Building on the DMI and EDFI foundations

J.M. Maurer

Dept. of Electrical Engineering,
Mathematics and Computer Science
University of Twente
PB 217, 7500 AE Enschede
The Netherlands
j.m.maurer@cs.utwente.nl

29 augustus 2005

Graduation committee
ir. P.G. Jansen
ir. F. Hanssen
ir. J. Scholten

Abstract

This paper gives a summary of the DMI and EDFI real-time scheduling algorithms, first presented by P. G. Jansen. During this discussion a number of imperfections in the existing theory are corrected.

Several additions to the existing theory are presented as well, including a dynamic task admission condition, support for non-preemptable nested critical sections and multi-use resources.

Both the DMI and EDFI schedulers together with their accompanying admission control mechanisms have been implemented in the RT-Linux operating system. The DMI and EDFI scheduler organisation has been adapted to work within the existing RT-Linux scheduler framework. Special care has been taken during the implementation of the schedulers to keep their worst case overhead behavior as low possible.

The implementation has been verified for its correctness using several tools that we developed. A variety of tests have been executed to get an indication of the scheduler performance under various conditions. Results show that the scheduling overhead is relatively low, less than one percent for a task set that could arguably resemble a real-life situation.

Samenvatting

Dit paper geeft een samenvatting van de DMI en EDFI real-time scheduling algoritmen, voor het eerst beschreven door P.G. Jansen. Tijdens deze verhandeling worden enkele onvolkomenheden in de bestaande theorie gecorrigeerd.

Enkele toevoegingen aan de bestaande theorie worden gepresenteerd, zoals een dynamische taak toevoegingsconditie, ondersteuning voor niet-onderbreekbare geneste kritieke secties en zogenaamde multi-use resources.

Zowel de DMI als EDFI scheduler zijn, samen met hun bijbehorende toelatingscontrole-mechanismen, geïmplementeerd in het RT-Linux besturingssysteem. De DMI and EDFI scheduler organisatie is aangepast om te werken binnen het RT-Linux scheduler raamwerk. In onze implementatie hebben we gepoogd het worst-case overhead gedrag van de schedulers zo laag mogelijk te houden.

De implementatie is gecontroleerd op zijn correctheid door gebruik te maken van enkele door ons ontwikkelde gereedschappen. Een verscheidenheid van tests is uitgevoerd om een indicatie te krijgen van de scheduler prestaties onder verschillende condities. Resultaten laten zien dat de scheduling overhead relatief laag is, minder dan één procent voor een taakset die een real-life situatie probeert na te bootsen.

Contents

Abstract i

Samenvatting ii

List of Figures vii

1 Introduction 1

1.1 Goals . . . 1

1.2 Overview . . . 2

2 Real-time scheduling 3

2.1 Basic concepts . . . 3

2.2 Overview . . . 4

2.2.1 Periodic real-time scheduling . . . 4

2.2.2 Resource scheduling . . . 5

3 DMI and EDFI foundations 7

3.1 Introduction . . . 7

3.2 Resource scheduling . . . 7

3.2.1 Deadline inheritance . . . 8

3.3 Protocol organisation . . . 9

3.3.1 Properties . . . 10

3.4 Feasibility analysis . . . 10

3.4.1 DMI feasibility . . . 10

3.4.2 EDFI feasibility . . . 11

3.5 Dynamic task admission . . . 14

3.6 Non-preemptable nested critical sections . . . 16

3.6.1 Problem . . . 16

3.6.2 Theory . . . 16
3.6.3 Specification . . . 17
3.7 Multi-use resources . . . 17
3.7.1 Model . . . 17
3.7.2 Specification . . . 18
3.7.3 Sufficient feasibility analysis . . . 18
3.7.4 Necessary feasibility analysis . . . 19
3.7.5 Online operation . . . 20

4 Design **23**
4.1 Platform determination . . . 23
4.2 Real-Time Linux workings . . . 23
4.2.1 Real-Time Linux scheduler . . . 24
4.3 DMI and EDFI in RT-Linux . . . 25
4.3.1 Task preemption . . . 26
4.3.2 Task completion . . . 26
4.3.3 Task releasing . . . 27
4.3.4 Missing deadlines . . . 27
4.3.5 Nested critical sections . . . 27
4.3.6 Idle time . . . 28

5 Implementation **31**
5.1 Application Programming Interface . . . 31
5.2 Data structures . . . 32
5.3 Task creation . . . 32
5.4 Admission control . . . 32
5.5 NCS usage . . . 33
5.6 Performance considerations . . . 33
5.6.1 Signaling . . . 33
5.6.2 NCS traversal . . . 34
5.6.3 Avoiding overhead peaks . . . 34
5.6.4 Latency over overhead . . . 34
5.6.5 Compiler based optimizations . . . 34

6 Scheduler analysis **39**

6.1 Validation 39
6.2 Test setup 41
6.2.1 Overhead breakdown 41
6.2.2 Data acquisition 42
6.2.3 System configuration 42
6.3 Results 43
6.3.1 Test 6-1: DMI scheduler overhead 43
6.3.2 Test 6-2: EDFI scheduler overhead 45
6.3.3 Test 6-3: Periodic scheduler overhead 46
6.3.4 Test 6-4: Job execution jitter 47
6.3.5 Test 6-5: Dynamic task admission and removal 48
6.3.6 Test 6-6: NCS usage 49
6.3.7 Test 6-7: Typical task set 52
6.3.8 Discussion 53

7 Future work **55**
7.1 Theory 55
7.1.1 DTA condition analysis 55
7.1.2 Multi-use resources 55
7.2 Implementation 56
7.2.1 Resource acquisition 56
7.2.2 Semantical correctness 56
7.2.3 Timer latencies 56
7.2.4 Backwards compatibility 57
7.2.5 Measurements 57
7.2.6 Testing 57

8 Conclusions **59**

Bibliography **61**

A Real-time task example **63**

B Utilities **65**
B.1 rtlinux-si 65
B.2 rtlinux-measurements 65

B.3 rtlinux-testsuite 65
B.4 rtlinux-admctrl 65
B.5 rtlinux-sync 66
B.6 rtlinux-ksymoops 66

List of Figures

2.1 Execution of Γ under DM 4

3.1 DMI / EDFI protocol organisation 9
3.2 L_B proof outline 13
3.3 Example run stack for Γ 15
3.4 Example run stack for Γ after admission of τ_4 15

4.1 Real-Time Linux architecture 24
4.2 RT-Linux task list example 24
4.3 DMI / EDFI task list organisation 25
4.4 Task states and transitions 26
4.5 NCS structure example 28
4.6 Example of tasks executing in idle time 29

5.1 Data structure overview 36
5.2 Performance optimized NCS structure 37
5.3 Compiler based optimizations 37

6.1 Scheduler Insight diagnostic tool 40
6.2 Job execution showing scheduler overhead 41
6.3 Example pthread routine 41
6.4 DMI scheduler overhead for Γ_{6-1} 44
6.5 DMI scheduler overhead for Γ_{6-1} on a CPU with 1MB cache 45
6.6 EDFI scheduler overhead for Γ_{6-1} 46
6.7 Scheduler overhead for Γ_{6-3} 47
6.8 Job execution jitter 48
6.9 Dynamic task admission and removal 50

6.10 NCS overhead 51
6.11 Typical task set scheduler overhead 52

Chapter 1

Introduction

This report describes the research that was conducted as part of a graduation assignment. The subject centers around real-time computing, or to be more precise, real-time scheduling. Broadly stated, real-time scheduling is the field of research where the timely execution of tasks must be guaranteed. Tasks are generally executed on a periodic basis, with a fixed time interval between each successive execution. The execution must be guaranteed to be finished before a specific point in time, regardless of other jobs that the system might have to perform.

Guarantees are not too hard to give when there are only one or two tasks in the system. However, as one can imagine it becomes more difficult to give any timing guarantees when there are a tens of tasks to execute, each possibly with different computational demands.

Quite a number of algorithms have been devised over the last few decades with the purpose to assign tasks to the central processing unit of a system in such a way that the required timing guarantees can be given. Such algorithms are generally known as *real-time scheduling algorithms*.

This paper deals with two scheduling algorithms which were first described by Jansen[8] called *Deadline Monotonic with Inheritance* (DMI) and *Earliest Deadline First with Inheritance* (EDFI). Both algorithms can give the required timing guarantees, even when handling complex task sets. Additionally, both algorithms have the capability to still ensure the timely execution of tasks when they are allowed to acquire *resources*. Resources are (often) physical entities with a special capability a task might require. Think for example about a radio that can be used by tasks to transmit data over a radio channel. Often only a single task is allowed to use the radio at any moment in time. If two tasks try to send data simultaneously over the same channel, their signals might interfere and come out scrambled.

1.1 Goals

The goals of the research described in this paper can be summarized as:

- Create a complete implementation of the DMI and EDFI theory described by Jansen[8].
- Expand the DMI and EDFI theory to allow for dynamic task admission, non-preemtable nested critical sections and multi-use resources. Implement these additions as well.
- Measure the overhead of the implemented DMI and EDFI algorithms

By implementing the DMI and EDFI algorithms, we can verify that the theory can be used in practice. Furthermore, the implementation can be used as a testbed for various other research projects. The token based real-time network RTnet[7] for example could be reimplemented on an EDFI based real-time kernel, as it currently depends on the non real-time Linux kernel.

The expansion of the existing theory in the areas described above make the DMI and EDFI scheduling algorithms more useful in practice. All expansions are simple to describe, trivial to verify for their correctness, and relatively easy to implement. The expansions are part of this paper as this paper focuses for a big part on the practical aspects of DMI and EDFI.

Insight into the overhead that the algorithms generate is importance to their usefulness in practice. This paper will present an analysis of the overhead that the implemented algorithms generate, albeit not an in depth analysis. While the overhead can be determined with great precision[1], such an exhaustive research is outside the scope of this paper.

1.2 Overview

A short overview of the current state of affairs in the world of real-time scheduling will be presented in chapter 2. It introduces the reader to the basic concepts of real-time scheduling algorithms and places this research into its proper context.
Chapter 3 will describe the existing DMI and EFDI theory and will add several corrections to the original work. Several additions to the existing theory will be presented, such as a sufficient condition under which dynamic task admission is allowed, a solution to allow non-preemptable nested critical sections, and a theory under which multi-use resources can be allowed.
Chapter 4 will give an overview of the Real-Time Linux operating system architecture. Adaptations to this architecture will be presented to allow for an efficient implementation of DMI and EDFI.
Chapter 5 will describe our implementation of DMI and EDFI in Real-Time Linux. The additions and changes to the application programming interface are documented, processes are described in terms of low level functionality, and several performance considerations will be addressed.
Chapter 6 will describe how the correctness of the scheduler implementation was verified. Furthermore the scheduler performance will be analysed. Scheduling data will be retrieved by executing a number of carefully chosen test cases on a particular target platform. Tests are executed to measure the behavior of various scheduler aspects and to determine the average and worst case scheduling overhead under various conditions.
Chapter 7 will describe the future work that could be done based on this research.
Chapter 8 will finally present several conclusions that can be drawn from the research presented in this paper.

[1]The commercial VxWorks operating system developed by the company Wind River is an example of a real-time system for which the scheduler overhead has been examined into the finest detail

Chapter 2

Real-time scheduling

Paragraph 2.1 will discuss the basic concepts used in the field of real-time scheduling. These concepts will be used throughout this paper. Paragraph 2.2 will describe in short the existing real-time scheduling theory that is related or relevant to this research, and will place this research into the proper context.

2.1 Basic concepts

For the reader who is unfamiliar in the field of real-time scheduling we will start by presenting several commonly used definitions:

Definition 1 (Task τ_i)**.** *A real-time task τ_i is described by the tuple (D_i, C_i, T_i). D_i denotes the relative deadline of τ_i, C_i its load and T_i its period.*

Tasks are executed periodically. Each specific occurrence is called a *job*.

Definition 2 (Job τ_i^j)**.** *A job τ_i^j is the jth invocation of task τ_i. It is described by the tuple (r_i^j, D_i^j, C_i^j), where r_i^j represents the starting time or release time of τ_i^j, D_i^j its relative deadline and C_i^j its load.*

The interval between two subsequent jobs τ_i^j and τ_i^{j+1} is equal to the period of the task to which they belong, T_i. The relative deadline is the time interval in which the job should finish its execution, starting from job's release time. The condition $D_i \leq T_i$ is thus assumed to hold. The load of a task is the amount of computation time that is needed to execute a job. This implies that the condition $C_i \leq D_i$ should always hold.

Definition 3 (Task set Γ)**.** *A task set Γ is a set of tasks $\{\tau_1, \tau_2, ..., \tau_n\}$ that is presented to the scheduler for execution.*

A *scheduler* assigns jobs given a task set to the processor for execution at specific times. The collection of assignments is called a *schedule*. The scheduler constructs this schedule by executing a *scheduling algorithm*, several of which will be discussed in paragraph 2.2.
A *real-time scheduler* has the additional ability of giving guarantees about the timely execution of a job. It can predict in advance if a task set scheduled using particular scheduling algorithm is possible or not: if all jobs will meet their deadline the task set is said to be *feasible* under that algorithm; it is said to be *infeasible* otherwise.

Definition 4 (Utilisation U)**.** *The utilisation U denotes the amount of load that a task set Γ imposes on a system. The utilisation U is expressed by a value $0 \leq U \leq 1$, and is defined as:*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

2.2 Overview

2.2.1 Periodic real-time scheduling

One of the first described periodic task scheduling algorithms is called *Rate Monotonic* (RM). This algorithm is a simple rule that assigns fixed priorities to tasks. The shorter the period, the higher the priority. Tasks are preemptable, and their relative deadlines are assumed to be equal to their period. Liu and Layland[11] showed that RM is optimal[1] in the class of fixed priority algorithms.

A fixed priority scheduling algorithm closely related to RM is called *Deadline Monotonic* (DM), first proposed by Leung and Whitehead[13]. The difference with RM is that tasks can have a relative deadline that is smaller than their period. Table 2.1 shows a specification for an example task set called Γ. When scheduling this task set under DM, a schedule as depicted in figure 2.1 can be constructed. The up pointing arrows denote the start of a new period (also called a job *release*), while the down pointing arrows denote absolute job deadlines. The gray boxes depict that a job is executing. As can be seen in this example, the highest priority task τ_1 will preempt the lowest priority task τ_3 at time $t = 8$.

Γ	D	T	C
τ_1	3	4	1
τ_2	4	5	1
τ_3	7	7	3

Table 2.1: Example task set Γ

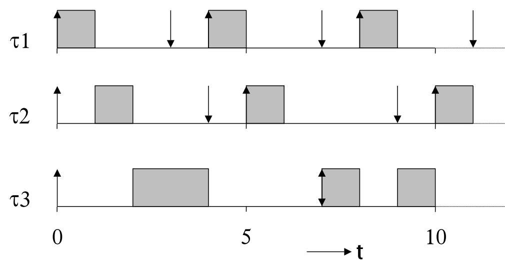

Figure 2.1: Execution of Γ under DM

Earliest Deadline First (EDF) is, in contrast to the previous algorithms, a dynamic scheduling algorithm. Priorities are assigned based on the absolute deadline of a task. The earlier the deadline of a task, the higher its priority will be. Tasks are assumed to be preemptable. EDF is optimal in the class of dynamic priority algorithms. Liu and Layland[11] showed that a task set is schedulable under EDF if, and only if: $U \leq 1$.

A number of variations on the scheduling algorithms described above have been presented over the past few decades. Some algorithms allow the execution of aperiodic tasks while still providing guarantees for periodic tasks[2], some handle precedence constraints between tasks[3] and even others are suited for use on multiprocessor systems[4]. Although interesting, these algorithms are out of the scope of this paper.

[1] Optimal in this context means that if a task set can be scheduled using any fixed priority assignment other than RM, then it can also be scheduled by RM.

[2] A well known system for handling aperiodic requests in a real-time system is the Total Bandwidth Server, proposed by Spuri and Buttazzo[18][19].

[3] Lawler was the first to solve the simple scheduling problem of a set of tasks with no preemption, identical arrival times, and a precedence relation among them[12].

[4] Most multiprocessor scheduling problems are NP-complete and they are the cause of an interesting set of multiprocessor anomalies, called Richard's anomalies[5].

2.2.2 Resource scheduling

Another group of real-time scheduling algorithms not mentioned in the previous paragraph is the group that deals with *resource scheduling*. This is the area of real-time scheduling that describes the context for research presented in this paper. Resource scheduling provides the functionality to create *critical sections* on a scheduler level. Within these sections tasks can access resources, such as a radio transmitter, without interference from other tasks that want to access the same resource.

The means by which critical sections are created differs between various algorithms, each having their own advantages and drawbacks. Audsley[1] has given a good overview of the existing techniques for dealing with resource control.
One of the most widely used techniques is called *priority inheritance*. This technique makes a task adopt the priority of another task to block tasks that would normally have a higher priority from executing. When not using this technique carefully, phenomena such as *deadlocking* and *chained blocking* can cause severe problems. A deadlock occurs when tasks are waiting for each other to release their resources, leaving the system in a state that cannot be left. Chained blocking occurs when a task must acquire resources that are currently held by more than one task; the task is blocked from executing until all the required resources are released by the tasks holding them.
Another problem that arises when using priority inheritance is called *priority inversion*[3], p184. This problem occurs when a low priority task has acquired a resource that a higher priority task also needs to acquire. This high priority task is then blocked from execution. Now the low priority task (that must release its resource before the high priority task can execute) can be preempted by a medium priority task that does not require that resource.

The algorithms based on priority inheritance that are of the most interest to this paper are listed below. While we will shortly address their workings here, the reader is encouraged to familiarize himself with the theories behind these algorithms:

- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Stack Resource Policy (SRP)

The PIP rule set[17] allows a resource R to be acquired when it is free. When R is already acquired by a lower priority task, the lower priority task will inherit the priority of the higher priority task. While this prevents the priority inversion problem from occurring, it does not prevent deadlocks or chained blocking.
The motivation of the PCP is to address the deadlock and chained blocking problems. The PCP rule set assigns a priority ceiling to each resource equal to the highest priority of all the tasks that acquire it. A task can only acquire a resource if its priority strictly greater than the ceilings of all currently acquired resources. Similar to the PIP a task executes at its normal priority unless it blocks a higher priority task, upon which it will inherit the priority of the blocked task[5]. A drawback of PCP is that it is relatively pessimistic in terms of blocking time[16][17].
The SRP provides several improvements to the PCP, such as support for multiunit resources[6] and support for tasks with dynamic priorities. A key feature of this protocol is a fixed preemption level that is assigned to each task that controls how tasks may preempt each other. A task with a low preemption level will be prevented from preempting a task with a high preemption level[2]. Several ideas found in the SRP have influenced the creation of the DMI are EDFI algorithms that are discussed in the next chapter.

[5]A number of variations on the PCP have been presented over the years, such as the Semaphore Control Protocol[14], the Dynamic Priority Ceiling Protocol[4] and the Ceiling Semaphore Protocol[15]. Although these algorithms present a lot of interesting contextual information to the research presented in this paper, their exact workings are outside the scope of this paper.

