Extending RTAI/Linux with FPDS

Bergsma, M.

Award date:
2009

Extending RTAI/Linux with FPDS

Mark Bergsma

System Architecting & Networking
Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven

July 31, 2009

Supervisors

Mike Holenderski MSc.
dr. ir. Reinder J. Bril
prof. dr. ir. Johan J. Lukkien

Abstract

Fixed-priority scheduling with deferred preemption (FPDS) has been proposed in the literature as a viable alternative to fixed-priority preemptive scheduling (FPPS), that both reduces the cost of arbitrary preemptions and removes the need for non-trivial resource access protocols.

The goal of this Masters project is the extension of the real-time operating system RTAI/Linux with a stable, efficient, extendable, maintainable, compatible and freely available implementation of FPDS, as well as a discussion of design decisions and tradeoffs. The project is carried out in the context of the ITEA2/CANTATA project, at the System Architecting & Networking group of the University of Technology, Eindhoven.

We discuss some related work concerning earlier cooperative scheduling techniques and applications, and provide a summary of the theory and notations used in fixed-priority scheduling, and FPDS in particular. The architecture of the RTAI/Linux is described along with some implementation details relevant to our work. We lay out our design assumptions and list the key performance indicators that we use to check our results. In a discussion of possible alternatives to map an FPDS task set onto a real-time operating system, we choose to implement FPDS using non-preemptive tasks split up into subtasks using a kernel API primitive. In the following chapters, we follow an incremental approach in designing and implementing both FPNS and FPDS in RTAI. Tests indicate that the resulting implementation correctly implements FPDS scheduling, and through a series of measurements we verify that the overhead introduced into the system is low.

After some investigation we identify the use of system calls as the biggest source of overhead in our FPDS design, and follow up with a newer and more efficient FPDS implementation with optional preemption points that avoids this overhead. Comparative measurements of both FPDS implementations show a significant improvement, with the overhead of the new implementation being almost as low as the original FPPS scheduler in RTAI.

As part of future work, we investigate an extension of FPDS, utilising knowledge of the subtask structure of tasks: the monitoring and enforcement of FPDS jobs. We describe how FPDS tasks and the RTAI kernel can cooperate to detect and act against tasks that will overrun their stated computation times and cause interference with other tasks.

Finally, we conclude that our chosen approach and design resulted in an FPDS extension that meets the stated requirements, and that the use of FPDS in a real-world system is feasible with low overhead.

Part of this work resulted in a paper that was published in the proceedings of the OSPERT 2009 workshop held in conjunction with the Euromicro Conference on Real-Time Systems (ECRTS) conference in Dublin [3].

Contents

Chapter 1

Introduction

With software systems increasing in complexity seemingly without bounds, the amount of possible scenarios and states a system can reach is quickly outgrowing the capacity of their human developers to fully grasp. The number of bugs and occurrences of unexpected behaviour is therefore increasing as well, and the ongoing immersion of society in technology means that the consequences can become quite disastrous.

The field of Real-Time Systems attempts to improve on part of this problem, by focusing on the *predictability* of systems in the aspect of timeliness. In many real-world situations, the result of an action arriving *on time* is as important as its correctness.

While generally the solution direction appears to be increased (time) control and enforcement of behaviour of components from above, for instance in the area of scheduling with full priority based preemption and reservations of resources, recently there has been some renewed research interest in more cooperative schemes as well. It is this domain, cooperative scheduling, that provides the context for the work described in this thesis.

1.1 Problem statement and motivation

The goal of this Masters project is to extend RTAI/Linux [1] with support for Fixed-Priority Scheduling with Deferred Preemption (FPDS). The design and implementation should be stable, efficient, extendable, maintainable, compatible, and be freely available to everyone for future research experiments. Discussion of architectural tradeoffs and design decisions is an important element of the assignment.

FPDS [6, 8, 9, 11, 14] has been proposed in the literature as an alternative to Fixed-Priority Nonpreemptive Scheduling (FPNS) and Fixed-Priority Preemptive Scheduling (FPPS) [20]. FPDS is a generalisation of both: it splits up a task in arbitrarily many nonpreemptive subtasks with *preemption points* between them. *Preemption* is the temporary suspension of an executing task in favour of one or more newly arrived, higher priority tasks. Preemption points are locations inside a non-preemptive task that explicitly allow preemption, and thereby allow the granularity of preemptions to be selected by the programmer or compiler, finding a balance between the context switching overhead of FPPS and the coarse-grainedness of FPNS. Preemption points can be placed at positions which are deemed optimal, for efficiency reasons, and for access control of shared resources without the need for complex resource access protocols. The fact that subjobs are small leads to FPDS having a better response time for higher priority tasks than with FPNS.

