved budget during a period, it has to be suspended until the budget gets replenished again, which is the beginning of a new period. This is called *budget overrun*. Tasks which for some reason exceed their budget may affect schedulability of other tasks in case the total utilization exceeds maximum schedulable utilization. That is not acceptable with regard to the temporal isolation definition. On the other hand, if neither of the jobs overruns, the CBS behaves as a plain EDF (in case the low-level scheduling algorithm is EDF) which yields the lemma.

Lemma 1. *[Abeni and Buttazzo, 1998] A hard task τ_i with parameters $(C_i,\ T_i)$ is schedulable by a CBS with parameters $C_i \leq Q_i$ and $T_i = T_s$ if and only if τ_i is schedulable with EDF.*

A disputable issue is how to asses and handle tasks exceeding their budgets. A lot of various cases of this effect may occur, tight execution time estimate, infinite cycle or lack of budget awareness on the side of task and so on. For the scheduler it is almost impossible to identify the cause and thus to decide what to do with the job. This responsibility lies on the scheduler designer. The least controversial solution is to suspend the task until the end of the period as soon as the budget reaches zero and then enable again. However, of a rather practical concern is problem of execution consistency, when the task is enabled after a forced suspension, the execution most probably starts in the middle of some procedure and the expected periodicity is

not maintained any more. Therefor, another possibility is to just suspend the task forever.

6.4.2 Unblock rule

In order to extend the explanation of properties of the temporal isolation, it is necessary to say that not only the budget overrun must not occur, but also a blocking periods have to be taken into account. These do not follow the classical periodic activation pattern and the utilization based schedulability analysis is not sufficient any more. There has to be another rule created in order to maintain real-time properties of system. However, note, that this behavior should be considered a missed deadline of the respective task.

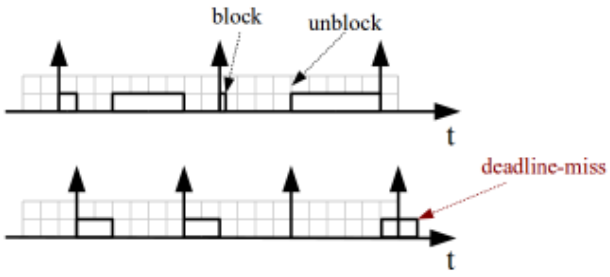
The presence of blocking periods during the task execution may occur. Unless the blocking period is limited and counted as a part of task execution, which decreases the schedulable utilization because we force the processor to become idle, a deadline miss may occur for one of the following reasons. Either the blocking period is too long so that it results in an immediate deadline miss of the current task or the task gets unblocked too close before deadline. This may cause a deadline miss of a consequent task as it is presented in [Cucinotta and Checconi, 2010]. In both cases there is no correct solution adhering the current model (Section 5.1.1), the system is not schedulable and the given set of tasks is to be handled in such a way that real-time capabilities of no task but the violating one are affected (Figure 6.1).

In the article [Cucinotta and Checconi, 2010] the situation of a deadline miss due to unexpected blocking period is described. In case a running task is blocked and becomes runnable again too close to its current deadline without changing the absolute deadline and thus is the most eminent one to be run. In order to avoid this effect, the following decision has to be made. If the remaining computation time C_{ijrest} divided by the time left until the current deadline d_{ijrest} exceeds the assumed utilization of the task U_i

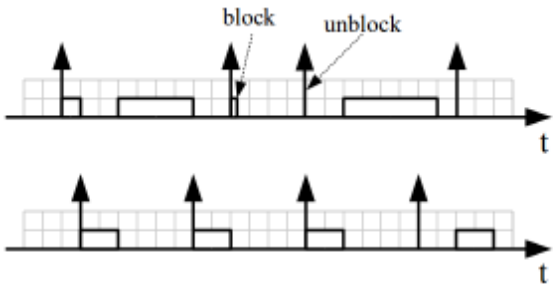
$$\frac{C_{ijrest}}{d_{ijrest}} > \frac{C_i}{T_i}, \tag{6.1}$$

the current deadline d_{ij} is to be reset to $T_{current} + T_i$.

Lemma 2. *If a task is unblocked and the condition 6.1 is satisfied, then the deadline is reset to the current time plus the reservation period, and the current budget to the allocated reservation budget.*



(a) Deadline miss of second task due to a long blocking period of first task.



(b) Deadline postponed so that no deadline miss of the other task occurs.

Figure 6.1: Unblock rule [Cucinotta and Checconi, 2010].

6.4.3 Bandwidth inheritance

Another protocol [Lipari et al., 2004] to be included is *Bandwidth Inheritance* (BWI). This protocol is a straightforward extension of priority inheritance protocol for reservation-based scheduling. Tasks sharing mutually exclusive resources can be blocked for a certain time before acquiring the resource owned by another task. The contribution of BWI says that not only a priority of the blocking task should be inherited, but also the budget to consume from. This is another necessary condition of task isolation.

In case of single processor systems, the BWI rules are quite straightforward [Lipari et al., 2004]. The example how the protocol was implemented into Linux kernel during AQuoSA project is described in [Faggioli et al., 2008]. A more complicated extension to multicore systems and symmetrical multiprocessing is denoted BWI-M and described in [Faggioli et al., 2010].