[6]Multiunit resources allow multiple readers and/or writers to simultaneously acquire a unit of that resource until all units are handed out.

Chapter 3

DMI and EDFI foundations

Our research is based upon the DMI and EDFI scheduling algorithms, which were both originally presented by Jansen[8]. To make the research presented in this paper more accessible, we will first recapitulate the foundations of DMI and EDFI. This is done in the sections 3.1 to 3.4[1]. These sections also contain a number of corrections to errors that were present in the original work.
As we do not simply want to rehash the original work, we will only present the theorems, equations and definitions that are needed to understand this research. If the reader wants a broader background on both algorithms, he is advised to read Jansen's original work.
Several extensions to the existing DMI and EDFI theory are presented in the sections 3.5 to 3.7. These extensions include support for dynamic task admission, non-preemptable nested critical sections, and multi-use resources.

3.1 Introduction

The DMI and EDFI scheduling algorithms are, the names imply, closely related to the DM and EDF algorithms respectively. DMI is basically DM with support for *deadline inheritance*. Similarly is EDFI basically EDF with deadline inheritance. The additional support for deadline inheritance allows the scheduler to support the use of resources. The beauty of these algorithms lies in the fact that the means by which resource scheduling is achieved is both elegant and easy to comprehend. This in contrast to the relative complexity of Stack Resource Protocol for example.

3.2 Resource scheduling

To support resource scheduling in a transparent way to application developer, the scheduler needs to know which resources a task uses, how long it uses them, when it uses them, and if it just "reads" the resources, or "writes" them as well. Without this information, the scheduler can not calculate the feasibility of a given task set or prevent deadlocks and the priority inversion problem (see paragraph 2.2.2).

A resource usage specification language has been devised to describe all the resource usage information that the scheduler needs. The language we present here is similar to the, slightly less generic, language described by Jansen[9]:

[1] This research will only focus on the DMI and EDFI theory that is based on nested critical sections (NCS). Transaction based DMI and EDFI scheduling is considered to be a special case of NCS based scheduling

ρ_{list} : $float\ '\{'\ R_{list}\ \rho_{list}\ '\}'\ \rho_{list}{}^*$
R_{list} : $R\ R^*$
R : $'a'..'z'\ |\ 'A'..'Z'$

A particular resource is denoted by a letter. By convention, a capital letter is used when a tasks writes the resource. A non-capital letter is used if the task just reads the resource.

An example resource usage description could look like "$2.0\ \{A\}$", which would translate as: *a task writes resource A for 2.0 units of time*. A more complex example might look like "$3.0\ \{A\ B\ 2.0\ \{c\}\ 0.5\ \{d\}\}$", which would translate as: *a task writes both resources A and B for 3.0 units of time; within that time, resource C is read for 2.0 units of time and after resource C is released, resource D is read for 0.5 units of time.*

Every task is accompanied by resource usage description which consists of zero or more sections separated by curly brackets, '{' and '}'. The sections can be embedded in each other, as the grammar and the examples in the previous paragraph show; every section hence called a *nested critical section*, or NCS in short. While executing, a task will enter and leave its nested critical sections in a subsequent manner. *Critical* stems from the fact that when a task enters particular NCS, it must be guaranteed that it can acquire the resources specified in that NCS without interference from other tasks.

Definition 5 (NCS length $C_{i,j}$)**.** *The length of the jth NCS of task τ_i is denoted by $C_{i,j}$.*

3.2.1 Deadline inheritance

Whenever more than one task reads and/or writes the same resource, interference problems might arise. Resources are expected to be written to in a mutual exclusive manner; ie. a task must be able to write to a resource without interference from other tasks that want to read or write that resource. Other tasks can only acquire that resource for reading or writing after it is released again by the task that acquired it. Reading a resource is assumed to be allowed to take place in parallel.

To prevent tasks from acquiring resources that are already in use, so called $resource floors$ have been defined by Jansen[8], p32-36. Each resource R has a distinct *read floor* D_R^r and *write floor* D_R^w. We have adapted the original definitions slightly to express their purpose more clearly:

Definition 6 (Read floor D_R^r)**.**

$$D_R^r = \min\{\{D_i | \tau_i \in \gamma^w(R)\} \cup \{\infty\}\} \tag{3.1}$$

where $\gamma^w(R)$ is the set of tasks that write R.

Definition 7 (Write floor D_R^w)**.**

$$D_R^w = \min\{\{D_i | \tau_i \in \gamma^{w+r}(R)\} \cup \{\infty\}\} \tag{3.2}$$

where $\gamma^{w+r}(R)$ is the set of tasks that read and/or write R.

Given these definitions, Jansen defined the *inherited deadline* $\Delta_{i,j}$ for NCS j of task τ_i as:

Definition 8 (NCS inherited deadline $\Delta_{i,j}$)**.**

$$\Delta_{i,j} = min(\{D_i\} \cup \{D_R^r | R \in \phi_j^r(\tau_i)\} \cup \{D_R^w | R \in \phi_j^w(\tau_i)\}) \tag{3.3}$$

where $\phi_j^r(\tau_i)$ is the set of resources that are read by τ_i in NCS j, and $\phi_j^w(\tau_i)$ the set of resources that are written by τ_i in NCS j.

This inherited deadline definition can be used to prevent tasks from acquiring resources that are already in use. The way in which this is achieved will be discussed in the next section.

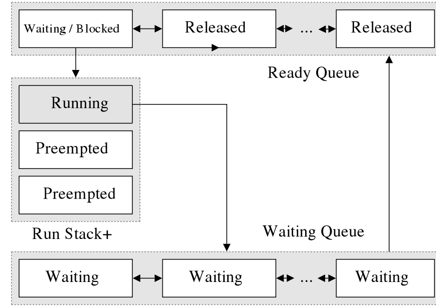

Figure 3.1: DMI / EDFI protocol organisation

3.3 Protocol organisation

The DMI and EDFI protocols share the same organisation, which is graphically depicted in figure 3.1. The organisation consists of a so-called *run stack*, *ready queue* and *waiting queue*. Tasks in the waiting queue are waiting until their periodic release timer is fired. When this happens for a particular task, it is moved to the ready queue. The tasks in the ready queue are waiting until the scheduler allows them to run on the processor. Whenever they are selected for execution, the tasks are placed on top of the run stack. Only the topmost task on the run stack is executing, while the tasks below it are preempted. When a task finished executing, it is moved to the waiting queue again. The life cycle of the task will then repeat itself.

To acquire resources in a mutually exclusive manner, a task "enters" one of its nested critical sections. When a task τ_i enters its jth NCS, it will adopt the inherited deadline level $\Delta_{i,j}$ of that NCS, and assign it to its own inherited deadline level, Δ_i. The rule sets that define DMI and EDFI then ensure that the task can acquire the resources specified in the entered NCS without interference, Jansen[8], p36 and p51:

Definition 9 (DMI protocol rules)**.** *Deadline Monotonic with Inheritance is defined by the following rules:*

1. *Released but not yet running jobs or preempted jobs are ordered by their deadline intervals D_i.*
2. *Job τ_f from the ready queue with the shortest deadline, say D_f, is selected for processor competition.*
3. *τ_f will preempt the running job τ_r iff $D_f < \Delta_r$, where Δ_r is the current inherited deadline level of τ_r.*

Note that because of the existence of nested critical sections, Δ_r is not a fixed value and can change over time.

Definition 10 (EDFI protocol rules)**.** *Earliest Deadline First with Inheritance is defined by the following rules:*

1. *Released but not yet running jobs or preempted jobs are ordered by their absolute deadlines d_i^j.*
2. *Job τ_f^j from the ready queue with the shortest relative deadline, say d_f^j, is selected for processor competition.*
3. *τ_f^j will preempt the running job τ_r^l iff $(d_f^j < d_r^l) \wedge (D_f < \Delta_r)$, where Δ_r is the current inherited deadline level of τ_r.*

Again, note that because of the existence of nested critical sections, Δ_r is not a fixed value and can change over time. Observe that these rule sets maintain an ordered run stack. This ordering is responsible for making sure resources can be acquired without interference.

3.3.1 Properties

The DMI and EDFI algorithms share several important properties, which are stated by theorem 1, 2 and 3. These properties are important to this research as several theorems and their proofs that are presented in this paper depend on them.

Theorem 1 (Blocking condition)**.** *If τ_i^k is blocked by τ_j^l then $\Delta_j \leq D_i < D_j$*

Theorem 2 (One blocker)**.** *A job τ_f^k can be blocked by at most one running or preempted job.*

Theorem 3 (Automatic mutex)**.** *The DMI and EDFI protocols guarantee mutual exclusion of shared resources.*

The validity of each of these three properties has been proved for both DMI and EDFI by Jansen[8].

3.4 Feasibility analysis

3.4.1 DMI feasibility

The DMI feasibility analysis is built around the fixed priorities of the DM algorithm. The feasibility analysis algorithm for a task set $\Gamma = \{\tau_i, \tau_{i+1}, ..., \tau_n\}$ (sorted by priority) starts by checking the feasibility of a task set that contains only the highest priority task τ_i. The maximum blocking $C_{b,i}$ task τ_i can encounter would be:

$$C_{b,i} = \max_{j=i+1..n,k} \{C_{j,k} | \Delta_{j,k} \leq D_i < D_j\} \tag{3.4}$$

If $C_i + C_{b,i} \leq D_i$ holds, ie. tau_i meets its deadline, then the task set $\{\tau_i\}$ is feasible. The algorithm then restarts and adds the second highest priority task τ_{i+1} to the task set to investigate. To check the feasibility of the resulting task set $\{\tau_i, \tau_{i+1}\}$, we only need to check if τ_{i+1} can complete before its deadline D_{i+1}. τ_{i+1} won't interfere with the execution of τ_i, as τ_i has a higher priority. It can only possibly block τ_i, but that blocking was already taken into account in the previous run of the algorithm. The task set is therefore still feasible iff: $C_{i+1} + C_{b,i+1} + \left\lfloor \frac{D_{i+1}}{T_i} \right\rfloor C_i \leq D_{i+1}$, where $\left\lfloor \frac{D_{i+1}}{T_i} \right\rfloor C_i$ is of course the load of the invocations of τ_i imposed on the execution of τ_{i+1}, and $C_{b,i+1}$ the blocking that τ_{i+1} can encounter. If the task set is still feasible, then algorithm is restarted to investigate the task set $\{\tau_i, \tau_{i+1}, \tau_{i+2}\}$ in a similar way. This process will continue until infeasibility is determined, or Γ is dubbed feasible.

Jansen[8], p42 presented a pseudo code algorithm that performed the DMI feasibility analysis process. We will use a slightly modified version of that algorithm which supports nested critical sections (as described above) and resolves some imperfections[2] that were present in the original algorithm:

Algorithm *DMI feasibility analysis*
1. $doneList \leftarrow (); interfereList \leftarrow (); todoList \leftarrow ((0,1),(0,2),...,(0,n));$
2. $schedulabe \leftarrow true;$
3. **while** $(non_empty(todoList)$ && $schedulable)$ {
4. $\quad (t, j) \leftarrow shift(todoList);$
5. $\quad putOrdered(doneList, (t, j));$
6. $\quad W \leftarrow max\{C_{k,l} | \Delta_{k,l} \leq D_j < D_k\};$
7. $\quad interfereList \leftarrow doneList;$
8. $\quad (t, i) \leftarrow first(interfereList);$
9. $\quad$ **while** $((t < D_j)$ && $((t == 0) \mid\mid (W > t)))$ {
10. $\quad\quad W \leftarrow W + C_i;$

[2]The original algorithm missed the $((t == 0) \mid\mid (W > t))$ condition shown here in line 9, which resulted in a positive feasibility outcome at time $t = 0$ for any task set

```
11.             shift(interfereList);
12.             putOrdered(interfereList, (t + T_i, i));
13.             (t, i) ← first(interfereList);
14.         }
15.         schedulable ← (W ≤ D_j);
16. }
```

An optimized version of DMI feasibility analysis algorithm was presented by Jansen as well[10]. The optimized version does not have to recalculate the interference during every iteration of the algorithm caused by the subset of tasks that was already found being feasible. It is however harder to intuitively comprehend and it would need some minor adjustments to support nested critical sections.

3.4.2 EDFI feasibility

The EDFI feasibility analysis is based around the processor demand function $H(t)$, which describes the amount of work that should be completed at a given time. $H(t)$ is defined as:

$$H(t) = \sum_{i=1}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor C_i \tag{3.5}$$

If at time a particular time L the amount of work that should be finished is greater than L, ie. $H(L) > L$, the task set can not be feasible. Liu and Layland showed[11] that a task set Γ scheduled by EDF is feasible iff:

$$\forall L : \sum_{i=1}^{n} \frac{L - D_i + T_i}{T_i} C_i \leq L \tag{3.6}$$

Theorem 4 (EDF bounded feasibility)**.** *Given equations 2.1 and 3.5, equation 3.6 can not be false anymore after a certain point L_b, also known as the Baruah bound:*

$$L_b = \frac{\sum_{i=1}^{n}(1 - \frac{D_i}{T_i})C_i}{1 - U} \tag{3.7}$$

if $L_b \neq 0$.

In other words, feasibility can be concluded after L_b if it is non-zero. While Jansen[8], p58 states correctly that equation 3.7 is valid for large values of $U < 1$, we will show that it is in fact valid for *all* values of $U < 1$. A special case occurs when L_b equals 0. Feasibility can then be concluded if $U \leq 1$. Infeasibility can be concluded otherwise.

Proof. This is a special case of the proof we will give for theorem 5. Equations 3.7 and 3.12 differ in the fact that the latter includes blocking. As blocking does not occur in proper EDF, we can define in the proof for equation 3.12 the blocking function $C_b(t)$ as $C_b(t) = 0$ for all $t \geq 0$, and define the maximum blocking C_m that can occur as being 0. The proof can then be applied to equation 3.7 as well. □

For the feasibility of a task set under EDFI, blocking has to be taken into account. Jansen [8], page 59 extends equation 3.6 to include blocking, stating that a task set Γ is feasible under EDFI iff:

$$\forall L : 0 \leq L \leq L_I : H(L) + C_b(L) \leq L \tag{3.8}$$

with

$$C_b(t) = \max_{i:1..n}\{C_i | \Delta_i \leq t < D_i\} \tag{3.9}$$

and L_I the shortest interval $[0, t]$ for which $W(t) = t$ holds, with

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{3.10}$$

Observe that 3.9 only applies to blocking for transactions. We will use a blocking equation that supports nested critical sections:

$$C_b(t) = \max_{i,j}\{C_{i,j} | \Delta_{i,j} \leq t < D_i\} \tag{3.11}$$

Without proof we state that this does not have any effect on the validity of the feasibility condition expressed by equation 3.8.

Similar to the upper bound for EDF presented in theorem 4 we can give an upper bound for the feasibility analysis of EDFI as well. Jansen[8], p58 states that for large values of U this upper bound is in fact equal to the upper bound given by equation 3.7. While correct, "large" is quite arbitrary and difficult to use in an implementation. Even more, if U is "small" the equation does not apply.
Nonetheless, the upper bound given by L_I might be improved upon for *all* values of $U < 1$, as we show in the following theorem:

Theorem 5 (EDFI bounded feasibility)**.** *Equation 3.8 can not be false after a certain point L_B:*

$$L_B = \frac{\sum_{i=1}^{n}(1 - \frac{D_i}{T_i})C_i + C_m}{1 - U} \tag{3.12}$$

with

$$C_m = \max_{i,j}\{C_{i,j} | \tau_i \in \Gamma\} \tag{3.13}$$

if $L_B \neq 0$.

In other words, feasibility can be concluded after L_B if it is non-zero.

Proof. The outline of this proof is depicted graphically in figure 3.2. Using equations 2.1, 3.5 and 3.13 we construct a linear function $f(t) = at + b$ in such a way that $f(t) \geq H(t) + C_b(t)$ holds for all $t \geq 0$. After $f(t)$ has been constructed we calculate for which value of L_B the equation $f(L_B) = L_B$ holds. If $H(t) + C_b(t) \leq t$ holds for all values of $t : 0 \leq t < L_B$, then $H(t) + C_b(t) \leq t$ can not be invalid anymore for $t \geq L_B$, as $H(t) + C_b(t) \leq f(t)$ by definition.

Following this outline, we find:

$$\begin{aligned} H(t) &= \sum_{i=1}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor C_i = \sum_{i=1}^{n} \left\lfloor 1 - \frac{D_i}{T_i} + \frac{t}{T_i} \right\rfloor C_i = \sum_{i=1}^{n} \left\lfloor C_i - \frac{C_i D_i}{T_i} + t\frac{C_i}{T_i} \right\rfloor \\ &\leq \sum_{i=1}^{n} \left(C_i - \frac{C_i D_i}{T_i} + t\frac{C_i}{T_i} \right) = Ut + \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} C_i \right) \end{aligned} \tag{3.14}$$

As $C_b(t) \leq C_m$ for all $t \geq 0$ (by definition), we can state:

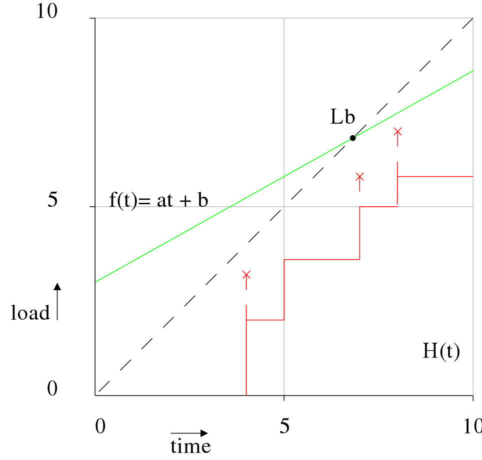

Figure 3.2: L_B proof outline

$$H(t) + C_b(t) \leq Ut + \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} C_i\right) + C_m \tag{3.15}$$

Equation 3.15 gives us a candidate for $f(t) = at + b$, where $a = U$ and $b = \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} C_i\right) + C_m$. Solving $f(t) = t$ gives us the upper bound L_B:

$$\begin{aligned} t &= Ut + \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} C_i\right) + C_m \\ t - Ut &= \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} C_i\right) + C_m \\ (1 - U)t &= \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} C_i\right) + C_m \\ t &= \frac{\sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} C_i\right) + C_m}{1 - U} = L_B \end{aligned} \tag{3.16}$$

□

Observe that a special case occurs when L_B equals 0. This can only happen when there is no blocking at all (ergo, the schedule is a normal EDF schedule) and all relative deadlines are equal to their periods. b from the processor demand upper bound function $f(t) = at + b$ equals 0 whenever this happens, reducing the function to $f(t) = at$. As we found that $f(t) = Ut$ is a suitable upper bound, it is enough to check that $U \leq 1$ to ensure the validity of equation 3.8.