Chapter 7

Implementation of scheduler

This section describes and implementations of EDF/CBS according to the design from Chapter 5 and Chapter 6. The first question was raised when a possibility to implement these two approaches independently was discussed. Since the EDF is a low-level scheduling algorithm being used by a higher level reservation-based CBS policy, it was assumed these two entities would be implemented separately. However, it was found out that either serious implementation complications would emerge or execution overhead would increase. The CBS requires some rules directly residing in EDF algorithm which means a pure EDF as a separate entity is not our aim.

First, the way how a Pluggable scheduler interface in order to connect a scheduler to RTEMS will be described in Section 7.1. Consequently, the scheduling algorithm implementation itself will be described in Section 7.2. The Section 7.3 presents how an application programming interface for the CBS scheduler was created and Section 7.4 deals with incorporating the RTEMS CBS scheduler into the framework as a resource.

7.1 Pluggable Scheduler interface description

Since any documentation on RTEMS pluggable scheduler infrastructure is not published anywhere yet (but definitely will be soon) except for [OAR, 2010], I will introduce the pluggable scheduler interface with some comments to the single functions and steps necessary to reach a working scheduler. Moreover, I will try to point out some problems I encountered and problems that may be of a general concern.

First of all, an user has to specify that he/she wants to connect an user

scheduler instead of a built-in one and then specify memory requirements per thread and per scheduler (e.g. ready queue).

```
#define CONFIGURE_SCHEDULER_USER
#define CONFIGURE_SCHEDULER_USER_ENTRY_POINTS
#define CONFIGURE_MEMORY_FOR_SCHEDULER
  (_Configure_From_workspace(sizeof(EDF_Chain_Control)))
#define CONFIGURE_MEMORY_PER_TASK_FOR_SCHEDULER
  (_Configure_From_workspace(sizeof(RBT_Node)))
```

Consequently, the `_Scheduler` structure has to be defined by means of call out functions that are being called after important actions occur in the RTEMS core.

```
 #define SCHEDULER_ENTRY_POINTS \
{ \
_Scheduler_edf_Initialize,    /* initialize entry point */ \
_Scheduler_edf_Schedule,      /* schedule entry point */ \
_Scheduler_edf_Yield,         /* yield entry point */ \
_Scheduler_edf_Block,         /* block entry point */ \
_Scheduler_edf_Unblock,       /* unblock entry point */ \
_Scheduler_edf_Allocate,      /* allocate entry point */ \
_Scheduler_edf_Free,          /* free entry point */ \
_Scheduler_edf_Update,        /* update entry point */ \
_Scheduler_edf_Enqueue,       /* enqueue entry point */ \
_Scheduler_edf_Enqueue_first, /* enqueue_first entry point */ \
_Scheduler_edf_Extract,        /* extract entry point */ \
_Scheduler_edf_Priority_compare /* compares two priorities */ \
}
```

Although this does not exactly correspond to the guide [OAR, 2010], it is how it works. If you want to make sure that the current implementation of RTEMS Core is the same as I am describing, read `confdefs.h` to figure it out. That file is responsible for all RTEMS configuration.

7.2 Scheduling implementation

Now, I will introduce the implementation of CBS/EDF scheduler. The full theoretical background to the scheduling concept was described in Section 5.2.4.

Aim of arbitrary scheduler is to determine a *heir*, that is a thread which is to be executed as next on a CPU. In RTEMS, this is represented by a `Per_CPU_Control` structure[1].

```
 /**
 *  @brief Per CPU Core Structure
 *
 *  This structure is used to hold per core state information.
 */
typedef struct {
#if (CPU_ALLOCATE_INTERRUPT_STACK == TRUE) || \
    (CPU_HAS_SOFTWARE_INTERRUPT_STACK == TRUE)
  void  *interrupt_stack_low;
  void  *interrupt_stack_high;
#endif
  uint32_t isr_nest_level;
  Thread_Control *executing;
  Thread_Control *heir;
  Thread_Control *idle;
  volatile bool dispatch_necessary;

} Per_CPU_Control;
```

The `executing` thread is the one currently running on a specific CPU, the `heir` is the one to switch to as soon as the `executing` one leaves the CPU. The `idle` thread is a special thread with the lowest possible priority running every time the processor does not have anything else to do. We say the processor is *idle*.

7.2.1 Thread Control Block

Each thread in the RTEMS operating system is represented by a structure called *Thread Control Block* (TCB). This structure[2] maintains all necessary information about threads and states of threads. The most important properties are for instance `current_state` saying whether the thread is runnable or not (and why), `current_priority` obviously indicating the priority of a thread, which is of the main significance with respect to scheduling. Another attributes worth noticing are e.g. `is_preemptible` preemptibility indicator