Equations 3.8 and 3.12 show that there is no need to investigate feasibility beyond $min\{L_B, L_I\}$. We can thus extend the feasibility analysis algorithm presented by Jansen[8], p62 (which is a codified representation of equation 3.8) to include this condition. Including the additional support for nested critical sections we get the EDFI feasibility algorithm we will use for our implementation:

Algorithm *EDFI feasibility analysis*
1. $H \leftarrow 0; W \leftarrow 0; schedulable \leftarrow unknown;$
2. **while** $(schedulable = unknown)$ {
3. $(t, flag, C) \leftarrow GetNextEvent$
4. **case** $(flag)$ {
5. deadline:
6. $H \leftarrow H + C$
7. $C_b = \max_{j,k}\{C_{j,k} | \Delta_{j,k} \leq t < D_j\}$
8. **if** $(H + C_b > t)$ **then** $schedulable \leftarrow no$
9. release:
10. **if** $(t > 0 \wedge W \leq t)$ **then** $schedulable \leftarrow yes$
11. $W \leftarrow W + C$
12. }
13. **if** $(schedulable = unknown \wedge t > L_B)$ **then** $schedulable \leftarrow yes$
14. }

3.5 Dynamic task admission

A real-time system without the possibility to add or remove tasks at run-time can be too inflexible for practical use. The ability to add to a running real-time system is called *dynamic task admission*, or DTA in short.

Several issues have to be taken into account when allowing DTA under DMI or EDFI, some of which are design issues, and some of which are implementation issues. The design issues are discussed in this section, while implementations issues such as performance impact and scheduling overhead are discussed in the chapters 5 and 6. Conditions under which DTA is allowed will be derived using the EDFI protocol rules. Note however that these conditions hold for DTA under DMI as well. The complete breakdown for DMI will be left as an exercise to the reader.

The DTA process is complex, and involves re-executing the feasibility analysis using the combined running task set and the task set that wants to enter the system. This means that the properties of the resource usage of both task sets have to be combined, the resource floors and ceilings have to be recalculated, and the inherited deadlines of the nested critical sections have to be adjusted. If the combined task set turns out to be feasible, one has to ensure that the admission of the new tasks does not interfere with, or obstruct the current task execution.

Problem 1 (Δ level changes)**.**
One of the most obvious problems that arises when new tasks are added into an already running system is the change in inherited deadlines values of the tasks that are already in the system. To explain this problem, consider task set Γ shown in table 3.1.

Γ	D	T	C	$\Delta_{i,1}$	**Resources**
τ_1	4	5	1	4	0.1{A}
τ_2	4	6	1	4	0.5{A B}
τ_3	5	6	1	5	1.0{C}

Table 3.1: Specification of Γ

Assume a new task $\tau_4 = (3, 4, 1)$ with resource usage specification "0.2{C}" is added to Γ. As a result, the inherited deadline of τ_3 will drop from 5 to 3. The problem of recalculating the inherited deadline values and applying those the changes to every nested critical section while the system is online is considered to be an implementation issue. The details on how this can be accomplished will be discussed in chapter 5.

Problem 2 (Run stack ordering)**.**
The run stack can not be guaranteed to stay ordered if at any moment in time the inherited deadline value of a

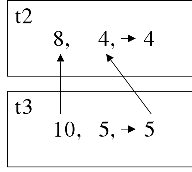

Figure 3.3: Example run stack for Γ

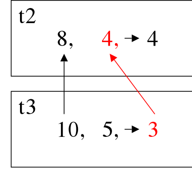

Figure 3.4: Example run stack for Γ after admission of τ_4

nested critical section can change. Consider the example run stack for Γ shown in figure 3.3.

Assume that at this moment in time τ_4 would enter the system. This would cause Δ_3 to drop from 5 to 3 if τ_3 had entered its NCS. As a result, the stack ordering would no longer be preserved, as can be seen in figure 3.4.

There are several solutions to the run stack ordering problem. The most trivial one would of course be to only allow new tasks to enter the system when the run stack is empty. No ordering has to be preserved at that point, and the admission will cause no problems. This approach is overly pessimistic though, and can be improved upon using some simple logic. Another option would be to walk down the entire run stack to see if the ordering would still be preserved if the new tasks were to be admitted at this moment in time. This however can be a very costly operation, as the scheduler overhead would become dependent on the number of tasks on the run stack. Furthermore, when the check fails, then the next time the run stack ordering must be checked one must walk down the entire run stack again. This process could be optimized slightly by remembering which task should at least disappear from the run stack before attempting another check, but clearly this is approach not desirable in a hard real-time system as well.

To come up with a better approach, we need to examine which task properties making up the run stack ordering can actually change as a result of DTA. Consider figure 3.3 displaying an example run stack. The leftmost number in each box represents the absolute deadline of a task. Clearly this value will not change as a result of DTA. The middle number represents the relative deadline of a task. Again, this value will not change as a result of DTA. Finally, as was shown in figure 3.4, the value of the rightmost number representing the inherited deadline *can* change as a result of DTA. So to solve the run stack ordering problem, we only need to deal with the possible changing of inherited deadline values. Observe that the inherited deadline can only lower as a result of DTA, thus raising the priority level of tasks it can block).

Let Γ_1 be the set of tasks that are already entered into the system and Γ_2 the set of tasks that want to enter the system. Recall that new inherited deadline values will be computed for the nested critical sections in both Γ_1 and Γ_2 during the feasibility analysis of the combined task set $\Gamma_1 \cup \Gamma_2$. Let Δ_{hidc} be the *highest inherited deadline* value that would be *changed* in Γ_1, if Γ_2 entered the system. Δ_{hidc} will be ∞ if no nested critical sections will see a drop in its Δ level.

Theorem 6 (DTA condition)**.** *The scheduler can allow DTA without breaking the run stack ordering when the following condition is met:*

$$\Delta_s \geq \Delta_{hidc} \tag{3.17}$$

where τ_s is defined as the top of the run stack.

Proof. The DTA condition will not break the run stack ordering. For the "> case" holds:

- all tasks on the run stack have a Δ level higher than Δ_{hidc} (due to the run stack ordering),

- and hence their inherited deadline will not change (due to the definition of Δ_{hidc}). □

For the "= case" holds: observe that Δ_s can drop as a result of DTA. However, no other task on the run stack will be affected by such a drop, as their inherited deadline level is strictly higher than Δ_s. Since only the task at the top of the run stack will be affected, the run stack ordering will be preserved.

3.6 Non-preemptable nested critical sections

An additional scheduler feature that might be interesting for application developers is the ability to create *non-preemptable nested critical sections* (NPNCS). This could for example be used to reduce the *jitter* when accessing resources periodically, or allow to use resources without being interrupted.

3.6.1 Problem

Suppose a task should be programmed to send a continuous, non-interrupted data stream over a radio channel. When using DMI or EDFI based on transactions, one could simply create a "virtual resource" that is added to the task with the shorted deadline interval, and to the transaction that should not be preemptable, as proposed by Jansen[8], p67.

A similar trick could be used when using DMI or EDFI based on nested critical sections[3]. You could create a virtual write resource for every NCS that needs to be non-preemptable and add that resource to every other NCS. Note however that while this approach works fine in theory, the drawbacks will likely be too big to be useful in practice. We will not discuss the drawbacks in detail, but it is easy to see that this approach will cause an increase in administrative overhead, especially when dynamic task admission and removal is allowed.

3.6.2 Theory

A different approach to allow for NPNCSs however can be easily constructed within the DMI and EDFI framework. As we will see, this approach will not have the drawbacks mentioned in the previous paragraph. We will explain how to create a NPNCS under EDFI, but the results will apply to DMI as well.

Suppose that the ith NCS of a task τ_t must be non-preemptable. Recall that with job τ_t^l at the head of the run stack and τ_f^k at the head of the ready cue, τ_f^k preempts τ_t^l iff $d_f^k < d_t^l$ and $D_f^k < \Delta_{t,i}$ (see paragraph 3.3). If the NCS should not be preemptable, we have to make sure that no job at the head of the ready queue can satisfy the preemption criterion.
The absolute deadline d_t^l can not be modified directly in such a way that the preemption criteria are never met, as it simply depends on D_t and the release time of the job. Modifying D_t is not a good option either, as that would change the priority of the whole task, instead of just a single NCS. The only variable left with influence on the preemption criteria is the inherited deadline, $\Delta_{t,i}$ in this case. Dropping $\Delta_{t,i}$ to a value lower than, or equal to D_f would have the the desired effect, since $D_f < \Delta_{t,i}$ would never hold. This however would have a negative impact on performance, as the value for $\Delta_{t,i}$ would have to be recalculated every time new task is placed at the front of the ready queue. This problem can be solved by simply dropping $\Delta_{t,i}$ all the way down to 0. This solution would have no additional impact on the performance of the scheduler. As $0 \leq D_f$ holds for any task τ_f, this inherited deadline level change would result in the desired NPNCS.

To allow for NPNCSs we redefine the inherited deadline level $\Delta_{i,j}$ for NCS j of task τ_i as:

[3]Observe that under DMI all nested critical sections of the highest priority task are automatically non-preemptable

Definition 11 (NCS inherited deadline $\Delta_{i,j}$)**.**

$$\Delta_{i,j} = \begin{cases} 0 & \textit{if NCS } j \textit{ of task } \tau_i \textit{ should be a NPNCS,} \\ min(\{D_i\} \cup \{D^r_R | R \in \phi^r_j(\tau_i)\} \cup \{D^w_R | R \in \phi^w_j(\tau_i)\}) & \textit{otherwise (unchanged from definition 3.3)} \end{cases} \quad (3.18)$$

where $\phi^r_j(\tau_i)$ is the set of resources that are read by τ_i in NCS j, and $\phi^w_j(\tau_i)$ the set of resources that are written by τ_i in NCS j.

The feasibility analysis processes for both DMI and EDFI do not need any adjustments to allow for NPNCSs if they use this new inherited deadline level definition.

3.6.3 Specification

An application programmer needs a way of specifying the requirement for one or more NPNCSs. We therefore adjusted the resource specification grammar presented in paragraph 3.2. The change involves an additional exclamation mark flag '!' that can be added to a NCS if it needs to be non-preemptable. The updated grammar becomes:

$\rho_{list} : \ float \, '\{' \ R_{list} \ \rho_{list} \ '\}' \ \rho_{list}{}^*$
$R_{list} : \ R \ R^*$
$R : \ 'a'..'z' \mid 'A'..'Z' \mid '!'$

Example 1 (Non-preemptable nested critical sections)**.**
According to NPNCS grammar, a valid resource usage description would be "2.0$\{! \ A\}$", which specifies a NPNCS in which resource A is acquired for 2.0 units of time[4]. Another valid construct would be "2.0$\{A\}$1.0$\{!\}$", which specifies a NCS which acquires resource A for 2.0 units of time, followed by a NPNCS which requests a non-preemptable section for 1.0 unit of time. As can be seen in the last example, it is not mandatory to acquire a resource in a NPNCS.

3.7 Multi-use resources

In the previous sections we assumed that resources could be read by an infinite number of simultaneous readers, and could only be written by one task at any given time. However some resources might allow to be written by more than one task at the same time, or only allow to be read by limited amount of readers simultaneously. We call such resources *multi-use resources*, MUR in short. The EDFI and DMI theory can be extended to allow for such multi-use resources.

3.7.1 Model

Every resource must specify the number of simultaneous readers and writers it allows. The notation $R_{\rho,\omega}$ specifies that resource R can simultaneously be read ρ times, and ω times written. R_ρ will be referred to as the resource's *readcount*, while R_ω will be referred to as the resource's *writecount*. Note that up till now, a resource R was assumed to be a $R_{\infty,1}$ resource. This new model does nothing more than making the model that we used up till now slightly more generic[5].

[4] Observe that the resource description "2.0$\{! \ A\}$" will give the same result as "2.0$\{!\}$", as in both cases the inherited deadline level will be set to 0. However, we do not restrict the resource usage specification to the use of either resources or a "!" flag, as the result can differ when using multi-use resource support, a theory which will be discussed in section 3.7.

[5] The extent to which this model is useful has yet to be determined. This paper will present two examples where we think this model provides an advantage over the existing model. We expect that this model could prove useful in far more real-life situations, which have yet to be explored. On the other hand this model provides a flexibility that might be unneeded in practice. For example, we do not know how a $R_{4,5}$ or $R_{22,3}$ resource would function in practice, what use it might have or if even if it could be created in the first place.

As before, readers and writers can not access a resource simultaneously. When a reader acquires a resource, only other readers (if any) can acquire the same resource. Similarly, when a writer acquires a resource, only other writers can acquire the same resource. A task is assumed to only acquire a resource once. The reason for the existence of this assumption will be discussed in paragraph 7.1.2.

Example 2 (MUR usage)**.**
Consider a classical tape device used for storing large amounts of data on tape drives. The seek time of such devices is typically very large; seek times of several minutes are not rare. Since seek times are so large, we do not want the read actions of several tasks interfering with each other. While these actions could be coordinated on a level above the scheduler, this would require effort from the application developer. The MUR model offers a solution at scheduler level. In this case we could simply model the tape device as $T_{1,1}$. Multiple readers (and writers) would now automatically be stopped from interfering with each other by the scheduler.

Another use case is inspired by the Casini-Huygens spacecraft[6]. This craft carried a single radio device that could transmit data back to earth over two separate channels. These two channels could be modeled as separate resources, or the radio could be modeled as a single $R_{0,2}$ resource. The first approach would imply a fixed assignment of channels to tasks which might be suboptimal during operation: even if one data channel is still available, a task can be blocked from sending its data back to earth if its assigned channel is already in use. The single $R_{0,2}$ resource approach would not have this problem, as it would always allow two simultaneous writers.

3.7.2 Specification

An application programmer needs a way of specifying the resource read and writecount. We therefore adjusted the resource specification grammar presented in paragraph 3.6.3. The change involves an optional section '['ρ,ω']' that can be added to every resource to specify its read and writecount. The updated grammar becomes:

$$\rho_{list} : \ float \ '\{' \ R_{list} \ \rho_{list} \ '\}' \ \rho_{list}{}^{*}$$
$$R_{list} : \ R \ ('['C,C']')^{?} \ \ R_{list}{}^{*}$$
$$R : \ 'a'..'z' \mid 'A'..'Z' \mid '!'$$
$$C : \ 'inf' \mid integer$$

The resource usage description "1.0 { R[inf,2] }" would for example specify a resource R that has an infinite readcount and a writecount of 2. If for a resource R no read and writecount are specified, $R_{\infty,1}$ is assumed.

3.7.3 Sufficient feasibility analysis

The read and write floor definitions presented in paragraph 3.2.1 will have to be adjusted to support MUR. First of all, observe that when the number of tasks that write resource R is less than or equal to R_{ω}, no blocking will ever occur on this resource for reading. Similarly, when the number of tasks that read resource R is greater than R_{ρ}, readers will be blocked over this resource. Previously. readers of a resource would never be blocked by other readers of the same resource and writers would always be blocked by readers as well as writers.

This results in new definitions for the read floor D^r_R and write floor D^w_R of a resource R :

Definition 12 (Read floor $D^r_{R_{\rho,\omega}}$)**.**

$$D^r_{R_{\rho,\omega}} = \begin{cases} min(\{D_i|\tau_i \in \gamma^{w}(R)\} \cup \{\infty\}) & \textit{if } \mu^r(R) \leq \rho, \\ min(\{D_i|\tau_i \in \gamma^{w+r}(R)\} \cup \{\infty\}) & \textit{otherwise} \end{cases} \tag{3.19}$$

where $\mu^r(R)$ is the number of readers of R.

[6] Cassini-Huygens was an international collaboration between NASA, ESA and the Italian Space agency. The Cassini spacecraft's mission was to explore the Saturn system of rings and moons from orbit, while the Huygens Probe was to dive into Titan's thick atmosphere. See http://saturn.jpl.nasa.gov/overview/index.cfm for details.

Definition 13 (Write floor $D^w_{R_{\rho,\omega}}$).

$$D^w_{R_{\rho,\omega}} = \begin{cases} min(\{D_i|\tau_i \in \gamma^r(R)\} \cup \{\infty\}) & \textit{if } \mu^w(R) \leq \omega, \\ min(\{D_i|\tau_i \in \gamma^{r+w}(R)\} \cup \{\infty\}) & \textit{otherwise} \end{cases} \tag{3.20}$$

where $\mu^w(R)$ is the number writers of R.

The inherited deadline calculation as described in paragraph 3.2.1 can be left unmodified if used with these new resource floor definitions. The feasibility analysis processes presented in section 3.4 can also be left unmodified, as these read and write floor definitions are worst-case conditions. Ie. when the number of readers of $R_{\rho.\omega}$ exceeds ρ or the number of writers exceeds ω, all readers and writers are taken into account. Only when the number of simultaneous readers or writers are less or equal than ρ and ω respectively, a resource is omitted from the feasibility analysis.

Example 3 (Sufficient condition resource floors).
Consider the example task set shown in table 3.2 along with table 3.3 showing the resource definitions. Observe that the number of writers of A is greater than its writecount. This means that the write floor of A, D^w_A, will be set to 4, with or without MUR. Similarly, A_ρ equals ∞ which is greater than the number of tasks that read A (only τ_3). This means that D^r_A will be set to 4, with or without MUR.
Things get different when looking at resources B and C. B_ω is specified as being 2 and the tasks that write B are τ_2 and τ_4. Since the number of tasks that write resource B does not exceed B_ω and no task reads B, D^w_B will be set to ∞. Without MUR D^w_B would have been set to 4 (from τ_2).
Resource C is read by τ_3 and τ_4 and no task writes C. As C_ρ equals 1, D^r_C will be set to 5 (from τ_3), where it would have been set to ∞ without MUR.

Γ	D	T	C	$Resources$
τ_1	4	5	1	0.1{A}
τ_2	4	6	1	0.5{A B}
τ_3	5	6	1	1.0{a c}
τ_4	6	9	3	2.0{A} 1.0{B c}}

Table 3.2: MUR example task set

$Resource$	ρ	ω
A	∞	2
B	∞	2
C	1	1

Table 3.3: MUR example resource specifications

3.7.4 Necessary feasibility analysis

The resource floor definitions presented in the previous paragraph are too pessimistic for several common MUR applications. Consider for example three tasks that all have one NCS with the following specification: “1.0 { A[inf,2] B }”. No task will be able preempt another task that has entered its NCS, even though resource A has a writecount of 2. This is caused by the fact that every NCS also needs to acquire resource B for writing, which has a writecount of only 1. The sufficient feasibility analysis will assign a write floor to A equal to the shortest deadline of the tasks using it, while in practice is could be set to ∞.