[1] percpu.h
[2] Located in thread.h

of a thread or `scheduler_info` a generic pointer to arbitrary additional information necessary for a specific schedulers. Another important attributes will be presented in the course of time as they gain significance.

```
/**
 *  This structure defines the Thread Control Block (TCB).
 */
struct Thread_Control_struct {
  Objects_Control          Object;
  States_Control           current_state;
  Priority_Control         current_priority;
  Priority_Control         real_priority;
  uint32_t                 resource_count;
  Thread_Wait_information  Wait;
  Watchdog_Control         Timer;
#if defined(RTEMS_MULTIPROCESSING)
  MP_packet_Prefix        *receive_packet;
#endif
#ifdef __RTEMS_STRICT_ORDER_MUTEX__
  Chain_Control            lock_mutex;
#endif
     /*================= end of common block ================*/
  uint32_t                                suspend_count;
#if defined(RTEMS_MULTIPROCESSING)
  bool                                    is_global;
#endif
  bool                                    is_preemptible;
#if __RTEMS_ADA__
  void                                   *rtems_ada_self;
#endif
  uint32_t                                cpu_time_budget;
  Thread_CPU_budget_algorithms            budget_algorithm;
  Thread_CPU_budget_algorithm_callout     budget_callout;
  Thread_CPU_usage_t                      cpu_time_used;
  void                                   *scheduler_info;
  Thread_Start_information                Start;
  Context_Control                         Registers;
#if ( CPU_HARDWARE_FP == TRUE ) || ( CPU_SOFTWARE_FP == TRUE )
  Context_Control_fp                     *fp_context;
#endif
  struct _reent                          *libc_reent;
```

```
  void                        *API_Extensions[ THREAD_API_LAST + 1 ];
  void                                 **extensions;
  rtems_task_variable_t                 *task_variables;
};
```

For sake of the EDF scheduling the variable part of TCB, `scheduler_info`, was created as another structure containing the EDF specific information.

```
typedef struct RBT_node_struct {
     Deadline_Control abs_deadline;
     Deadline_Control rel_deadline;
     EDF_Node  *left;
     EDF_Node  *right;
     EDF_Node  *parent;
     Node_Color   color;
     EDF_Chain_Control *ready_chain;
     uint8_t is_enqueued;
     uint32_t cmp_time;
     rtems_id timer_id;
     uint8_t flags;
} RBT_Node;
```

The pointers to `EDF_Node` structures which are actually the TCBs are necessary for handling a position in the ready queue (Red-Black tree). The main scheduling parameter, `rel_deadline` is the deadline or a period of the task and does not change in the course of time. The `abs_deadline` is job's deadline relative to system start. This is basically the same value as `real_priority` in TCB. The color of a node in the Red-Black tree is determined by `color`.

Budget for each thread (or its server) is set in `cmp_time`. As soon as a new period of a task starts, the `cmp_time` is copied into TCB's `cpu_time_budget` which corresponds to CBS server replenishment and the RTEMS core automatically performs gradual consumption of the `cpu_time_budget` as long as the thread is running.

Moreover, it has to be pointed out that implementation of the BWI protocol (as discussed in Section 6.4.3) is provided thanks to the way we handle the budget. Since the task consumes its budget only in a running state and not all the time after start, it is not necessary to perform any additional steps. The task is given only as much budget as it was agreed.

7.2.2 Red-Black trees

The ready queue of tasks for scheduling with dynamic priorities can not be carried out as simply as in case of static priorities where it is sufficient to enqueue the released jobs to the end of the chain. In the case of EDF it is necessary to have a possibility to insert a released job into arbitrary position in the queue. For this purpose a tree representation, in particular Red-Black trees, is usually used.

The Red-Black trees have not been implemented from scratch. The complete implementation was overtaken from a former EDF implementation by [Molnár, 2006]. The RB Tree just undertook a couple of simple adjustments in order to be able to handle threads represented by TCB and `RBT_Node`.

7.2.3 Pluggable scheduler callbacks

The actual scheduling rules corresponding to handling the ready queue in terms of inserting and extracting jobs according to their deadlines is managed by filling out the pluggable scheduler call-out functions Section 7.1. The complete implementation can be found in the source codes `scheduler_edf.c`.

One function to point out is the priority comparison mentioned in Section 5.2.4.

```
int _Scheduler_edf_Priority_compare
    (Priority_Control p1, Priority_Control p2) {
      uint32_t time = _Watchdog_Ticks_since_boot;
      // now sort priority levels
      if (p1 < EDF_HYBRID_MASK)
      p1 = (p1 - time) % EDF_HYBRID_MASK;
      if (p2 < EDF_HYBRID_MASK)
      p2 = (p2 - time) % EDF_HYBRID_MASK;
      if (p1 > p2) return -1;
      else if (p1 < p2) return 1;
      else return 0;
}
```

In order to make use of the Priority Inheritance Protocol already implemented in RTEMS, this function has to be incorporated to the set of pluggable scheduler callbacks. It has to be invoked not only upon selecting a position in the ready queue, but also when deciding whether to inherit a

priority of another task in accordance to PIP after a mutex is locked. This occurrence is the only one in the entire RTEMS core where it is necessary to introduce this explicit priority comparison using this approach (Section 5.2.4).

As it was mentioned in Section 6.4.2, it may be necessary to shift deadlines in certain cases as a job is unblocked to avoid a deadline miss of another job. This is allowed when a flag `EDF_LATE_UNBLOCK` is raised. Then, before the corresponding job is enqueued into a ready queue in `_Scheduler_edf_Unblock`, its deadline is shifted.

```
if (node->flags & EDF_LATE_UNBLOCK) {
  uint32_t Q_left = the_thread->cpu_time_budget;
  uint32_t P_left = node->abs_deadline - _Watchdog_Ticks_since_boot;
  uint32_t P = node->rel_deadline;
  uint32_t Q = node->cmp_time;
  if(P*Q_left > Q*P_left)
  edf_postpone_deadlines(the_thread);
}
```

7.2.4 EDF API

As periodic tasks consist of jobs, after each period the task is supposed to indicate a finished execution of current job, so that a deadline can be shifted and new job released. This is in RTEMS already implemented in terms of *Rate Monotonic* (RM) manager which is responsible for maintaining information about periodic executions such as missed deadlines and some statistics.

The main feature that we can make use of is waiting until the end of current period. As soon as the execution of job is finished, it is announced to the RM manager which suspends the task execution until the end of period. Consequently, the job is reenabled again in order to continue executing a next period. However, no deadline handling was present at this stage.

As you can see the RM manager is a very useful tool and we can build up the EDF API on it. The `edf_next_period()` is the function to be called by a task after finishing a period or before start of the first period. This function basically shifts the deadline and calls the `rtems_rate_monotonic_period()`.

Another functions are `edf_deadline_init()` and `edf_deadline_cancel()` that basically initiate and cancel a periodic behavior of a calling task. When a task is not assigned any deadlines it is considered as background task and

scheduled on a priority basis as described in Section 5.2.3. However, these two functions are called only when using a simple EDF. For the sake of FRSH treating CBS as a resource another API is used (see Section 7.3).

7.3 CBS API

As for the resource reservation of RTEMS CPU time it is necessary to come up with an interface which is going to be used by either an application programmer or, which is of our major concern, the FRSH layer of framework, namely the resource allocator specific for this particular resource of RTEMS CPU. The API should provide with a set of methods for creating and destroying a CBS server as well as with a possibility to attach and detach threads to the servers. Also, for tasks it is essential to have a possibility to be informed about its scheduling parameters especially the remaining budget, so that the tasks have some degree of feedback in order to always fit into the given budget.

For sake of consistency with previous AQuoSA project with totally the same focus, it was settled to use the same interface defined in [aqu, 2008] in header files `qres_lib.h` (QRES Library Application Programming Interface) and `qsup_lib.h` (The QoS Supervisor Application Programming Interface). So far, the entire interface is not utilized, especially in terms of spare capacity reservation, some of the functions are not implemented. However, it is necessary and logical to maintain the possibility of having the same set of features in the future.

At this stage, the CBS does not include any spare capacity handling and is limited to having only one task attached to a server at a time.

7.4 Adding a RTEMS CPU resource

General hints how to add a new resources are stated in [Molnár et al., 2008].

As a first resource added to the framework is obviously the CPU. As soon as we have a CPU scheduler, we can add this resource to this framework under name `cpu_rtems`. As we know from Chapter 2 in order to add a resource it is necessary to implement a Resource manager (FRM) and Resource allocator (FRA).

The FRM is a simple routine calculating a total utilization. If the total utilization (`sum_utilization`) including a new task negotiating its contract is lower than 88%, the contract is approved. The factor of 88% is partially arbitrarily selected in order to provide some space for background tasks even if the processor is fully loaded. Moreover, a space for context switching overhead is left.

The FRA is responsible for creating servers and assigning threads to the servers along with providing some feedback information about execution to the framework. This would not be difficult to implement from scratch. However, a decision has been made to take over a current API of AQuoSA (Section 6.3) for the CBS (Section 7.3). This makes things easier since these two projects have completely the same aim and there is completely no point in making up some new API. The AQuoSA API is defined in `qres_lib.h` [aqu, 2008]. Therefor the FRA for RTEMS CPU was just copy-pasted from AQuoSA FRA.

A block diagram how the final communication infrastructure looks like is depicted on Figure 7.1.

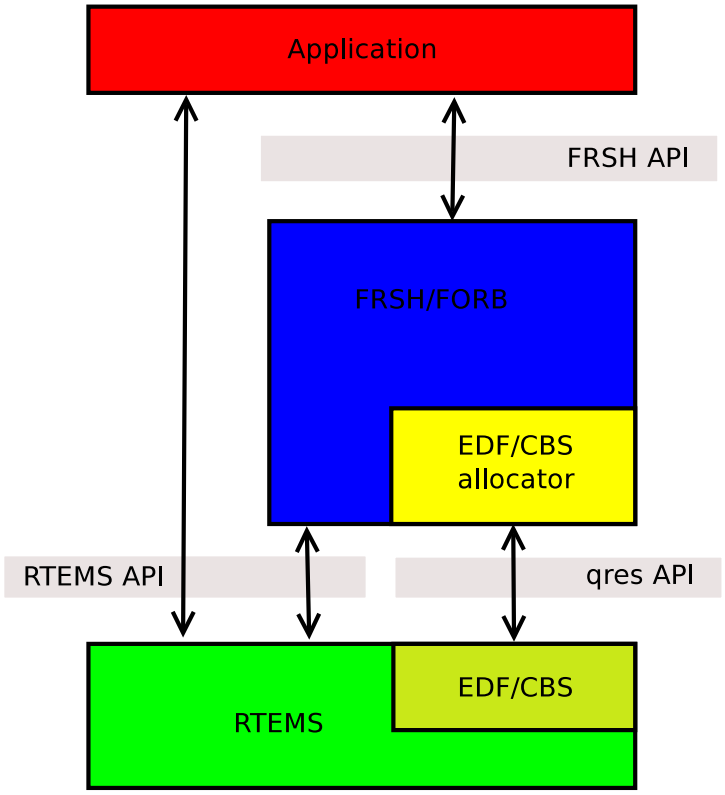


Figure 7.1: Block diagram of FRSH/FORB middleware including the RTEMS CPU resource.