The solution to finding the optimal resource floors is a NP-complete problem, as it involves generating all possible NCS stack orderings and checking which stacks are actually valid. The worst case blocking that could actually occur at run-time can then be determined by looking at the nested critical sections at the top of the

valid stacks. We propose the following "*Optimal resource floors*" algorithm to determine the optimal resource floor values:

Algorithm *Optimal resource floors*
1. $todoList \leftarrow sortDesceding(tasks);$
2. **for** $(R \in resourceList)\{D^r_R \leftarrow \infty; D^w_R \leftarrow \infty\ \}$
3. **while** $(non_empty(todoList)\ \{$
4. $\quad t \leftarrow shift(todoList)$
5. $\quad ncsStacks \leftarrow generateStacks(t, todoList)$
6. $\quad$ **for** $(s \in ncsStacks)\ \{$
7. $\quad\quad$ **if** $(isValid(s))\ \{$
8. $\quad\quad\quad$ **for** $(R_{\rho,\omega} \in resourceList)\ \{$
9. $\quad\quad\quad\quad (\tau_i, ncs_j) \leftarrow top(s)$
10. $\quad\quad\quad\quad$ **if** $((D_i < D^r_R) \wedge (curReaders(s, R) = \rho))$ **then** $D^r_R \leftarrow D_i$
11. $\quad\quad\quad\quad$ **if** $((D_i < D^w_R) \wedge (curWriters(s, R) = \omega))$ **then** $D^w_R \leftarrow D_i$
12. $\quad\quad\quad \}$
13. $\quad\quad \}$
14. $\quad \}$
15. $\}$

The $generateStacks()$ function is the hardest part of this algorithm to implement. It must generate all NCS stacks with task t placed at the bottom, and using the remaining tasks in the $todoList$ to construct all possible combinations of NCS stacks on top of that. The $todoList$ is sorted so $generateStacks()$ can easily make sure that no tasks with longer relative deadline are stacked upon a task with a shorter or equally long relative deadline; the stack ordering still needs to be preserved.
The $isValid()$ function verifies that a given NCS stack could actually be constructed at runtime. As $generateStacks()$ makes sure the relative deadline ordering of the items on the stack is sound, $isValid()$ only has to verify that resources on the stack are used properly. To be able to do this, $isValid()$ has to know how resource floors and inherited deadline values are calculated at run-time. This process is described in paragraph 3.7.5.
When a stack has been determined to be valid, the algorithm checks whether a resource acquired by the NCS at the top of the stack is maxed out in terms of simultaneous readers or writers. If that is the case then the read or write floor of that resource is set to the deadline of the task belonging to that NCS. This will only be done if the new value is smaller than any previous found value.
When the algorithm completes, all resources will have been assigned their worst case read and write floors that can actually occur at run-time. Using these values during the feasibility analysis will definitively determine the feasibility of a task set, this in contrast to the sufficient condition described in the previous paragraph.

3.7.5 Online operation

The online operational changes to support for multi-use resources are rather small. The most important change is the fact that the value of the resource floors, and thus the value of the inherited deadline of a NCS, now depends on the availability of the free read or write "units" of those resources. In a resource's read floor only readers are taken into account when the current number of readers is equal to the resource's readcount. By "current" the moment in time is meant when the inherited deadline value of a NCS is being calculated, that is on entering a NCS. Similarly for the write floor only writers are taken into account when the current number of writers is equal to the writecount[7].

The following read and write floor definitions, D'^r_R and D'^w_R, take the current number of readers and writers into account:

[7]Observe that the current number of readers or writers of a resource can not exceed the resource's readcount or writecount respectively; there can't be more read or write units of a resource handed out than those that are available.

Definition 14 (Online read floor ${D'}^{r}_{R_{\rho,\omega}}$)**.**

$$
{D'}^{r}_{R_{\rho,\omega}} = \begin{cases} min(\{D_i | \tau_i \in \gamma^{w+r}(R)\} \cup \{\infty\}) & \text{\textit{if} } (\mu^r(R) > \rho) \wedge (\nu^r(R) = \rho), \\ min(\{D_i | \tau_i \in \gamma^{w}(R)\} \cup \{\infty\}) & \text{\textit{otherwise}} \end{cases} \tag{3.21}
$$

where $\mu^r(R)$ is the number of readers of R and $\nu^r(R)$ is the current number of readers of R.

Definition 15 (Online write floor ${D'}^{w}_{R_{\rho,\omega}}$)**.**

$$
{D'}^{w}_{R_{\rho,\omega}} = \begin{cases} min(\{D_i | \tau_i \in \gamma^{r+w}(R)\} \cup \{\infty\}) & \text{\textit{if} } (\mu^w(R) > \omega) \wedge (\nu^w(R) = \omega), \\ min(\{D_i | \tau_i \in \gamma^{r}(R)\} \cup \{\infty\}) & \text{\textit{otherwise}} \end{cases} \tag{3.22}
$$

where $\mu^w(R)$ is the number of writers of R and $\nu^w(R)$ is the current number of writers of R.

Observe that both cases in the read and write floor definitions can be pre-calculated offline. Given these definitions we can redefine the definition for the inherited deadline level to make its value depend on the online resource read and write floors:

Definition 16 (NCS inherited deadline $\Delta_{i,j}$)**.**

$$
\Delta_{i,j} = min(\{D_i\} \cup \{{D'}^{r}_{R} | R \in \phi^r_j(\tau_i)\} \cup \{{D'}^{w}_{R} | R \in \phi^w_j(\tau_i)\}) \tag{3.23}
$$

These new definitions imply that the resource usage has to be tracked at run-time, which was not needed previously. While the run-time changes to allow for MUR are small and easy to implement, they will have a negative impact on the scheduler performance. In chapter 6 a test will be described that measures the performance impact imposed by MUR on the scheduler.

Chapter 4

Design

4.1 Platform determination

A good number of real-time operating systems exist today, such as VxWorks[1], eCos[2], QNX[3] and various Linux based systems. The Linux based systems are generally availble under the GPL[4] license, which makes them suitable candidates for implementing and testing new scheduling methods.
Two main competitors (if you could call them that) based on Linux exist today, namely *RT-Linux*[5] and *RTAI*[6]. RT-Linux is a hard real-time kernel with a minimalistic approach. It does not include the numerous features that commercial operating systems such as VxWorks carry. RTAI (Real-Time Application Interface), originally based on RT-Linux, is a hard real-time extension to the Linux kernel. It incorporates a whole array of real-time features and is generally more complex than RT-Linux.
Other Linux based real-time operating systems include RedIce-Linux, Linux/RT, ART Linux, KURT and Linux/RK. They all have a different goal and/or means to reach that goal.
After an evaluation of the available real-time operating systems, RT-Linux has been chosen as the testbed our the DMI and EDFI implementations. Its licensing, minimalistic approach, and its hard real-time properties make it an ideal platform for this purpose.

4.2 Real-Time Linux workings

The RT-Linux kernel is a small, hard real-time kernel separate from the standard Linux kernel. Figure 4.1 gives an overview of the RT-Linux architecture. It contains a scheduler which schedules the real-time tasks. Real-time tasks are generally created in normal Linux kernel modules, and are thus running in kernel space. Interestingly, the scheduler treats the Linux kernel as a real-time task as well. The Linux kernel is artificially made a real-time task by setting its period to infinity and giving it the the lowest priority possible. This effectively means that the Linux kernel only runs when no real-time activity takes place. To accomplish this, the RT-Linux kernel requires a modification to the Linux kernel interrupt handling routines and the interrupt enabling/disabling macros. Whenever an interrupt is raised, it is trapped by the RT-Linux kernel. If the interrupt is related to any real-time activity, the RT-Linux kernel will handle it. Non real-time related interrupts are blocked until no real-time activity takes place. At this point, the Linux kernel will be executed and the queued interrupts can be handled.

[1] http://windriver.com/
[2] http://www.redhat.com
[3] http://www.qnx.com/
[4] http://www.fsf.org/licensing/licenses/gpl.html
[5] http://www.rtlinux-gpl.org/
[6] http://www.rtai.org/

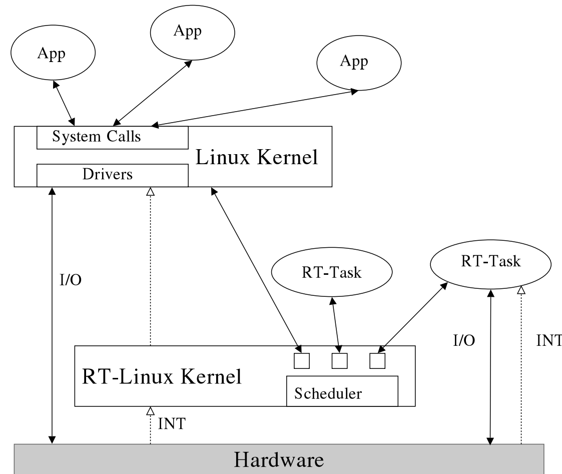

Figure 4.1: Real-Time Linux architecture

This “mini kernel” approach has some important advantages, such as a very low latency and reduced complexity. On the other hand it presents some drawbacks, such as faults in real-time tasks being able to bring down the kernel (since they run in kernel space) and the inability to use the device drivers that come with the Linux kernel. A detailed technical overview of the workings of RT-Linux has been given by Yodaiken[20].

4.2.1 Real-Time Linux scheduler

RT-Linux contains by default a simple, priority based scheduler. It does not support resource scheduling, and as such does not have any priority inversion or chained blocking prevention mechanisms. The task model it uses is very simple as well, only supporting the periodic execution of tasks. Tasks do not have any deadlines, and are just assumed to be finished before the next period starts. A deadline overrun detection mechanism is therefore not existent either.

The real-time tasks, in varying states, are placed in a single task list by the scheduler. An example task list is shown in figure 4.2. Each task can be in one of the RUNNING, RELEASED, WAITING or PREEMPTED states at any moment in time when the default scheduling algorithm is used. These states denote:

- RUNNING: the task is currently executing
- RELEASED: the task is ready to run on the processor (its periodic release timer fired)
- WAITING: the task is waiting until its periodic release timer fires again
- PREEMPTED: the task’s execution is interrupted by a higher priority task

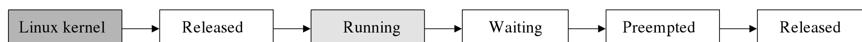

Figure 4.2: RT-Linux task list example

As can be seen in figure 4.2, the first task in the list is the pseudo real-time ’Linux kernel’ task. As this task is always present, it will be automatically be scheduled when there are no real-time tasks in any of the RELEASED, PREEMPTED or RUNNING states.

The rationale behind the simplicity of the scheduler is that most real-time projects require a very distinct set of features from the scheduler. As it is impossible to suit them all, the default scheduler is left simple. This makes the scheduler easy to adapt to the specific needs of the projects that use it.
The opinion of the principal author of RT-Linux, V. Yodaiken, might have had some influence as well in keeping several complex mechanisms out of the default scheduler. Yodaiken is for example a strong opponent of using priority inheritance mechanisms in a real-time scheduler to ensure mutual exclusion, one of reasons being “the significant amount of complexity that is added to the operating system”[21]. We do not share this view and will show that adding inheritance (as used by DMI and EDFI) will add very little additional complexity to the system.

4.3 DMI and EDFI in RT-Linux

To fit the DMI and EDFI schedulers into RT-Linux, we need to adapt the scheduler organisation presented in paragraph 3.3 to work in the RT-Linux framework. We can not copy the organisation verbatim, as it is not optimal in terms of implementation performance and it does not, obviously, deal with the fact that there should always be a Linux task scheduled when there are no real-time tasks waiting to be executed.

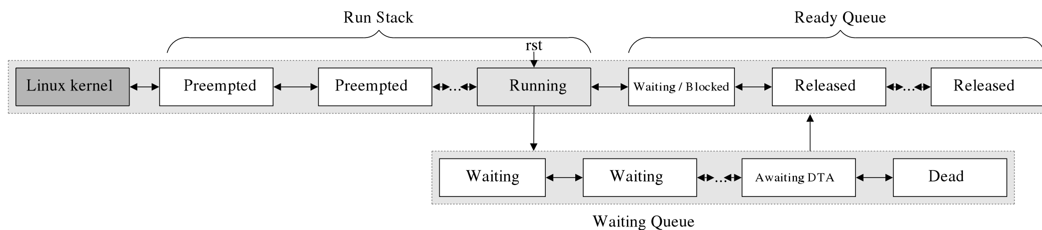

Figure 4.3: DMI / EDFI task list organisation

The design that has been adopted for both DMI and EDFI in RT-Linux is shown in figure 4.3. The run stack and the ready queue are still separate logical entities, but physically they are concatenated using a double linked list structure. Starting at the Linux kernel task, the task priorities increase all the way up to the task pointed at by the `rst` pointer. The tasks in the ready queue on the other hand have a decreasing priority from the left to the right[7].

Each task has to maintain a pointer to its successor called `next`, and a pointer to its predecessor called `prev` given the linked list structure of the task list. A *run stack top* pointer, or `rst` in short, is used to keep track of the top of the run stack.

A task can be in any of the following states: RELEASED, WAITING, PREEMPTED RUNNING, INITIALIZING, DELETING, AWAITING DTA and DEAD. The first four states were already depicted in figure 3.1, and should need no further explanation. The other remaining four states are:

- INITIALIZING: the task is in the process of initializing itself; this is typically done in idle time. It is not being scheduled periodically yet. See paragraph 4.3.6 for details.
- DELETING: the task is in the process of being deleted; this is typically done in idle time. See paragraph 4.3.6 for details.
- AWAITING DTA: the task is initialized and is waiting for the DTA condition to become valid. When this happens, the task is released into the system and will be scheduled periodically[8].

[7] The only difference between the DMI design and the EDFI design is the way in which the ready queue is ordered; see section 3.3 for details.

[8] Tasks in the AWAITING DTA state are not placed in the ready queue to keep the normal scheduling decision as fast as possible. If

- DEAD: the task has missed a deadline and is excluded from execution. See paragraph 4.3.4 for details.

A state diagram showing the states a task can be in, as well as the possible transitions between those states is depicted in figure 4.4.

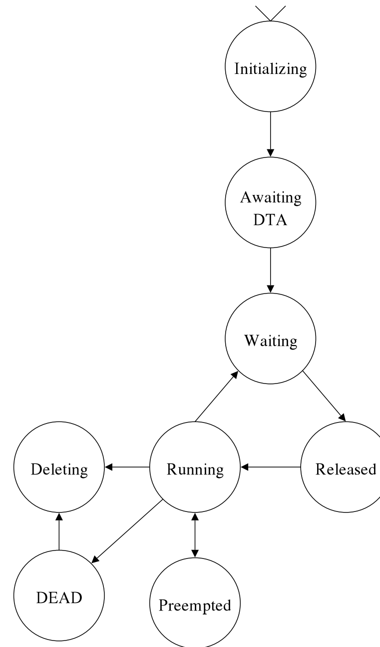

Figure 4.4: Task states and transitions

4.3.1 Task preemption

The `rst` pointer allows the scheduler to make a preemption decision very quickly. To make this decision, only the task at the top of the run stack and the task at the front of the ready queue are needed. The top of the run stack can instantly be accessed using the `rst` pointer, and the task at the front of the ready queue can simply be reached by dereferencing the `next` pointer of the task pointed at by `rst`.

4.3.2 Task completion

Whenever a task completes its execution, it is moved to the waiting queue. This queue has no ordering at all, and as such, tasks can simply be added to the front of the list. This is obviously an $O(1)$ complex operation.

Before the task is moved to the waiting queue, the `rst` pointer must be set to the `prev` pointer of the completed task. This ensures that the scheduler will use the correct tasks in the preemption test when it makes the next scheduling decision. If the task at the front of the ready queue is not selected for execution, the scheduler will notice that the `rst` pointer has changed, and will wake up the preempted task that is now at the top of the run stack.

Observe that this mechanism will automatically ensure that the Linux kernel when executed if the run stack and the ready queue are empty.

tasks awaiting DTA would be placed in the ready queue, then the scheduler can't simply follow the `next` pointer of the task pointed at by `rst` anymore to make a scheduling decision; the task pointed at might not be eligible to run if it is awaiting DTA. In a worst case scenario the whole ready queue has to be traversed to look for a ready task that can be used to base a scheduling decision on.

4.3.3 Task releasing

Tasks have to be released periodically, which involves moving them from the waiting queue into the ready queue. The ready queue is an ordered list, and inserting the released task into the list must not invalidate this ordering. The linked list structure will make this operation $O(n)$ complex, where n is the number of tasks in the ready queue.

Inserting an element into an ordered list can be done in $O(log(n))$ time, so the proposed design with its $O(n)$ complexity is not optimal. However, if one wants to achieve $O(log(n))$ complexity, the ready queue must be a continuous array in memory when it is going to be implemented. Since an array is a rigid structure, we deem it unfit to for use in the proposed scheduler architecture. The benefits of the reduced algorithmic complexity do not outweigh the drawbacks of the increased resource usage and the array management overhead, such as making space for new tasks when the array is full and preventing gaps from occurring.

4.3.4 Missing deadlines

The solution to the question of what action to undertake when a job exceeds its deadline is a hard one. Should the whole system stop? Should the task be removed from the system and should the system try to continue as best as it can? Or even something else? As the right answer might be different for different types of task sets, or might not even exist at all, we have chosen for a solution that seems “good enough“ for most cases. Note that “good enough“ is quite subjective and could use some more insight, especially when the design is used outside the confined walls of the laboratory.

Whenever a task exceeds its deadline, it will be removed from the run stack and will be flagged DEAD. It will then be moved into the waiting queue, where it will stay until the application designer removes it from the system. It will then leave the system via the DELETING state as depicted in figure 4.4.

Note that a problem arises when a task that exceeds its deadline holds any resources. It will need to be signalled to release its resources immediately. Only after all acquired resources are released the task can be flagged DEAD. A proper implementation will only send such a signal if needed, as the scheduler knows when any resources are actually acquired.

4.3.5 Nested critical sections

Each NCS is modeled as a separate entity which holds several properties. These properties include:

- an inherited deadline,
- the set of resources that are read and/or written,
- the amount of time the resources are acquired,
- a pointer to a possible embedded NCS called the *child NCS*, and
- a pointer to a possible successive NCS which we will refer to as the *next NCS*.

An example NCS entity diagram is shown in figure 4.5. This particular diagram shows the NCS structure for the resource description: “$2.0\{radio\}3.0\{ram1.0\{DISK0.5\{NETWORK\}\}1.2\{flash\}\}1.0\{RADIO\}$”. Note that not all NCS properties are depicted in the figure.

Given the structure of the nested critical sections, every real-time task only has to maintain a single pointer to the first NCS entity it will access. Using that NCS as a starting point (the '*radio*' entity in the example shown in figure 4.5) the scheduler is able to traverse the entire tree via the various “c”hild and “n”ext pointers. This allows the application designer to simply tell the scheduler to enter the next NCS, or leave the current NCS, without the need to explicitly state which NCS to enter of leave: the scheduler knows the NCS structure and can figure it out on its own.

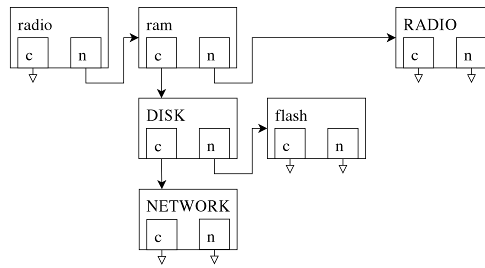

Figure 4.5: NCS structure example

4.3.6 Idle time

Several processes, such as the feasibility analysis, task initialisation and task deletion must be executed without interfering with the execution of real-time tasks. This can be achieved by:

- reserving a fixed amount of processing time (for example by creating a virtual real-time task) to execute the non real-time processes in
- executing the non real-time tasks in idle time[9]

The former will have a negative impact on scheduling performance, and the feasibility analysis could mark otherwise feasible task sets as being infeasible because of the reserved processing time. The latter solution is preferable, as it does not have these problems. However, depending on the implementation it might not be possible to execute non real-time tasks without any interference with the real-time task execution.

Fortunately, given the architecture of RT-Linux and the scheduler design depicted in figure 4.3 it is quite easy to execute tasks in idle time. In fact, there are two distinct ways to execute non real-time work:

1. The RT-Linux architecture allows the Linux kernel to be handled and scheduled as if it is normal real-time task. The DMI/EDFI design makes sure the Linux kernel is only scheduled when the run stack and the ready queue are empty (by giving it by definition the lowest priority). Ergo, the Linux kernel is only executed in idle time. By executing the feasibility analysis process from within the Linux kernel it is automatically executed in idle time.
2. Observe that the bottom of the run stack is the task directly following the Linux kernel task (see figure 4.3). Similarly to the Linux kernel, initialising tasks or tasks that are in the process of begin deleted are also given by definition the lowest priority possible. They are then placed directly after the Linux kernel as is depicted in figure 4.6. This will ensure that they are only executed in idle time.

A task that is executing in idle time, the Linux kernel included, will immediately be preempted by the scheduler whenever a real-time task is placed in the ready queue. The newly arrived task will by definition have a higher priority, and thus preempt the task that is executing in idle time.

[9]Idle time occurs where there are no real-time tasks awaiting execution

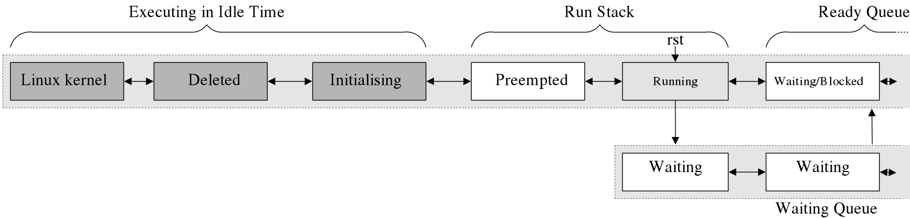

Figure 4.6: Example of tasks executing in idle time

Chapter 5

Implementation

In this chapter we will describe the implementation of the DMI and EDFI schedulers and the accompanying admission control mechanisms in RT-Linux. We will discuss most of the implementation on a relatively high level, as the source code details are of less interest to this paper. Several utilities, listed in Appendix B, have been created during the implementation process to ease the development of the schedulers.

5.1 Application Programming Interface

Real-time tasks in RT-Linux are modelled as POSIX[1] threads, commonly known as *pthreads*. Conveniently, most of the normal POSIX thread APIs can be applied to them. RT-Linux also implements the minimal POSIX-RT standard. This standard includes the specification of POSIX timer and signaling interfaces. Our implementation tried to keep these interfaces working as expected, but this hasn't been tested extensively.

RT-Linux has expanded the POSIX-RT interface with new functions to create and control real-time tasks. Similarly, we modified some of these additions and created several new ones ourselves as well. The most important additions are shown below:

- `rtl_task_create()`: construct a new real-time task
- `rtl_admctrl_admit_if_feasible()`: admit a set of tasks if the addition results in a feasible system
- `rtl_task_free()`: frees all resources held by a task
- `pthread_ncs_enter()`: enter the next nested critical section
- `pthread_ncs_leave()`: leave the current nested critical section

Obviously these are not the only functions we had to add to the kernel. Quite a number of non-public functions were needed to make these new interfaces work. The rather complex mechanisms behind the seemingly simple `rtl_task_create()` and `rtl_admctrl_admit_if_feasible()` interfaces will be discussed in paragraph 5.3 and 5.4 respectively. The operation of the other new functions will be briefly addressed in this section as well.

Appendix A gives a simple example of how these interfaces can be used to create a real-time task, how to insert the task into the system if the feasibility analysis allows it and finally how to remove it from the system again. The source code with its included comments should speak for itself.

[1]POSIX, short for Portable Operating System Interface, is developed by the Institute for Electrical and Electronic Engineering. Its purpose is to define the interface that a conforming operation system should offer to applications.

5.2 Data structures

Some of the most important data structures are (partially) displayed in figure 5.1 that can be found at the end of this chapter[2]. These data structures give some idea of how the properties of tasks, nested critical sections, resources and the scheduler are stored and related in our implementation. Throughout this chapter we will explain and refer to these data structures.

5.3 Task creation

A new task, or to be more precise a new `rtl_task_struct` structure, can be created using the `rtl_task_create()` function. This function accepts several arguments, the most important being:

- `period`: the task's period (in ns)
- `deadline`: the task's relative deadline (in ns)
- `load`: the task's load (in ns)
- `resouce_usage`: the task's resource usage description

The first three variables should speak for themselves. All arguments are of the type `hrtime_t`, which is basically a C `long long`. The values must be specified in nanoseconds, as that are the time units the scheduler works with.
The `resource_usage` parameter is a simple character string. The full resource specification grammar presented in paragraph 3.7.2 can be used to describe the task's resource needs. A valid, rather exotic resource string could for example look like: "1000{ ! } 500 { RADIO[0,2] network}". Strings not conforming to the grammar will be refused. Strings that do conform to the grammar are automatically transformed into a NCS structure (see paragraph 4.3.5) using the internal `__rtl_admctrl_parse_ncs()` function. This structure is then stored in the `ncs` member of the newly created `rtl_task_struct`, ready to be used by the feasibility analysis algorithm.

An interesting detail is the fact that resources are reference counted. All currently used resources are stored in the scheduler's `rtl_resources` list. The `ref_count` member of a `rtl_resource_struct` denotes the number of nested critical sections that use that resource. The reference count is automatically incremented when the `__rtl_admctrl_parse_ncs()` function determines that a resource is being parsed that is already present in the system. A new `rtl_resource_struct` with an initial reference count of 1 is created when a new resource is encountered.
When a task is deleted from the system, all its nested critical sections will be destroyed. The resources referenced by those nested critical sections will automatically have their reference count decremented. When the reference count reaches 0, the resource is destroyed as well.

5.4 Admission control

The complexity of the admission control mechanism is hidden behind the simple `rtl_admctrl_admit_if_feasible()` function. We will only briefly discuss the overall workings of this function, as the rather awful details of the implementation are outside the scope of this paper[3].

[2]Note that not all variables are shown in figure 5.1. For example the `rtl_resource_struct` structure will contain additional member variables when MUR support is enabled, such as the current number of readers and writers.

[3]One of these details for example has to do with the Baruah bound calculation used in the EDFI feasibility analysis. Because floating point calculations are not by definition supported by the kernel, the value of equation 3.12 had to be calculated using fixed point math.

When `rtl_admctrl_admit_if_feasible()` is called with a list of new tasks (`rtl_task_struct`'s), the first thing this function will do is combining the resources used by the new tasks with those already present in the system. Then it will recalculate all resource floors of all resources. If multi-use resources are supported, the application designer can specify if the sufficient or necessary resource floor conditions should be used.
When the resource floor calculation is completed, the inherited deadline value of every NCS (for both the new ones and for those already present in the system) will be (re)calculated. Then with these values at hand, the feasibility analysis for DMI or EDFI can be performed. Based on the kernel configuration the correct algorithm is automatically executed. When the combined task set has been found to be feasibly, a `pthread` is created and assigned to every `rtl_task_struct`. This new `pthread` is picked up by the scheduler and handled as described in section 4.3.
The hard part of implementing the admission control mechanism was making it function correctly while the system was currently executing. The current implementation fully supports online operation.

5.5 NCS usage

Only two functions are needed to traverse the NCS structure:

- `pthread_ncs_enter()`, and
- `pthread_ncs_leave()`

Whenever a task makes a call to `pthread_ncs_enter()` the scheduler will automatically enter the next NCS in line. This can be either a "child" NCS or a "next" NCS (see paragraph 4.3.5 and figure 4.5). The currently entered NCS is left again when a call is made to `pthread_ncs_leave()`. The scheduler will determine itself which "parent" NCS to return to. If no parent NCS exists, the task will set the inherited deadline level to the relative deadline of the task itself (see figure 5.2). In paragraph 5.6.2 we will discuss this process in more detail.
The application designer does not have to maintain any administration to traverse the nested critical sections of a task. He should only make sure that the call pattern of the `pthread_ncs_enter()` and `pthread_leave_enter()` functions follows the structure of the resource usage description initially passed to `rtl_task_create()`. The scheduler will report an error if this structure is not followed.

5.6 Performance considerations

5.6.1 Signaling

Release and deadline timers are all implemented in software, ie. they are not interrupt driven. Every time the scheduler is executed, which could either be caused by the periodic timer interrupt or by a direct call to the scheduler due to a state change, the following timer related events are handled:

1. If the task at the top of the run stack is not the Linux kernel, the task is checked for a deadline overrun. Ie. the current time should be "less" than the task's release time + relative deadline.
2. Every task in the waiting queue is checked to see if it should be released.

The first event is obviously an $O(1)$ operation, while the latter is $O(n)$, with n being the number of tasks in the waiting queue. As RT-Linux runs on "standard" hardware such as x86 and PowerPC systems, there are no dedicated hardware timers available which could simplify and/or speed up signaling.

Furthermore, since all task properties are specified in nanoseconds (see paragraph 5.3), special care had to be taken to prevent integer calculation overruns. The admission control mechanism is full of such pesky details.

5.6.2 NCS traversal

During the implementation we found that the NCS structure presented in paragraph 4.3.5 could not be traversed easily. Observe that the presented structure only allows a depth first search to find which NCS is the next to enter on a `pthread_ncs_enter()` call, or to which NCS to return after on a `pthread_ncs_leave()` call. Adding an "entered" flag for example to every NCS entity to remember which NCSs have already been visited would be a suboptimal solution: whenever a traversal of the complete structure has been completed the flags would have to be reset again. This is an $O(n)$ operation, where n is the number of NCS entities.

Our solution involves adding additional `enter` (green) and `leave` (red) pointers to each NCS entity as shown in figure 5.2. These pointers are already created during the `rtl_task_create()` call (see paragraph 5.3), so no run-time overhead is added. This new structure allows for a $O(1)$ traversal of the NCS structure: whenever a task enters a NCS, the scheduler will store the `enter` pointer of that NCS. The next time a `pthread_ncs_enter()` call is made, the scheduler will simply follow the stored `enter` pointer to enter the proper NCS. Whenever a `pthread_ncs_leave()` call is made, the scheduler will simply follow the `leave` pointer of the currently entered NCS.

5.6.3 Avoiding overhead peaks

Recall that the specified load for each task (and NCS) is a *worst case* value. This value is constructed based on the actual computing power that the task needs, the scheduling overhead and various other aspects. This value is then used during the feasibility analysis process.

Our implementation directly influences the scheduling overhead and we should therefore make sure that its behavior is deterministic (within bounds) and that it does not behave too erratic. Ie. the worst case behavior should not include high "peaks". Overhead peaks generally occur when changes are made to the system, such as the admission or removal of tasks. To prevent overhead peaks our implementation will not apply every change immediately.
For example, the NCS inherited deadline values of tasks that are already in system are recalculated during the admission control process. The changed values will be flagged "dirty" (hence the `flags` member of the `ncs_resources` struct shown in figure 5.1) and will not be adopted immediately when the DTA condition holds. Instead the change is applied the next time the NCS is entered.
Exactly the same holds for the adoption of recalculated online read and writefloors of resources when MUR support is enabled (see paragraph 3.7.5).

5.6.4 Latency over overhead

When tasks are removed from the system, the inherited deadlines of nested critical sections belonging to the tasks still in the system might be allowed to increase again. Our implementation however does not recalculate the resource floors and inherited deadline levels on task removal, as it is not strictly needed.
Observe that the remaining tasks will still form a feasible task set and jobs will still meet their deadlines. By not recalculating the inherited deadline levels we prevent additional overhead on the scheduler to occur. Whenever new tasks enter the system, the feasibility analysis will have recalculated all values which will then be gradually be adopted in the system (see paragraph 5.4 and 5.6.3).
The drawback of this approach is that tasks could be denied to run at a particular time, while they would have been allowed to run using the recalculated inherited deadline levels. This means that some tasks could have a latency that is higher than strictly needed.

5.6.5 Compiler based optimizations

During the development of our scheduler several experimental compiler based optimizations were added to the default RT-Linux scheduler. These optimizations included branch prediction improvements for P6 family

systems and the use of the GCC compiler specific macros such as `likely`/`unlikely`. These macros can be used to inform the compiler if a conditional statement is likely to be true or not.

Figure 5.3 gives a simple example of how such macros could be used in practice. This particular example code is derived from our scheduler code. It checks if the task at the top of the run stack has a deadline timer set, and if so, if it has exceeded its deadline. As every task except the Linux task will have a deadline timer set, the first condition will likely be true. The compiler is informed of this fact using the `likely` macro. For the second condition that checks for a deadline overrun, the compiler will be informed that this condition is `unlikely` to be true.

Our implementation does not use these compiler based optimizations at the moment, as they are still marked "experimental". Furthermore they do not attribute to code readability, which would be impractical when debugging our implementation. While outside the scope of this paper, experimenting with these new compiler based optimizations might prove interesting nonetheless.

```
struct rtl_task_struct
{
    hrtime_t period;                            /* the task period */
    hrtime_t deadline;                          /* the task deadline (relative) */
    hrtime_t load;                              /* the worse case task load */
    struct rtl_ncs_struct *ncs;                 /* list of NCSs belonging to this task */
    pthread_t pthread;                          /* the pthread assigned to this task when it is admitted */
    /* ... snip ... */
}

struct rtl_resource_struct
{
    char *name;                                 /* the resource name */
    hrtime_t read_floor;                        /* the resource read floor */
    hrtime_t write_floor;                       /* the resource write floor */
    int ref_count;                              /* the number of NCSs referencing this resource */
    /* ... snip ... */
}

struct rtl_ncs_struct
{
    struct rtl_ncs_resource_struct *resources;  /* the list of resources this NCS uses */
    hrtime_t length;                            /* the lenght of the NCS */
    hrtime_t inherited_deadline;                /* the inherited deadline of the NCS*/
    struct rtl_ncs_struct *child;               /* a pointer to a possible embedded NCS */
    struct rtl_ncs_struct *next;                /* a pointer to a successor of this NCS */
    /* ... snip ... */
}

struct rtl_ncs_resource_struct
{
    struct rtl_resource_struct *resource;       /* the resource that is acquired */
    char *flags;                                /* resource usage flags (eg. if a resource is read or written) */
    struct rtl_ncs_resource_struct *next;       /* the next resource in the list */
}

struct rtl_thread_struct
{
    hrtime_t resume_time;                       /* the time of the last period start */
    hrtime_t inherited_deadline;                /* the current inherited deadline level */
    struct rtl_task_struct* task;               /* the task belonging to this thread */
    struct rtl_ncs_struct* ncs_cur;             /* the currently entered ncs */
    struct rtl_ncs_struct* ncs_enter;           /* the ncs to enter on the next pthread_ncs_enter() call */
    struct rtl_thread_struct next;              /* a pointer to a possible next thread */
    struct rtl_thread_struct prev;              /* a pointer to a possible previous thread */
    /* ... snip ... */
}
#define pthread_t rtl_thread_struct*            /* a pthread is really just a rtl_thread_struct */

struct rtl_sched_cpu_struct                     /* the scheduler state belonging to a specific CPU */
{
    struct rtl_thread_struct *rtl_current;      /* the currently running task */
    struct rtl_thread_struct rtl_linux_task;    /* the linux task */
    struct rtl_thread_struct *run_stack;        /* task list containing: [linux task][run stack][ready queue] */
    struct rtl_thread_struct *run_stack_top;    /* the top of the preemption stack */
    struct rtl_thread_struct *waiting_queue;    /* the waiting queue :-) */
    struct rtl_resource_struct *resources;      /* the list of used resources */
    /* ... snip ... */
}
```

Figure 5.1: Data structure overview

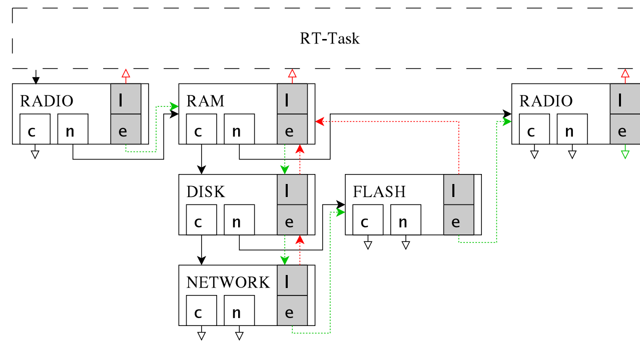

Figure 5.2: Performance optimized NCS structure

```
if (likely(test_bit(RTL_THREAD_DEADLINETIMERARMED, &sched->run_stack_top->threadflags)))
{
    if (unlikely(now >= sched->run_stack_top->resume_time + sched->run_stack_top->deadline))
    {
        /* handle deadline overrun */
    }
}
```

Figure 5.3: Compiler based optimizations

Chapter 6

Scheduler analysis

The scheduler analysis process consists of several areas. First there is the validation process, which is discussed in section 6.1. This involves verifying that the scheduler actually makes the correct scheduling decisions.
Then there is the process of determining the scheduler performance, which in itself consists of several important aspects. Determining which parts of the code actually contribute to the scheduler overhead is discussed in paragraph 6.2.1. A way to perform the various performance measurements efficiently is discussed in paragraph 6.2.2. Important is that the measurements themselves should not influence the results in an unacceptable manner. The system configuration is described in paragraph 6.2.3, as the performance of the scheduler depends on the system that is executing the tests.
Section 6.3 will finally present the results of measurements that have been conducted. Questions such as which test sets were used to measure the various scheduler characteristics[1], how were the tests executed, what were the expected outcomes and did the actual results conform to the expectations are discussed in this section.

6.1 Validation

To debug, verify and visualize the functioning of the scheduler, a tool called *Scheduler Insight* has been developed. This tool is capable of retrieving scheduling data from a previously stored scheduling logfile or acquiring it directly from a running real-time scheduler using a FIFO[2]. The scheduling information can be saved to a file and/or displayed graphically.

To validate the proper functioning of the scheduler a number of task sets were constructed. For each task set the resource floor calculations, inherited deadline calculations and feasibility analysis were performed manually and by our implementation. The results were then compared to verify their correctness. Online operational correctness was verified by executing several task sets and then examining the generated scheduler decisions using Scheduler Insight[3]. While these tests can never prove that our implementation is fully correct, we believe that, given the extensive testing that was performed, it gives a good indication about the implementation's overall quality.

An example that was used to verify the correctness of the scheduler is shown in figure 6.1. This particular example shows an executed DMI schedule for the task set shown in table 6.1. The tasks are sorted by their

[1] While we could solely construct test sets that measure a specific scheduler characteristic, we should note that it is important that the test sets are not too far away from "real world" use-cases. Measuring the overhead generated by thousands of tasks consisting of millions of nested critical sections might be interesting scientifically speaking, but the results would only be marginally useful when trying to improve the scheduler's design and/or implementation.

[2] A FIFO is a shared memory buffer using a *first in, first out* policy that can be accessed as a character device from kernelspace as well as userspace. This mechanism is provided by RT-Linux.