Chapter 8

Validation of the work and testing

This chapter presents what tests were undertaken in order to validate the functionality of implementations and adjustments. Since a part of work has been carried out purely in Linux, in Section 8.1 the results of refactoring the middleware into a single address space in Section 3.1 will be stated. Consequently, the scheduler tests in RTEMS were created in order to make sure the algorithm has been implemented propertly in Section 8.2. Last part after compiling RTEMS and FRSH/FORB together an integration test is to be made in order to verify the capabilities already from the joint RTEMS+FRSH/FORB user perspective. This is presented in Section 8.3. It would be very desirable to run the same integration/system test directly on hardware, but there was no time to manage it any more.

8.1 Linux wvtests

The FRSH/FORB project includes a set of automatic tests (wvtests) which serve as a easy-to-use validation tool for the overall functionality of the framework. These tests help the developers to make sure that the entire framework works properly after each incremental update of the project. One part of work done purely on the Linux platform Section 3.1 has been tested exactly by this set of tests. After finishing this part of work all test results turned green saying OK.

A couple of new tests validating the proper functionality of different invocation methods in the FORB layer have been added to the set. The tests in the directory `src/forb/src/tests/executor/` examine the cases of remote

and inter-thread invocation of functions. Both of the cases finish successfully.

8.2 Scheduler tests

8.2.1 EDF test

A situation of two concurrent tasks has been made up. Task 1 is defined by period of 7 ticks (smallest unit of time resolution) and computation time just a fraction of one tick. Task 2 has a period of 10 ticks and computation time of 2 ticks. Since the total utilization is much lower than 1, the situation is schedulable by EDF and no deadline miss should occur. The figure Figure 8.1 is a console output of a program representing the introduced situation of two concurrent tasks. Every line is an information from a running task about its state related to periodic behavior. The letter S stands for starting time, F stands for finishing time and p indicates the task's priority (or deadline). Let me show that the scheduler works properly and the tasks are assigned CPU in a correct order on an example starting at time 160 where task 2 starts its execution with a deadline of 170. However, at time 161 a task 1 is also released with a deadline of 168 which is higher priority than currently running task 2. This causes a preemption of task 2 and a computation of task 1 continues. Consequently, after the execution of task 1 is done, the task 2 can finish its computation at time 162. Since the total utilization is very low, there is no other opportunity where the two tasks meet each other in the ready queue observable on the output screen. The program is asked to terminate at the time of 180, so this is the last output.

This test was first intended for QEMU emulator, however, it turned out that the timing properties of QEMU are not sufficient and fully deterministic. Therefore this test was carried out directly on 386 laptop invoked only after boot.

8.2.2 CBS test

This test shows the capabilities of CBS, where two concurrent tasks share processor. The tasks have a limited budget which they are aware of. They do not have any precomputed execution time but they keep running as long as they have some budget left. The remaining budget information is provided to the tasks using the CBS API.

The scheduling parameters for task 1 are period 7 and budget 3 ticks.

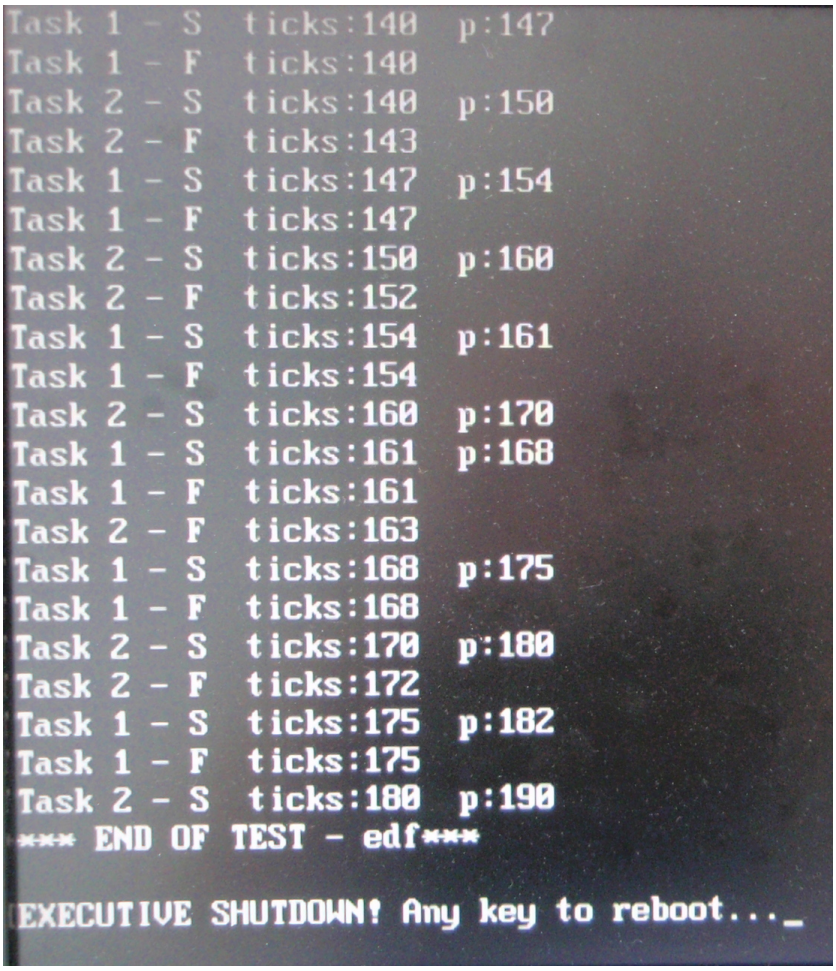


Figure 8.1: EDF scheduler output screenshot.