[3] The set of test applications that was created to test the behavior of the schedulers under various conditions is shorty discussed in paragraph B.3. This test suite can be used to test admission control, (dynamic) task admission and deletion, deadline overrun and NCS usage (including NPNCS and MUR support) behavior, amongst other things.

Γ	D	T	C	**Resources**
τ_1	400	500	100	90{ radio FLASHROM }
τ_2	500	800	100	80{ radio 20{ FLASHROM 10{ NETWORK }}}
τ_3	600	900	200	20{ flashrom } 170{ NETWORK 130{ flashrom }}

Table 6.1: Scheduler Insight example task set (values in milliseconds)

priority. Job execution is represented by blue horizontal bars. The figure also shows the execution of a forth task, shown in dark red, which is the Linux kernel that is executed in idle time only. The large green arrows pointing up denote the firing of task release timers, while the large red down-pointing arrows denote absolute task deadlines.

A colored line is drawn below the execution diagram for each task. This line represents the inherited deadline level of that task. When the line is black and flat, no deadline level is inherited. Whenever a task enters a nested critical section, the inherited deadline level might drop. This is represented by a dropping inherited deadline level line. When a nested critical section is left, the inherited deadline level might increase again. This is denoted by the line moving up. The various nested critical sections of a task are differentiated using different colors. The size of the change is visualized by the amount the line moves up or down, while the actual value of the inherited deadline is represented by a small arrow in the task execution bar using a color matching the NCS it belongs to.

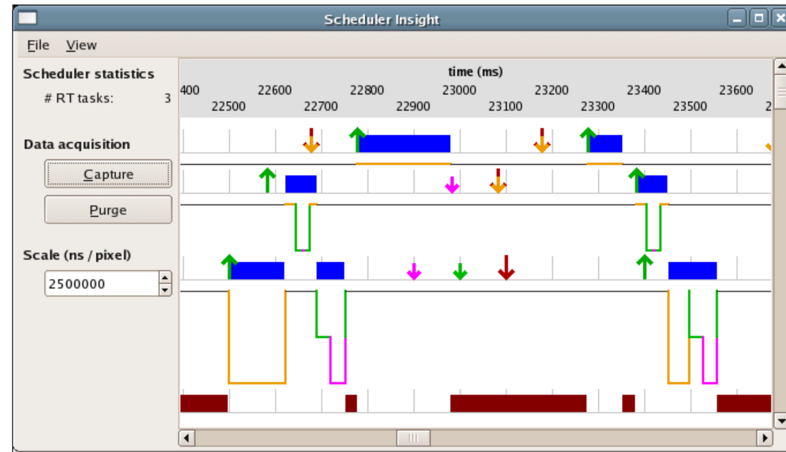

Figure 6.1: Scheduler Insight diagnostic tool

Take for example task τ_1 from table 6.1, which uses one NCS. We can see in figure 6.1 around $t = 22780ms$ that τ_1 enters the NCS immediately after its execution starts. The NCS is left again just before the execution finishes[4]. The time period during which the NCS was acquired is represented by the flat orange line. The flatness shows us that acquiring the NCS did not have any influence on its inherited deadline level.
A more interesting example is given by task τ_3. It acquires three nested critical sections, where the third NCS is nested in the second. At $t = 22500ms$ a new job for τ_3 starts after which it enters its first NCS immediately. We can see that its relative inherited deadline is set to $400ms$, inherited from τ_1. The small pink arrow at $t = 22900ms$ denotes the absolute inherited deadline level for both the first and third NCS, as they share the same inherited deadline level. Around $t = 22580ms$ the release timer for task τ_2 fires. While the priority of τ_2 is higher than the priority of τ_3, it is not allowed to start: its execution is blocked over the "flashrom" resource currently held by τ_3. When τ_3 releases this resource the job execution for τ_2 is allowed to start, preempting τ_3. Around $t = 23400ms$ the release timer for τ_2 fires just before that of τ_3. This time τ_2 does not have to wait.

[4] Observe that the job of task τ_1 that starts around $t = 22780ms$ uses more computation time than specified in table 6.1. The cause of this problem is discussed in paragraph 6.2.2.

6.2 Test setup

6.2.1 Overhead breakdown

To get accurate scheduler overhead measurements we first need to determine where the overhead can actually occur. Figure 6.2 shows a schematic overview of the job execution of the real-time task code shown in 6.3. The overview is marked with various points of interest.

The job's life cycle begins when it starts its period, denoted by the left-most CTX (context switch). After the context switch itself, the task will perform various administrative tasks such as signal handling. When the job execution reaches point S (start), the job will starts its normal execution. The execution will continue until point F (finish) is reached. The interval starting at point F to the point where the job execution ends is also used for various administrative tasks, such as moving the task from the run stack to the waiting queue.

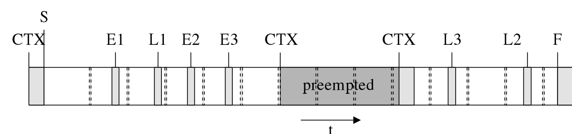

Figure 6.2: Job execution showing scheduler overhead

```
static void *pthread_routine(void *arg)
{
    ... task initialization ...
    while (1)
    {
        pthread_wait_np ();
        // Start
        /* ... do something ... */
        pthread_ncs_enter();
        /* ... do something ... */
        pthread_ncs_leave();
        /* ... do something ... */
        pthread_ncs_enter();
        /* ... do something ... */
        pthread_ncs_enter();
        /* ... do something ... */
        pthread_ncs_leave();
        /* ... do something ... */
        pthread_ncs_leave();
        /* ... do something ... */
        // Finish
    }
}
```

Figure 6.3: Example pthread routine

Every moment of the job's life-cyle that is not spent on performing its actual task is considered to be scheduler overhead. These are the areas shown light gray in figure 6.2. Overhead includes for example the interval between the initial CTX and point S, or the interval between point F and the point where the job ends. The dark gray area depicts an interval during which the job is preempted. The time during which the job is preempted can not be considered overhead for the currently running job. The preemption however does impose some overhead on the running job, such as the need for an additional context switch for example.

Observe that the task shown in figure 6.3 has three nested critical sections. The gray areas marked E1, E2, and E3 in figure 6.2 depict the overhead caused by entering the first, second and third nested critical section. Similarly the areas L1, L2 and L3 depict the overhead caused by leaving those nested critical sections again. Finally there are the dotted, periodic appearing areas. These areas depict the timed, periodic execution of the

scheduler. In our implementation the timer was programmed to generate interrupts at 2KHz, equal to the default RT-Linux setting.

We will use $O_{i,j}$ to denote the scheduling overhead imposed on job j of task i.

6.2.2 Data acquisition

After running several experiments using the Scheduler Insight tool to measure scheduler overhead, we found that the measurements themselves caused too great an impact on the result. This was caused by the overhead that the FIFO used by Scheduler Insight imposed on the kernel. During his research F. T. Y. Hanssen observed the same problem while performing measurements on scheduling performance in RT-Linux[6]. The impact of this problem is illustrated by the job execution of τ_1 shown in figure 6.1. Its two jobs depicted in the figure were executing exactly the same code. Observe that while the first job completes its execution in roughly 200ms, the second job is already finished within 100ms; a discrepancy of more than 100ms on a total execution time of around 100ms.

To get more accurate results, we had to slightly modify the real-time tasks and the scheduler itself. The modifications included time-stamping at specific moments in time (a relatively fast operation), and storing the results in fixed storage spaces. The timestamps were retrieved using the POSIX `gethrtime()` function, which returns the number of nanoseconds since system boot in a data type called `hrtime_t` (basically a C `long long`). Several `hrtime_t` variables were added to every task to store the important timing information. The overhead these additions imposed turned out to be negligible[5]. When a task was deleted from the system the timing information was printed to the standard UNIX syslog[6].

6.2.3 System configuration

All tests were carried out on a system containing an Intel Pentium IV processor running at 2.6GHz and holding 512Kb of cache memory. The system had 512 MB of RAM installed.

The base operating system was Fedora Core 3, which was modified to use the stock Linux 2.4.27 kernel. The RT-Linux kernel that was used was version 3.2-rc1, from which we modified the scheduler to implement DMI and EDFI. The compiler used to compile the Linux and RT-Linux kernels was GCC version 3.2.3. No changes were made to the default compiler flags.

Several system and scheduler aspects which influence the overhead of the scheduler could be configured. The configuration used in our tests was:

- Debugging support: disabled
- FPU support: disabled
- POSIX signals: disabled
- POSIX timers: disabled
- Tracer support: disabled (ability to log several scheduler aspects)
- All experimental optimizations: disabled

No system services, except networking, were enabled in our Fedora installation.

[5] Creating a time-stamp took around 30ns on our test system described in paragraph 6.2.3. This turned out to be negligible compared to the scheduler overhead results described in paragraph 6.3.

[6] We retrieved the data from the syslog for further processing using a simple “awk” script.

6.3 Results

This section will present the results of several tests that have been conducted. It must be noted that for some tests the absolute values presented in their results are not that important. If for example test 6-1 was executed on a system with a slower Pentium IV processor, then the measured absolute overhead values would turn out to be higher. For tests such as 6-3 and 6-4 the *relationship* between the various measurements is important. In test such as 6-1 and 6-2 the overall behavior of the schedulers is the important factor.

6.3.1 Test 6-1: DMI scheduler overhead

Test 6-1 was constructed to present an indication of the general behavior of the scheduler overhead. The used configuration used was:

- Scheduling policy: DMI
- Task set: $\Gamma_{6-1}(n)$, as shown in table 6.2
- Test duration: 300 seconds
- Number of tests: 51 tests were executed, with $n = 1, 10, 20, 30, .., 500$
- Measurement: the average scheduler overhead per job execution

$\Gamma_{6-1}(n)$	D	T	C
τ_1	1000	1000	1
τ_2	1000	1000	1
...	1000	1000	1
τ_n	1000	1000	1

Table 6.2: Specification for $\Gamma_{6-1}(n)$, values in milliseconds

Given this configuration, test 6-1 will take $51 * 300$ seconds to execute. For every task 300 jobs are executed given the period of 1 second. Every task measures the scheduler overhead (as specified in paragraph 6.2.1) its jobs experience. At the end of every test execution the amount of overhead that the tasks encountered is summed, and divided by the total number of jobs. This gives the average scheduler overhead per job execution (ASO):

$$ASO = \frac{\sum_{\substack{i=1..n \\ j=1..300}} O_{i,j}}{n * 300} \qquad (6.1)$$

Observe that no preemption will take place using task set Γ_{6-1}. Given this fact and the (hopefully) linear performance characteristics of the implementation as described in paragraph 4.3.3, we expected the ASO graph to be rising linearly with the number of tasks in the system.

Looking a the results shown in figure 6.4 we find that the actual results and our expected results are a close match. The average overhead per job execution is clearly linearly related to the number of tasks in the system. The results show an almost straight line, which seems to be primarily caused by the fact that tasks moving from the waiting queue to the ready queue can always be entered in the first position: all tasks have the same relative deadline, meaning equal priorities under DMI.
The jump of roughly $2\mu s$ that occurs when approximately 470 and 490 tasks are in the system is hard to explain. We suspect processor cache misses are occurring, resulting in a short delay where the processor fetches one or more cache-lines from the main memory. However, given the complexity of the Pentium IV caching mechanism we were unable to proof the correctness of this assumption.
To see if this assumption was plausible, we executed the same test on a different test system. This system contained an Intel Pentium M processor at 1.5GHz and holding 1MB of cache memory. Similar to our primary test system it had 512MB of RAM installed and it used the exact same operating system configuration. Given larger cache size of the processor, we expected the jump shown in figure 6.4 to disappear if cache misses were its cause. The results of this test are shown in figure 6.5. We find a flat line now, suggesting that the jump is indeed caused by processor cache misses.

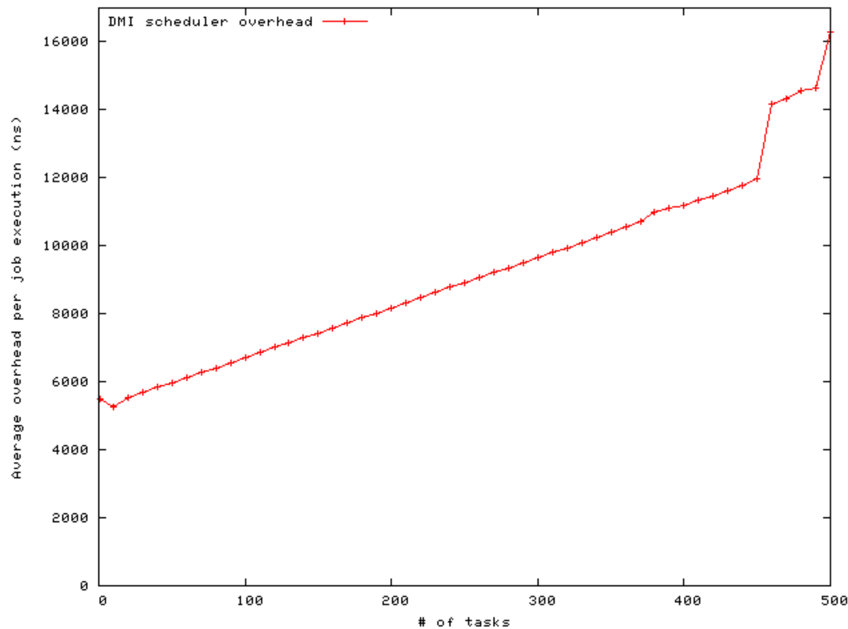

Figure 6.4: DMI scheduler overhead for Γ_{6-1}

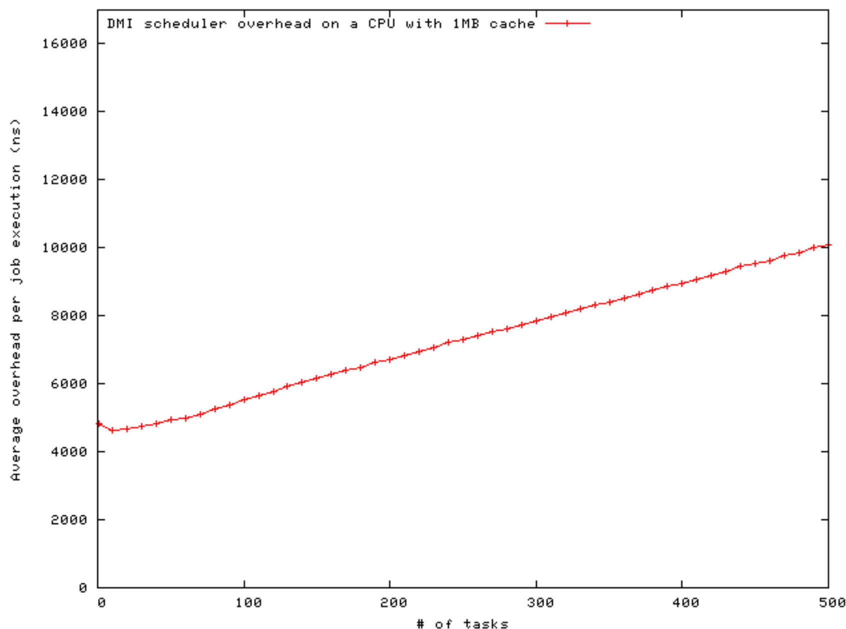

Figure 6.5: DMI scheduler overhead for Γ_{6-1} on a CPU with 1MB cache

6.3.2 Test 6-2: EDFI scheduler overhead

The configuration of test 6-2 is similar to that of test 6-1, except that the scheduling policy used is EDFI instead of DMI. The results of the test are shown in figure 6.6.

Again, no preemption will take place as all tasks share the same relative deadline. We expected the EDFI results to be roughly the same as the DMI result from test 6-1, but slightly more irregular as the difference in the ordering of the ready queue will start to be noticeable (see paragraph 4.3.3 for details on the queue insertion process). Especially when more tasks are admitted into the system at the same time.

The test results depicted in figure 6.6 show that the EDFI overhead behavior is indeed similar, but more erratic than that of DMI. Strangely enough we see that initially the average overhead drops (albeit only by roughly 1μs), only to start increasing when more than 30 tasks are in the system. While these results were easily reproduced, we could not find a proper explanation for this behavior[7].

[7] During the first presentation of this report a member of the audience suggested that, similarly to test 6-1 where cache misses were thought to be occurring, the initial drop might be caused by the processor cache “filling up”. This might prove to be a explanation for the observed behavior. However, if this is indeed the cause of the drop then we would have expected it to show up roughly equally strong in test 6-1 as well, but there the initial drop is a lot smaller.

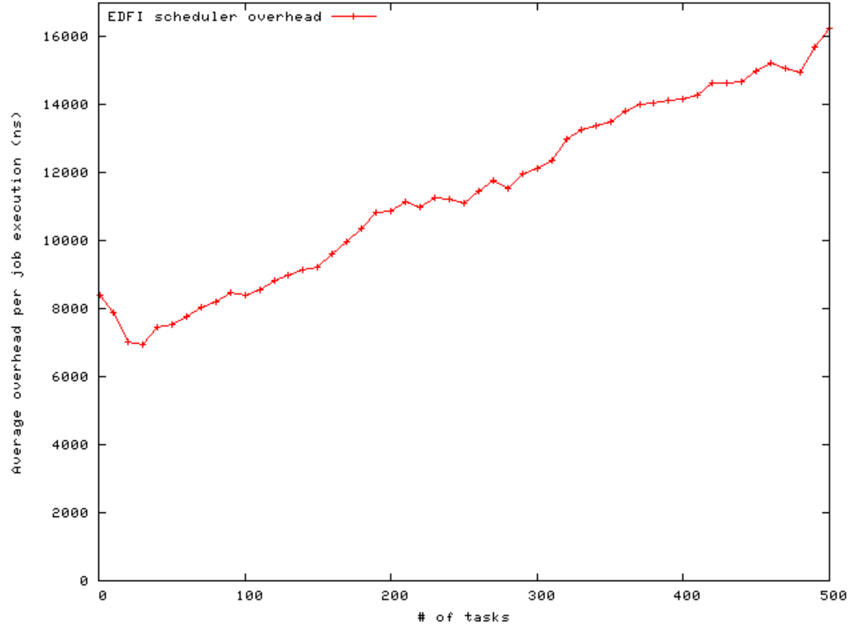

Figure 6.6: EDFI scheduler overhead for Γ_{6-1}

6.3.3 Test 6-3: Periodic scheduler overhead

Test 6-3 was designed to measure the average impact of the periodic scheduling interrupt generated at 2KHz. The used configuration used was:

- Scheduling policy: DMI and EDFI
- Task set: $\Gamma_{6-3}(c)$, as shown in table 6.3
- Test duration: 600 seconds
- Number of tests: 11 tests were executed for both DMI and EDFI, with $c = 1, 10, 20, ..., 100ms$
- Measurement: the average scheduler overhead per job execution

$\Gamma_{6-3}(c)$	D	T	C
τ_1	2000	2000	c
τ_2	2000	2000	c
...	2000	2000	c
τ_{10}	2000	2000	c

Table 6.3: Specification for $\Gamma_{6-3}(c)$, values in milliseconds

Observe that the periodic scheduler timer will interrupt a job with a load of 1ms twice, while a job with a load of 100ms will be interrupted 200 times. Therefore we expected that the scheduler overhead per job execution would slowly increase with an increasing job length.