The task 2 has period of 11 and budget of 5 ticks. The tasks' job is just a loop performing a dummy memory allocation as long as there is a budget left. In the first case of Figure 8.2 where the tasks are aware of the budget, no problems in terms of budget overrun and deadline miss occur. The second case of Figure 8.3 uses completely the same scenario with a difference in task 2. At the time of 140 ticks it decides to neglect the assigned budget and continues executing as long as it is possible. Therefor, the CBS scheduler is obliged to suspend this violating task in order to maintain proper execution of task 1. Consequently, only the task 1 is further scheduled.

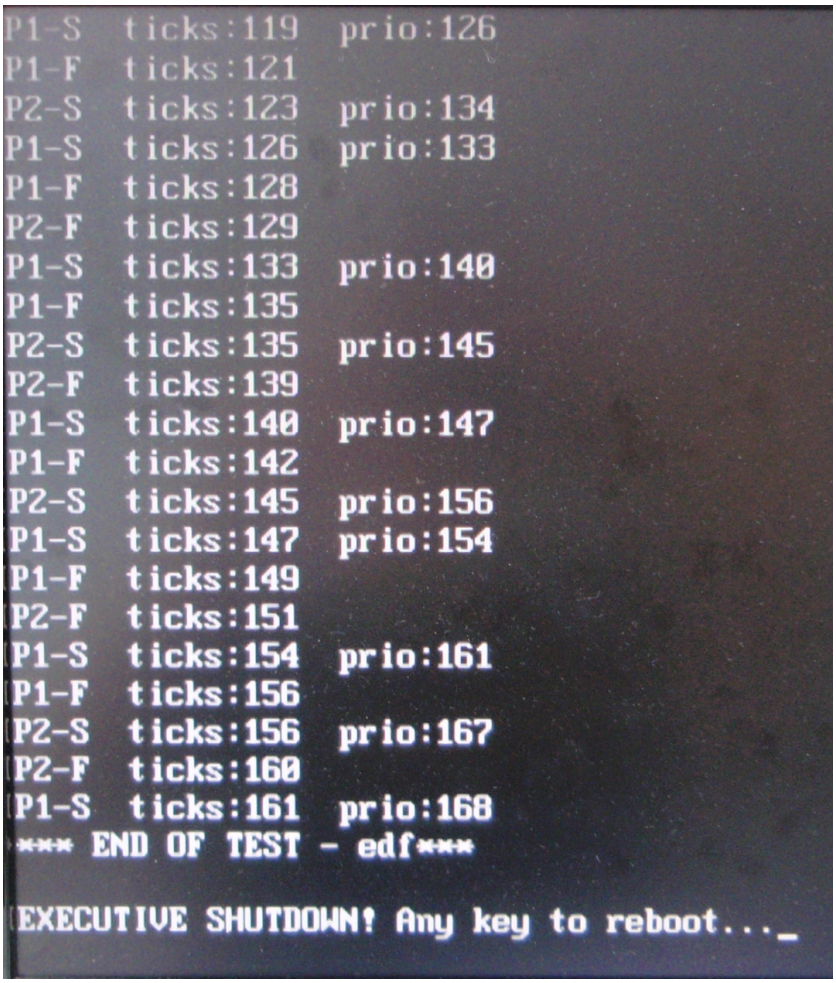


Figure 8.2: Test of CBS scheduler where both tasks watch out for their budget and yield the processor on time.

8.3 RTEMS+FRSH/FORB integration tests

The integration is not finished yet, the tests will be performed in a close future.