The test results shown in figure 6.7 match our expected outcome. The results show a significant increase in scheduler overhead for DMI as well as EDFI when the job load increases. We find for example that when the

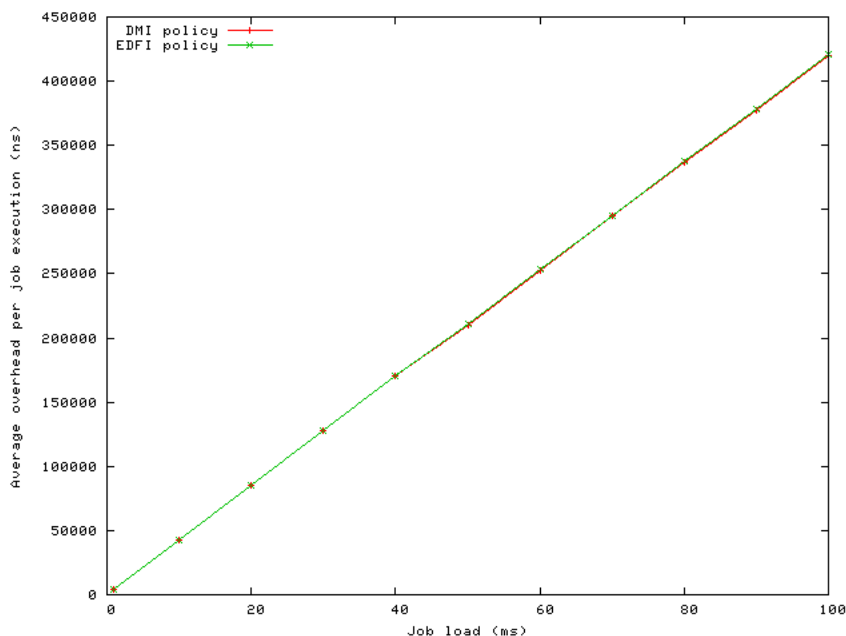

Figure 6.7: Scheduler overhead for Γ_{6-3}

load of a task becomes 100 times longer (from 1ms to 100ms), the scheduler overhead imposed on the task increases slightly less than 100 times (from 4639ns to 419715ns for DMI and from 4642 to 421054 for EDFI). The less than 100 times increase in scheduler overhead can easily be explained. Observe that when the load of a task increases the same schedule is generated, only stretched in time. Also observe that the total scheduler overhead measured during this test consists of two parts. The first part is the scheduler overhead that is needed to execute the schedule; this will obviously be exactly the same for every test run. The second part is the scheduler overhead that is caused by the periodic timer interrupts; when the load of a task doubles, the number of periodic timer interrupts imposed on the task doubles too. The second part is responsible for the linear correlation between the load of a task and the imposed scheduler overhead, while the first part is the cause of the smaller than 1 ratio between the increase in load of a task and the increase in imposed scheduler overhead.

6.3.4 Test 6-4: Job execution jitter

Test 6-4 was designed to inspect the job execution jitter, eg. the variations in job execution duration. The used configuration used was:

- Scheduling policy: DMI and EDFI
- Task set: Γ_{6-4}, as shown in table 6.4
- Test duration: 600 seconds
- Number of tests: 1 test was executed for DMI and 1 for EDFI
- Measurement: the job execution time

C^*: the workload for every job was to calculate the 9999999th Fibonacci number.

Γ_{6-4}	D	T	C^*
τ_1	2000	2000	-
τ_2	1950	1950	-
...	...	...	-
τ_{10}	1550	1550	-

Table 6.4: Specification for Γ_{6-4}, values in milliseconds

The periods and deadlines of each task were made different, so preemption could occur. Paragraph 6.2.1 describes what we consider to be the job execution length. Note that time interval during which a job is preempted will be subtracted from the measured job execution length.

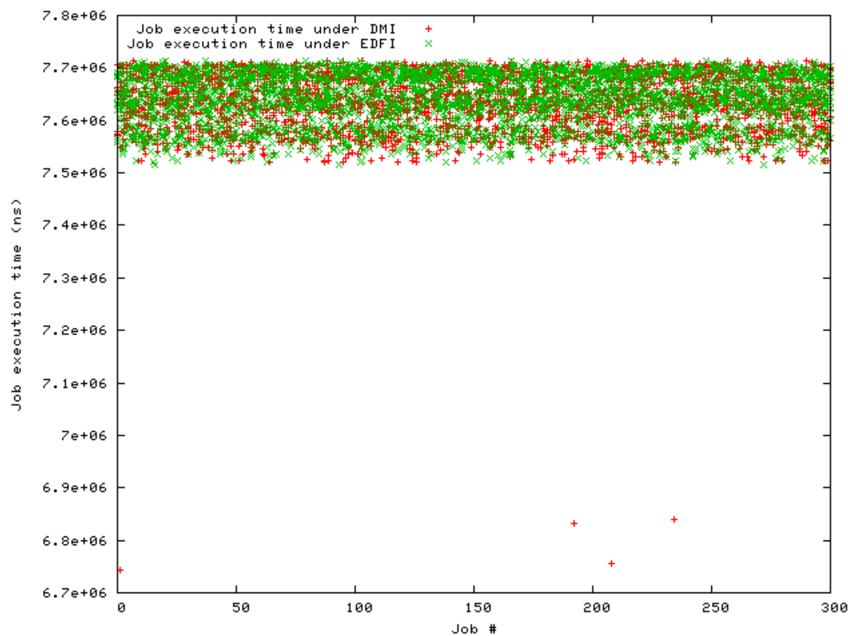

Figure 6.8: Job execution jitter

Variations in job execution length will come from factors that the scheduler can control, such as overhead differences (eg. differences in ready queue length or a job that is preempted or not) and from external factors, such as generated system interrupts. As all tasks have the same workload, we expected the job executions of every task to be roughly equal.

The test results shown in figure 6.8 overall match our expected outcome. Tasks scheduled under EDFI or DMI show the same job execution time. The job execution time for both DMI and EDFI varies slightly, roughly within a range of 250μs. Some DMI scheduled jobs however have a peculiar short execution length, roughly 900μs shorter than the rest. While we currently we can't explain this difference, we do not expect this result to be caused by the scheduler itself. The difference seems to be too high for that to be the case.

6.3.5 Test 6-5: Dynamic task admission and removal

Test 6-5 was designed to test the impact of task insertion and removal on an already running task set. The used test configuration was:

- Scheduling policy: DMI
- Task set: Γ_{6-5a} and Γ_{6-5b}, as shown in table 6.5 and 6.6
- Test duration: 180 seconds
- Number of tests: 5 tests were executed
- Measurement: the overhead experienced by a job at a particular moment in time

Γ_{6-5a}	D	T	C
τ_1	500	600	100
τ_2	500	600	100

Table 6.5: Specification for Γ_{6-5a}, values in milliseconds

Γ_{6-5b}	D	T	C
τ_3	700	800	200
τ_4	700	800	200

Table 6.6: Specification for Γ_{6-5b}, values in milliseconds

The test was configured to insert task set Γ_{6-5a} into the system, were it was allowed to execute for 180 seconds. After 70 seconds of execution task set Γ_{6-5b}, consisting of lower priority tasks, was admitted into the system. The tasks from Γ_{6-5b} were allowing to run for 40 seconds before being removed again. This process was repeated for five times.
We expected the scheduler overhead experienced by the jobs from Γ_{6-5a} to change by a very small amount when additional tasks were entered into the system. Of particular interest however is the scheduler overhead imposed on jobs from Γ_{6-5a} immediately after the admission of Γ_{6-5b}. A brief increase in overhead was expected.

The results for this test are shown in figure 6.9, which look even better than expected. The admission of new tasks did not interfere with the already running tasks at all. The scheduler overhead imposed on the jobs from Γ_{6-5a} does not change when Γ_{6-5b} enters or leaves the system. Interestingly the amount of scheduler overhead imposed on the tasks from Γ_{6-5b} differs every time it is inserted into the system. This might be caused by the different moments in time at which Γ_{6-5b} becomes part of the running schedule. Different starting times for example might cause less jobs from Γ_{6-5b} to be preempted, resulting in less overhead being imposed on them.

6.3.6 Test 6-6: NCS usage

Test 6-6 was designed to test the impact of entering an leaving nested critical sections. The used configuration used was:

- Scheduling policy: EDFI
- Task set: Γ_{6-6}, as shown in table 6.7
- Test duration: 450 seconds
- Number of tests: 1 test was executing with MUR enabled, and 1 without
- Measurement: the NCS overhead per job execution

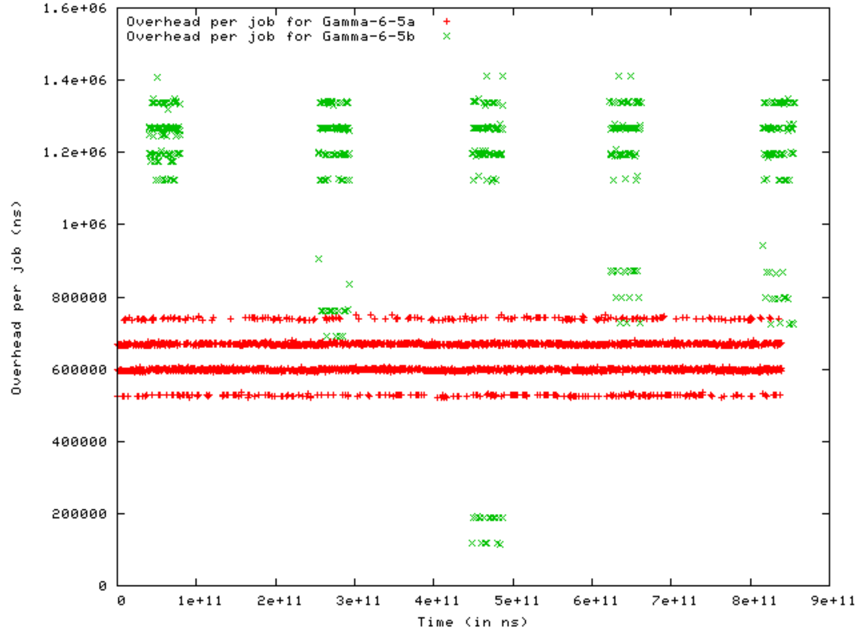

Figure 6.9: Dynamic task admission and removal

Γ_{6-6}	D	T	C	R^*
τ_1	1500	1500	300	-
τ_2	1500	1500	300	-
τ_3	1500	1500	300	-
τ_4	1500	1500	300	-

Table 6.7: Specification for Γ_{6-6}, values in milliseconds

Note that there is no need to perform the same test under DMI as well. The NCS implementation for both algorithms is exactly the same.

R^*: as the overhead of entering and leaving a NCS will be rather low (see paragraph 5.6.2), a task must have quite few NCSs to get some measurable results. Therefore each task will be equipped with a rather insane NCS:

```
300 {   r
        100{a 10{B} 10{C} 10{b} 10{c} 10{B c} 10{b C} 10{b c} 10{B C}}
        100{b 10{A} 10{C} 10{a} 10{c} 10{A c} 10{a C} 10{a c} 10{A C}}
        100{c 10{A} 10{B} 10{a} 10{b} 10{A b} 10{a B} 10{a b} 10{A B}}
}
```

It must be noted that measuring the NCS overhead itself generates overhead as well. A time-stamp is made on entering and leaving the `pthread_ncs_enter()` function. The same is done when entering and leaving the `pthread_ncs_leave()` function. Given the NCS used in this test, 28 enter and leave calls are made by each job. Making a time-stamp takes roughly 30ns (see paragraph 6.2.2), so the total overhead from the measurements alone will be around $28 * 2 * 2 * 30 = 3360$ns.

We expected the results in the non-MUR enabled test to be not much higher than the measuring overhead itself (see paragraph 5.6.2). For the MUR enabled test we expected the results to be slightly worse, as the inherited deadline calculation is performed online (see paragraph 3.7.2).

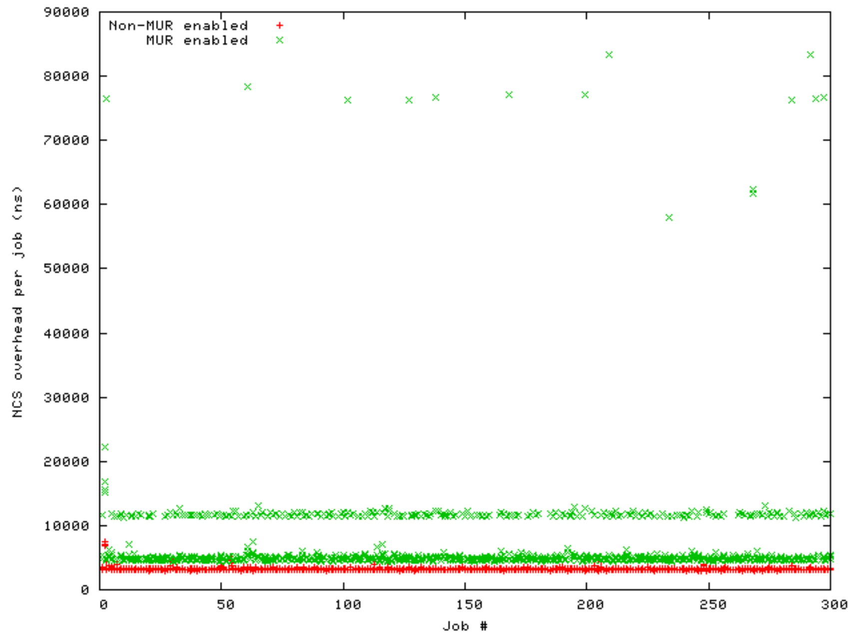

Figure 6.10: NCS overhead

The results depicted in figure 6.10 show that in the non-MUR enabled test the NCS overhead behaves as expected. All jobs experienced a NCS overhead of around 3600ns, which is just marginally higher than the overhead caused by the measurements itself. This means that the traversing a NCS structure is indeed as fast as we expected it to be. Notice that the first few data points (for both the non-MUR as the MUR enabled test) are about twice as high as the other data points. This result is expected, as the inherited deadline values calculated during the feasibility analysis are dynamically adopted for each NCS (see paragraph 5.6.3).
Looking at the MUR enabled test results we find that the overhead that is generated is substantially higher than in the non-MUR enabled case. This result was predicted in paragraph 3.7.5.

The results also show that compared to the rest of the jobs, a dramatically higher overhead is imposed on a few jobs; roughly 60μs higher. The cause of these spikes might lie in the way the measurements are performed: first a time-stamp is taken when `pthread_ncs_enter` is called (see section 5.5). Next, interrupts are disabled, followed by the process of performing the bookkeeping needed to enter a NCS and adapt the inherited deadline level. Finally interrupts are enabled again and a final time-stamp is taken. The interval between the two time-stamps is considered to be NCS overhead. A similar process is performed when `pthread_ncs_leave` is called.
Now imagine that a job is interrupted exactly in between the moment of taking a time-stamp and enabling or disabling the interrupts. This will cause the duration of the interrupt handling routine to be considered NCS overhead, which will show up as a spike in the test results.
The fact that we only see these spikes in the MUR enabled test might be caused by the longer time needed to enter or leave a NCS. During this process interrupts might occur that can be handled immediately after enabling interrupts again (that is, just before the final time-stamp is taken). The chance of this happening is simply greater in the MUR enabled test than in the non-MUR enabled test. We thus expect similar spikes to occur in the non-MUR enabled test if it is executed often enough. Whether or not this is indeed the cause of the spikes can be tested by moving the time-stamping in between the interrupt disabling and enabling calls. Note however that the actual measured NCS overhead would then show a result that is slightly too low.

6.3.7 Test 6-7: Typical task set

The last test was designed to measure the scheduler performance when execution a task set that could resemble a typical “real-life” situation. The used configuration used was:

- Scheduling policy: EDFI
- Task set: Γ_{6-7}, as shown in table 6.8
- Test duration: 1000 seconds
- Number of tests: 1 test was executed
- Measurement: the scheduler overhead per job execution

As an exercise to the reader, observe that this task set is infeasible under DMI.

Γ_{6-7}	D	T	C	$Resources$
τ_1	4	5	1	0.9 { a B }
τ_2	5	8	1	0.8 { a 0.2 { B 0.1 { C }}}
τ_3	6	10	2	0.2 { b } 1.7 { c 1.3 { b }}
τ_4	9	9	3	1.8 { a b }

Table 6.8: Specification for Γ_{6-7}, values in seconds

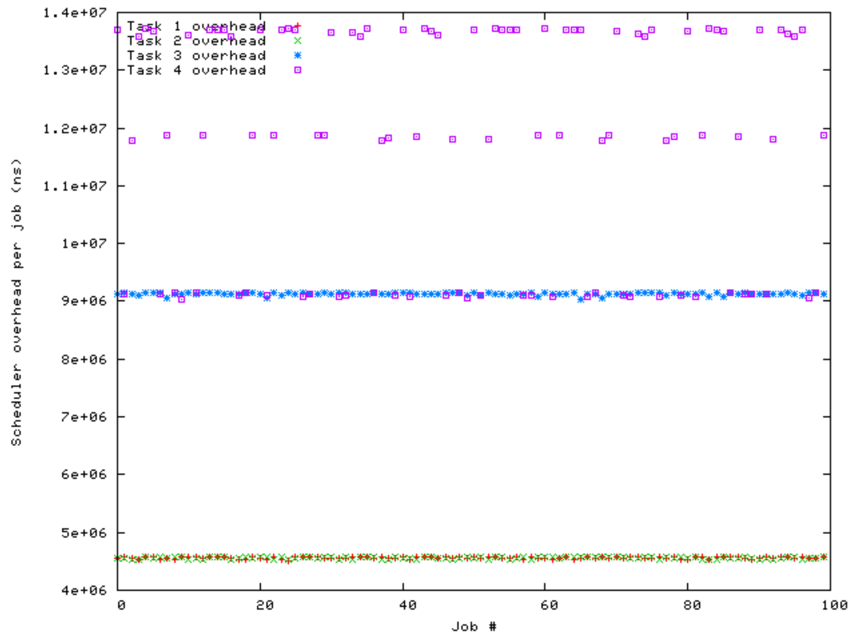

Figure 6.11: Typical task set scheduler overhead

The results depicted in figure 6.11 show that τ_3 and τ_4 experienced the most overhead. Looking at the generated schedule we found that τ_4 was frequently preempted because of its long relative deadline and short NCS length (observe that within its NCS, τ_4 will inherit the deadline from τ_1 over resource b, making it non-preemptable). τ_1 on the other hand was never preempted given its shortest relative deadline. τ_2 didn’t happen to be preempted

either because of its short relative deadline and the fact that it inherited for part of its execution the deadline from τ_1 over resource B. While τ_3 wasn't preempted due to inheriting the deadline of τ_1 and τ_2 for most of its execution time, its longer execution length made it experience an additional 4.5ms overhead compared to τ_1 and τ_2.