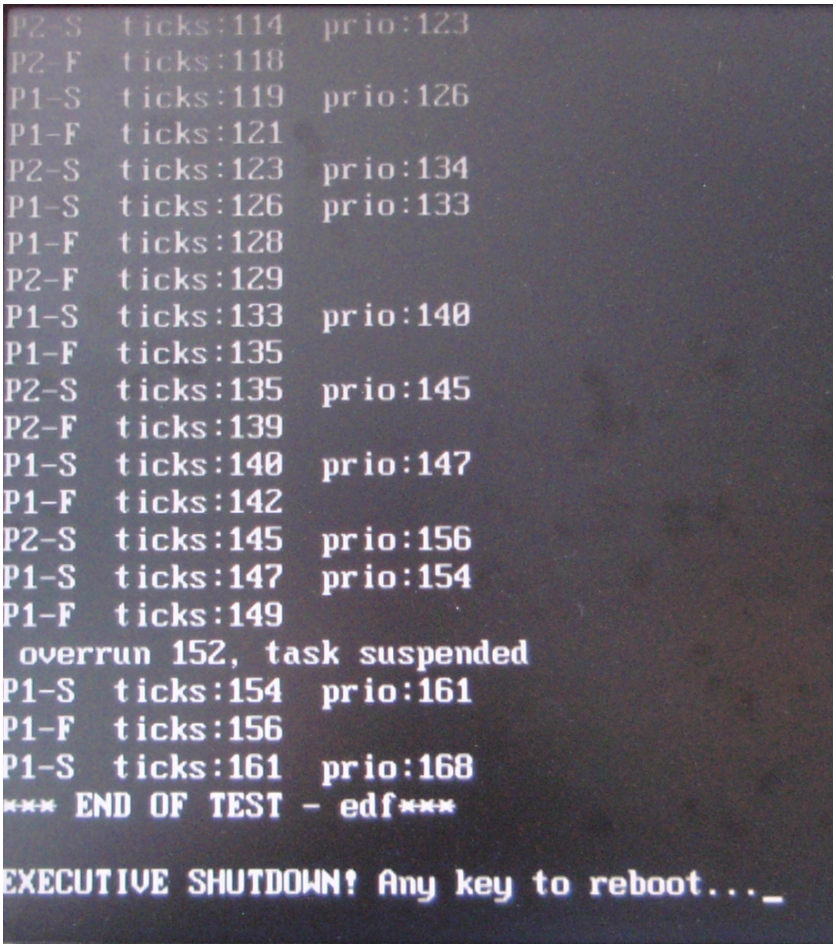


Figure 8.3: Test of CBS scheduler where Task 2 decides to neglect the budget limitation after time 140. The task is suspended by the scheduler.

Chapter 9

Conclusion

This thesis was rather a challenging work because the volume exceeds the usual time amount devoted to a master thesis. I have to point out that it was necessary first to get acquainted with almost the entire FRSH/FORB framework and a significant part of RTEMS OS just in order to comprehend the assignment. Moreover, this project has been performed under a supervision of RTEMS developers and FRSH/FORB developers while trying to search for optimal joint and fitting between these projects. That resulted into a significant number of group emails sent all around the world and valuable discussions.

The first part of project (refactoring into a single address space) was successfully finished. The EDF is fully working, even the temporally isolating CBS works, however, a proper testing of all features handling various scenarios has not been finished.

The RTEMS+FRSH/FORB fitting is not fully done yet since there is a couple of details missing or waiting for adjustments.

As far my personal experience given by this project is concerned, I have to admit that the thesis provided me valuable experience related to scheduling, operating systems theory and praxis, open-source software development as well as communication with world experts and a little bit of management in order to actually find out the practical requirements and sense of this project. To be honest, it was a good choice to apply for this topic.

9.1 Future work

Since the RTEMS EDF scheduler adopts only PIP protocol, more complicated applications might become deadlock-prone. This can be avoided by a simple implementation of NPCS protocol as mentionded in Section 5.2.2.

Proper integration/system testing has to be performed as soon as it is managed to suppress all minor issues and the FRSH/FORB+RTEMS is linked together.

An unifying library Ulevpoll [Píša, 2011] for Select, Epoll and other libraries dealing with servicing multiple waiting UNIX sockets might be used. Currently, the source code is not very readable because of conditional compilation.

As a continuation of this thesis was accepted for Google Summer of Code, the work will continue an all the missing adjustments finished within oncoming months.

Bibliography

[aqu, 2008] (2008). Adaptive quality of service architecture - web. http://aquosa.sourceforge.net. retrieved 08/05/11.

[fre, 2008] (2008). Frescor project website. http://www.frescor.org. retrieved 22/03/11.

[epo, 2010] (2010). Epoll - Linux programmer's manual. http://www.kernel.org/doc/man-pages/online/pages/man4/epoll.4.html. retrieved 05/01/11.

[omk, 2010] (2010). Omk online documentation. http://rtime.felk.cvut.cz/omk. retrieved 22/03/11.

[rti, 2010] (2010). Real-time systems group wiki, ctu. http://rtime.felk.cvut.cz/hw/index.php/MIDAM_MPC5200_DB1. retrieved 12/03/11.

[rte, 2010] (2010). RTEMS wiki. http://www.rtems.com/wiki/. retrieved 03/03/11.

[cor, 2011] (2011). CORBA website. http://www.corba.org.

[frs, 2011] (2011). Frsh/forb project website. http://frsh-forb.sourceforge.net. retrieved 22/05/11.

[qem, 2011] (2011). Qemu wiki. http://wiki.qemu.org.

[sel, 2011] (2011). Select - Linux man page. http://linux.die.net/man/2/select. retrieved 05/01/11.

[Abeni and Buttazzo, 1998] Abeni, L. and Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–, Washington, DC, USA. IEEE Computer Society.

[Abeni et al., 2005] Abeni, L., Cucinotta, T., Lipari, G., Marzario, L., and Palopoli, L. (2005). Qos management through adaptive reservations. *Real-Time Syst.*, 29:131–155.

[Brandenburg et al., 2000] Brandenburg, B. B., Block, A. D., Calandrino, J. M., Devi, U. M., Leontyev, H., and Anderson, J. H. (2000). Litmus: A status report*.

[Cucinotta and Checconi, 2010] Cucinotta, T. and Checconi, F. (2010). The IRMOS realtime scheduler. `http://lwn.net/Articles/398470/`.

[Faggioli et al., 2008] Faggioli, D., Lipari, G., and Cucinotta, T. (2008). An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time apps. in the Linux kernel.

[Faggioli et al., 2010] Faggioli, D., Lipari, G., and Cucinotta, T. (2010). The multiprocessor bandwidth inheritance protocol. In *ECRTS'10*, pages 90–99.

[Kim and Shin, 1997] Kim, B. K. and Shin, K. G. (1997). Scalable hardware earliest-deadline-first scheduler for atm switching networks. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 210–, Washington, DC, USA. IEEE Computer Society.

[Lipari et al., 2004] Lipari, G., Lamastra, G., and Abeni, L. (2004). Task synchronization in reservation-based real-time systems. *IEEE Trans. Comput.*, 53:1591–1601.

[Liu, 2000] Liu, J. W. S. W. (2000). *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

[Molnár, 2006] Molnár, M. (2006). The edf scheduler implementation in RTEMS operating system. Master thesis.

[Molnár et al., 2008] Molnár, M., Trdlička, J., Jurčík, P., Smolík, P., Sojka, M., and Hanzálek, Z. (2008). Wireless networks – documented protocols, demonstration.

[OAR, 2009] OAR (2009). RTEMS C users' guide. `http://www.rtems.com/`.

[OAR, 2010] OAR (2010). Writing an RTEMS scheduler plugin. draft 1.

[Palopoli et al., 2008] Palopoli, L., Cucinotta, T., and Lipari, G. (2008). Aquosa - adaptive quality of service architecture.

[Píša, 2011] Píša, P. (2011). ulan/universal light event poll library (ulevpoll). `ulan.sourceforge.net/pdf/ulevpoll.pdf`.

[Sojka, 2010] Sojka, M. (2010). Resource reservation and analysis in heterogenous and distributed real-time systems. Ph.D. thesis.

[Sojka et al., 2011] Sojka, M., Píša, P., Faggioli, D., Cucinotta, T., Checconi, F., Hanzálek, Z., and Lipari, G. (2011). Modular software architecture for flexible reservation mechanisms on heterogeneous resources. *Journal of Systems Architecture*, 57(4):366–382.

Appendix A

Getting FRSH/FORB and RTEMS

A.1 Compilation of FRSH/FORB

The FRSH/FORB project can be found on the website (frsh-forb.sourceforge.net) [frs, 2011] and downloaded from a GIT repository residing at

```
git clone  git://frsh-forb.git.sourceforge.net/gitroot/frsh-forb/frsh-forb
```

Detailed hints on how to set up and build the framework can be found in the README of the repository.

A.2 Building RTEMS

Versions. The current RTEMS release is 4.10 version available in a GIT repository, however, there are some brand new features implemented after the 4.10 release. It is available in CVS and named 4.11 already, although the version is not officially released yet. The motivation for picking the latest version of RTEMS from CVS HEAD was presence of Pluggable Scheduler infrastructure (see Section 4.3.1), which is very useful for us since we need to implement a CBS/EDF scheduler which is of course not a part of the RTEMS core.

Building toolchain. The drawback is that the latest 4.11 toolchain was not available and I was forced to build the tools myself. I did not encounter any problems during the procedure since the patches are consistent and up-to-date for all of the utilities. Since there are numerous descriptions how

to build the RTEMS toolchain [Molnár, 2006], I will not put any effort on explaining this issue.

Since of the main interest of this project are the embedded applications, the testing of results was (and still is) planned to be performed on a PowerPC board MIDAM MPC5200. This board is compatible with RTEMS Icecube BSP [rti, 2010]. In order to build RTEMS for this target, it is necessary to use a specific configuration

```
$ ../rtems/configure --disable-multilib --disable-cxx --enable-posix
    --enable-networking --target=powerpc-rtems4.11
    --prefix=$(PWD)
    --enable-rtemsbsp=icecube
```

However, for sake of having a better development environment in terms of simulator and debugger, decision has been made to build up RTEMS also with a the pc386 BSP. Moreover, it is was found that a very good and reasonable testing with this BSP can be performed directly on a laptop (see Chapter 8).

```
$../rtems/configure --disable-multilib --disable-cxx --enable-posix
    --enable-networking --target=i386-rtems4.11
    --prefix=$(PWD)
    --enable-rtemsbsp=pc386
    --enable-maintainer-mode target_alias=i386-rtems4.11
    --no-create  --no-recursion
```

A.2.1 Testing and debugging tools

The advantage of pc386 BSP is that it is possible to run the applications in a hardware emulator such as BOCHS of QEMU [qem, 2011] (there is also PSIM for the PowerPC targets). Moreover, you can even run the application on your development PC directly. I decided to perform all the testing in QEMU with a connected debugger simultaneously. The advantage of QEMU is that the console output is provided, however, there is just a few lines of output to see at once in the console, so if you require to log these outputs it is better to redirect them somewhere else. According to mailing-list, it is possible to redirect standard output into serial port (using `USE_COM1_AS_CONSOLE=1` flag), but I did not succeed with this. The problem lies probably rather on the side of RTEMS than in QEMU, because I can get the serial port output from GRUB running in QEMU. I have to note that I work in Linux, where

nothing called COM1 exists.

It was found, however, that QEMU is reasonable good tool in case timing properties are not of the highest interest but for a study of concurrent behavior it is not the best way. Moreover, for sake of execution time measurements of tasks it is not useful at all. For this purpose it is better to run the compiled applications directly on a laptop after boot (e.g. from GRUB).

As a reasonably good tool for debugging is *gdb* unless you are trying to examine timing properties and concurrent tasks. In that case breakpoints do not help out. In order to connect to your running application (in a simulator) by gdb, use following commands in gdb (or place them into .gdbinit file in your Linux home directory along with your required breakpoints, it will execute just after the start of gdb).

```
file $(relative_path_to_compiled_image)
target remote localhost:1234
```

Content of attached CD

To this thesis a CD including following content is attached:

- Directory `pdf`: Soft copy of this thesis
- Directory `src`: Source codes
- Directory `app`: Attached documents