Given the hyperperiod of 360 seconds of the task set, we summed the overhead imposed on all four tasks during that time period. The total amount of scheduler overhead generated was 1.328 seconds, resulting in an overhead percentage of just $\frac{1.328}{360} * 100\% = 0.369\%$.

6.3.8 Discussion

The test results regarding the behavior of our schedulers look very promising, as most results were in line with our expectations. Test 6-1 measured the correlation between the number of tasks in the system and the average scheduler overhead imposed on a job. The data is encouraging, showing only a doubling of the scheduler overhead imposed on a job when the number of tasks increased four-hundred-fold. The results also showed an unexpected jump in the average scheduler overhead graph when 470 tasks were admitted into the system. With reasonable certainty however we can attribute these results to processor cache misses, as the jump did not show up when testing with a processor having twice as much cache memory as our target processor. This also shows that, hardly surprising, the behavior of the scheduler depends on the hardware it is executing on.
Test 6-2 also performed in line with our expectations, with only a small initial drop in the average scheduler overhead graph. To a lesser extend this showed up in the results of test 6-1 as well. Further research needs to be done to find the cause of this result. Investigating the possibility of the processor cache filling up might prove to be a good starting point.
The influence of the periodic scheduler overhead was measured by test 6-3. The results of this test matched our expectations exactly; increasing the load of a task with a factor f also increases the amount of scheduler overhead imposed on the task, but with a factor slightly lower than f.
Job execution jitter was measured in test 6-4 by executing a number of jobs with an equal load. The results showed that the jitter was occurring in a narrow band of roughly 250μs. Strangely enough 3 out of the 3010 jobs executed completed about 900μs earlier than the rest. This could not have been caused for example by system interrupts, as that would increase the execution length of the jobs, not shorten it. While the shorter execution length obviously is not harmful to the feasibility of the system, it would be interesting to know what is causing these results.
The impact of task admission and removal on an already running task set was measured by test 6-5. Test results showed that neither task admission nor removal have a significant impact on the overhead that is imposed on the tasks that are already in the system. This is interesting, as it shows that the measures taken during the implementation to prevent overhead peaks from occurring work in practice (see paragraph 5.6.3). Scheduler overhead caused by entering and leaving a NCS was measured by test 6-6. As expected the overhead is almost to low to measure. When multi-use resources are used the overhead increases significantly, as predicted in paragraph 3.7.5. Even more, 15 out of the 1200 measurements taken showed a significantly higher overhead during the MUR enabled test. A possible cause of these unexpected results is discussed, which could use some further investigation.
Finally test 6-7 was created to measure the scheduler overhead that could resemble a typical real-life situation. The scheduler overhead made up 0.369% of the total execution time, which is a very respectable result.

Chapter 7

Future work

This chapter tries to identify several areas that could benefit from future work or research, some regarding the discussed theory and some regarding our implementation.

7.1 Theory

Several scheduling concepts presented in this paper deserve some more attention, especially the DTA condition and multi-use resource theory.

7.1.1 DTA condition analysis

The DTA condition we presented in section 3.5 followed naturally out of the existing DMI and EDFI theory. While the condition is better than several alternative solutions discussed in the same section, we can not determine how good or bad it actually is. Is it optimal, or not? And if not, could a better or even optimal solution be constructed that is also useful in practice? Answering these questions could make it possible to improve our implementation.

Observe that the DTA condition from section 3.5 can not be used together with support for multi-use resources. The DTA condition is based on the notion of static inherited deadlines, a concept that does not apply to MUR. The current implementation however does allow DTA together with MUR, but tasks are only allowed to enter the system when the run stack is empty[1].

7.1.2 Multi-use resources

At first glance the MUR theory has several commonalities with the SRP. It would be interesting to compare both theories in more detail. The optimal feasibility analysis for example is NP-complete for both MUR and the SRP. The MUR feasibility analysis however can be optimized considerable as whole groups of possible ncs-stacks can be determined to be infeasible at once. Even more, the models that MUR and the SRP deploy also look similar, but SRP allows simultaneous readers and writers. Determining the differences between both models might prove interesting.

As mentioned in paragraph 3.7.1, the current MUR model does not allow a task to acquire the same resource multiple times for reading or writing within a single NCS. To allow for this, several problems would need to be solved. For a start, the definition of a blocker would have to be redefined. Consider for example a task

[1]The run stack *will* be empty at some point in time, otherwise the system would not be feasible. This point will be reached within a time interval that is smaller than, or equal to the hyper-period of the task set.

at the front of the ready queue that acquires resource R twice for writing. Assume that R would also have a writecount of 2. Now if two tasks are on the run stack, each having acquired R for writing, then which task would be considered to be a blocker? The bottommost task on the run stack, the topmost task or perhaps both? The last option seems to be the most intuitive solution. In this case the feasibility analysis algorithms would have to be reconstructed as well, as neither the DMI nor the EDFI feasibility algorithm can deal with blocking coming from more than one task at the same time.
Until these issues are resolved (and probably more would come up in the process), the MUR model can not allow a resource to be acquired multiple times within a single NCS.

7.2 Implementation

Several implementation details could use some attention as well, mostly to improve the robustness of our implementation. We will briefly address the most important (in our opinion) outstanding issues in this section.

7.2.1 Resource acquisition

Every NCS specifies the maximum time interval that resources can be acquired. At run-time it is important that this interval is not exceeded, as that could endanger the feasibility of the whole system. Therefore the duration that resources are acquired should be monitored. While such additional checks will increase the scheduler overhead slightly, the safety of the system might be worth it.

Our current resource handling mechanism needs to be improved in the case that a task exceeds its deadline. As described in paragraph 4.3.4, the offending task will be moved to the waiting queue and will be flagged DEAD to prevent it from executing again. This approach works fine in the case no resources are acquired at the moment the deadline is exceeded. However, when resources *are* acquired, they will not be released again as the task is prevented from executing. To solve this problem, a signaling mechanism should be added to our implementation to force a task to release all its currently held resources when it exceeds its deadline.

7.2.2 Semantical correctness

Every task has a resource usage description, which is a character string that is specified by the application programmer. This string is automatically parsed into a structure that can be used by the admission control mechanism and the scheduler. The current implementation checks the resource usage description for syntactical correctness, but not for semantical correctness.
Additional checks could be added to verify semantical correctness of a resource usage description. For example a NCS embedded inside another NCS should not have a bigger load specified than the load of its parent. Additionally an embedded NCS should not re-acquire resources that one of its parents already acquired; the child NCS can safely access such resources without any additional specification. Finally, in the case of multi-use resources, the admission control mechanism could check if all tasks specify the read and writecount of all resources equally. If one task specifies a resource to be $R_{4,1}$ and another task specifies the same resource to be $R_{2,1}$, then clearly something is wrong. In such a case tasks should be prevented from entering the system.

7.2.3 Timer latencies

Currently the scheduler uses a periodic timer generating interrupts at 2Khz. While this will generally be "fast enough", it means that the release timer latency (the interval between the actual firing of the release timer and when it was supposed to fire) can be as long as 500μs. The same holds for the deadline timer latency.
Improvements to reduce the timer latencies can be made by not statically setting the next interrupt to be generated 500μs after the previous interrupt was generated, but by determining when the next timer event is actually

supposed to take place. The scheduler currently does have all the information needed to make such a decision, so it only needs some additional bookkeeping to keep track of the event sequence.

7.2.4 Backwards compatibility

While special care has been taken to allow existing (legacy) RT-Linux applications to run under the new schedulers, no guarantees for proper functioning can be given at the moment. As legacy applications do not have an explicit relative deadline (see paragraph 4.2.1), an artificial relative deadline is created automatically equal to the period of the task. Similarly legacy applications do not have a resource usage specification. In this case an empty specification is assumed.

7.2.5 Measurements

As described in section 6.3, most test results conformed to our expectations. However, some anomalies such as the initial drop in scheduler overhead in test 6-2, or the spikes in the MUR enabled test from test 6-6 are currently unexplained. These unexpected results should be looked into.

7.2.6 Testing

The interaction between several aspects of the theory and in some parts its complexity has lead to a relatively complex implementation. Fortunately the online operation of the scheduler has been kept relatively simple, with all computational and algorithmical complexity moved into the admission control mechanism. The DMI and EDFI feasibility analysis algorithms (including the use of the presented bounded feasibility theory), support for dynamic task admission, non-preemptable nested critical sections and support for multi-use resources (including both the sufficient and necessary feasibility analysis algorithms) have all been implemented and work in all possible combinations. An EDMI scheduler with support for MUR, NPNCSs and DTA can readily be used, for example.

Given the complexity and configurability of the implementation, and the huge variety of task sets that can be created using an equally wide range of NCS configurations, it is almost impossible to test every combination for functional correctness. While we worked hard to create a bug-free implementation, we can't guarantee its complete correctness, though no bugs are currently known.

Chapter 8

Conclusions

In this paper we have summarized and verified the existing DMI and EDFI theory. Imperfections in the DMI feasibility analysis, resource read and write floor definitions and NCS specification language have been corrected. Both the DMI and EDFI feasibility analysis have been modified to allow for nested critical sections. Finally we have presented a more precise feasibility bound for EDFI.

A number of additions to the exiting theory has been made. A dynamic task admission condition has been presented and proved for its correctness. A theory has been given to allow non-preemptable sections that can be easily implemented and used in practice. Finally a theory to allow the use of multi-use resources has been presented. This theory contains a sufficient as well as a necessary feasibility algorithm.

To implement the DMI and EDFI schedulers in RT-Linux, a design has been constructed that makes the RT-Linux scheduler framework cooperate nicely with the scheduler organisation required by DMI and EDFI. Our implementation supports all DMI and EDFI related theory discussed in this paper. Special care has been taken to guarantee that the worst case performance of the schedulers was as low as possible. The application programmers interface to control all scheduler features has been kept simple, abstracting all calculations away from the application designer.

A tool called Scheduler Insight has been created to verify the operational correctness of the schedulers. It can be used to present the decisions made by the scheduler in an orderly manner, even while the system is currently executing. Using this tool to visualize the behavior of a real-life DMI or EDFI scheduler might serve a didactic purpose as well, for example to familiarize students with the theory in an understandable way.

The overall behavior of the overhead generated by the schedulers matched our expectations. Some tests generated results that where not anticipated. Most of these unexpected results could be explained, while for some a possible cause was given that could be investigated further. The measurements show that the implementation behaves, within limits, in a predictable manner. This makes the implementation suitable for example for use in other research projects that require a resource scheduling real-time system.

Bibliography

[1] N. C. Audsley, “Resource Control for Hard Real-Time Systems: A Review”, Real-Time Systems Research Group, Department of Computer Science, University of York, York, UK, 5th August, 1991

[2] T. P. Baker, “A Stack-Based Resource Allocation Policy for Realtime Processes”, Proceedings 11th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, Dec. 1990

[3] G. C. Buttazzo, “Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications”, Scuola Superiore S. Anna, Pisa, Italy, Kluwer Academic Publishers, Fourth Printing, 2002

[4] M. I. Chen and K. J. Lin, “Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems”, Report No. UIUCDCS-R-89-1511, Department of Computer Science, University of Illinois at Urbana-Champaign, Apr. 1989

[5] R. Graham, “Bounds on the Performance of Scheduling Algorithms”, chapter in Computer and Job Shop Scheduling Theory, John Wiley and Sons, pp. 165-227, 1976

[6] F. T. Y. Hanssen, “Scheduling and resource allocation with Real-Time Linux”, Master’s thesis, University of Twente, The Netherlands, October 1998

[7] F. T. Y. Hanssen and P. G. Jansen and H. Scholten and S. J. Mullender, “RTnet, a distributed real-time protocol for broadcast-capable networks” , in Proceedings ICAS/ICNS 2005, 2005, accepted for publication

[8] P. G. Jansen, “A Generalized Scheduling Theory based on Real-Time Transactions”, University of Twente, The Netherlands, January 2003

[9] P. G. Jansen, S. J. Mullender, P. J. M. Havinga, H. Scholten, “Lightweight EDF Scheduling with Deadline Inheritance”, University of Twente, The Netherlands, May 9, 2003

[10] P. G. Jansen, “Deadline Monotonic with Inheritance”, Sheets for the course of Real-Time Systems, http,//wwwhome.cs.utwente.nl/ jansen/courses/rts1/inf/sheets/dmi.pdf, University of Twente, The Netherlands, Jan. 20, 2004

[11] C. L. Liu, J. W. Layland, “Scheduling algorithms for Multiprogramming in a Hard Real-Time Environment”, *Journal of the Association for Computing Machinery*, 20(1), 1973, pp 40-61

[12] E. L. Lawler, “Optimal sequencing of a Single Machine Subject to Precedence Constraints”, Management of Science, 19, 1973

[13] J. Leung and J. W. Whitehead, “On the complexity of fixed priority scheduling of periodic real-time tasks”, Performance Evaluation, 2(4), 1982.

[14] R. Rajkumar, L. Sha, J. P. Lehoczky and K. Ramamithram, “An Optimal Priority Inheritance Protocol for Real-Time Synchronisation”, COINS Technical Report 88-98, Department of Computer and Information Science, University of Massachusetts, Oct. 17, 1988

[15] R. Rajkumar, L. Sha and J. P. Lehoczky, “An Experimental Investigation of Synchronisation Protocols”, Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software, pp. 11-17, May 1989

[16] L. Sha, R. Rajkumar and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronisation”, CMU-CS-87-181, Computer Science Department, Carnegie-Mellon University, Dec. 1987

[17] L. Sha, R. Rajkumar and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronisation”, IEEE Transactions on Computers 39(9), pp. 1175-1185, Sept. 1990

[18] M. Spuri and G. C. Buttazzo, “Efficient aperiodic service under earliest deadline scheduling”, Proceedings of the IEEE Real-Time Systems Symposium, Dec. 1994

[19] M. Spuri and G. C. Buttazzo, “Scheduling aperiodic tasks in dynamic priority systems”, Journal of Real-Time Systems, 10(2), 1996

[20] V. Yodaiken, “The RTLinux Manifesto”, Department of Computer Science, New Mexico Institute of Technology, 1999

[21] V. Yodaiken, “Against Priority Inheritance”, FSMLabs Technical Report, Finite State Machine Labs, October 1, 2002

Appendix A

Real-time task example

```
/* RT-Linux DMI/EDFI TestSuite
 *
 * Copyright (C) 2004-2005 Marc Maurer <j.m.maurer@student.utwente.nl>
 * Released under the terms of the GNU General Public License Version 2
 */

#include <linux/slab.h>
#include <rtl.h>
#include <time.h>
#include <rtl_time.h>
#include <rtl_sched_admctrl.h>
#include <pthread.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Marc Maurer");
MODULE_DESCRIPTION("RTLinux DMI/EDFI test application");

static struct rtl_task_struct *tasks[2];

/*
 * The periodic task routine
 */
static void * start_routine(void *arg)
{
    struct rtl_task_struct *task = *((struct rtl_task_struct **)arg);
    if (!task)
    {
        rtl_printf("BUG: task is NULL in start_routine!\n");
        return 0;
    }

    pthread_init_finalize_np (pthread_self(), task); /* finish and signal the end of the initialization */
    while (1)
    {
        pthread_wait_np (); /* wait for the next release timer fires */

        /* TODO: implementation here */

        pthread_ncs_enter();
        /* the radio resource can now be acquired for reading,
           and the flashrom resource for writing */
        /* TODO: implementation here */
        pthread_ncs_leave();

        /* TODO: implementation here */

        pthread_ncs_enter();
        /* we are now in a non-preemtable NCS */
        /* TODO: implementation here */
        pthread_ncs_leave();

        /* TODO: implementation here */
    }
}
```

```
int init_module(void)
{
    /* Contruct an example task set (zero terminated array) with just
       one task. Feel free to add more. */
    rtl_task_create(&(tasks[0]), 500000000, 400000000, 100000000,
            "90000000{ radio FLASHROM } 50000000 { ! }",
            0, start_routine, (void*)(&tasks[0]));
    if (!tasks[0]) return 1;
    tasks[1] = 0;

    /* insert the new task set into the system, but only if the new tasks
       combined with the tasks that are already present in the system form
       a feasible task set */
    int feasible = rtl_admctrl_admit_if_feasible(tasks);

    rtl_printf("Task set is %s\n", (feasible ? "feasible" : "not feasible" ));

    return 0;
}

void cleanup_module(void)
{
    /* delete the tasks from the system */
    if (tasks[0])
    {
        if (tasks[0]->pthread)
            pthread_delete_np(tasks[0]->pthread);
        else
            rtl_admctrl_task_free(tasks[0]);
    }
}
```

Appendix B

Utilities

Several utilities have been created during the development of the DMI and EDFI schedulers and their accompanying admission control mechanisms[1]. This appendix will briefly discuss the purpose of each utility, so future developers could benefit from them as well.

B.1 rtlinux-si

This module contains the Scheduler Insight tool discussed in section 6.1. It can be used to debug, verify and visualize the functioning of the scheduler. It is capable of retrieving scheduling data from a previously stored scheduling log-file or acquiring it directly from a running real-time scheduler. The scheduling information can be saved to a file and/or displayed graphically.

B.2 rtlinux-measurements

This module contains the measurement programs that have been used to execute the tests discussed in section 6.3. These test programs can be used to duplicate the results we presented, or to produce results using a system with a different configuration from the one discussed in paragraph 6.2.3.

B.3 rtlinux-testsuite

The test suite contains a set of example programs that tests all aspects of the DMI and EDFI schedulers as well as the admission control mechanisms. We used this set of programs to verify the correctness of our implementation (see section 6.1).

B.4 rtlinux-admctrl

The `rtlinux-admctrl` program implements the same admission control mechanisms as those included in our kernel implementation. This allows application developers to test the feasibility of task sets in "userspace",

[1]The utilities discussed in this appendix, together with our modifications to the RT-Linux source tree, can be found in the CVS (Concurrent Versions System) repository located at `http://maas.cs.utwente.nl/cgi-bin/viewcvs.cgi/`.

and it allows easier debugging of the admission control implementation[2].

B.5 rtlinux-sync

A script called `rtlinux-sync` has been created to keep our local RT-Linux source tree synchronized with the official RT-Linux source tree[3]. All changes made to the official source tree since the last synchronization run can be merged into our local tree using this script. Changes to the scheduler code will be excluded from the synchronization process.

B.6 rtlinux-ksymoops

For debugging our scheduler and admission control implementations we initially used the `ksymoops`[4] utility. This program can produce call-traces by passing it a kernel crash report (the output that the kernel generates when it "panics"). Using these call-traces a programming error can be localized. Due to unknown reasons we experienced instabilities in `ksymoops` itself, making it segfault instead of producing a call-trace. This lead us to writing a similar tool ourselves, resulting in `rtlinux-ksymoops`. This tool is equally capable of generating call-traces by passing it a kernel crash report.

[2] Errors are typically harder to track down in kernelspace than in userspace. An error made in kernelspace can bring down the complete system, depriving the developer of the possibility to inspect the state of the system at the moment the error occurred.

[3] The official RT-Linux source tree is located at `http://rtlinux-gpl.org/`.

[4] The `ksymoops` utility is located at `ftp://ftp.kernel.org/pub/linux/utils/kernel/ksymoops/v2.4/`.