RTAI is a free software community project that extends the Linux kernel with

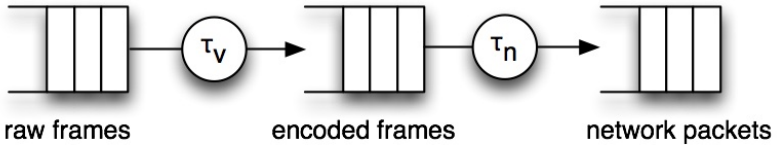

Figure 1.1: Tasks in the surveillance system

hard real-time functionality. It provides an Application Programming Interface (API) for programs that need hard timing constraints, and includes an FPPS scheduler for periodic and aperiodic tasks. The version of RTAI to be extended with FPDS is version `3.6-cv`.

1.2 Context & background

The Systems Architecting and Networking (SAN) group at the Computer Science department of the University of Technology is taking part in the ITEA2/CANTATA project. CANTATA (Content Aware Networked Systems Towards Advanced and Tailored Assistance) has a main goal of researching and developing a platform that is fully content aware and has understanding of the content that it is processing.[1] The SAN group is interested in the real-time aspects of this work, to gain insight in the requirements of a real-world system, and to validate the results of our research in a practical setting.

In particular, we are researching the following topics in the context of this project [16–18]:

- *FPDS* to reduce (context-switching) system overhead, and make use of its advantages in the area of exclusive resource access and implementation of mode changes;
- *Reservations* in the form of periodic, deferrable and sporadic servers in combination with FPDS, to guarantee resource availability;
- *Mode changes* to reallocate (memory) resources between scalable components in real-time systems.

FPDS was selected as a desirable scheduling mechanism for a surveillance system for monitoring a bank office, designed by an industrial partner. A camera monitoring the scene is equipped with an embedded processing platform running two tasks: a video task processing the raw video frames from the camera, and a network task transmitting the encoded frames over the network (See Figure 1.2). The video task encodes the raw frames and analyses the content with the aim of detecting a robbery. When a robbery is detected the network task transmits the encoded frames over the network (e.g. to the PDA of a police officer).

The advantages of FPDS for this system and industrial real-time systems in general are described in [16], which aims at combining FPDS with reservations to exploit the network bandwidth in a multimedia processing system from the surveillance domain, in spite of fluctuating network availability. In data intensive applications, such as video processing, a context switch can be expensive: e.g. an interrupted data transfer may need to retransmit the data when the transfer is resumed. Currently, in order to avoid the switching overhead due to arbitrary preemption,

[1] More details about CANTATA can be found at http://www.win.tue.nl/san/projects/cantata/ and http://www.hitech-projects.com/euprojects/cantata/

the video task is non-preemptive. Consequently, the network task is activated only after a complete frame was processed. The network task cannot transmit packets at an arbitrary moment in time (e.g. due to network congestion). Employing FPDS and inserting preemption points in the video task at convenient places will activate the network task more frequently than is the case with FPNS, thus limiting the switching overhead compared to FPPS and still allowing exploitation of the available network bandwidth. With response times found to be too long under FPNS, FPDS was considered to have the same benefits of lower context switch overhead compared to FPPS with its arbitrary preemptions.

1.3 Related work

In this section we will discuss some existing research and results that are related to our work: the area of cooperative scheduling in real-time systems.

1.3.1 Cooperative scheduling

FPDS is one form of a technique called *cooperative multitasking*, or *cooperative time-sharing* [11]. The general idea of this technique is that the time-sharing of a single resource (usually the CPU) is not fully regulated by a single arbitrator (e.g. the scheduler in the kernel), but programs or tasks work together on sharing the resource in a fair way, such that every contender can make progress. In the most common implementation of this technique, processes execute without interruption until they voluntarily cede time to other processes by offering a "de-scheduling" request to the kernel. The latter will then select a new process for execution according to some set policy, which will then run until it gives up its control of the resource as well. In this model the responsibility of scheduling is shared by the processes and the kernel, but other models where process selection is fully determined by programs themselves are also common.

Cooperative scheduling was one of the first techniques to be used to achieve multitasking, e.g. on interactive systems, before *preemptive scheduling* became the predominant mechanism. It has been used by many mainstream operating systems for personal computers, such as Microsoft Windows prior to Windows 95, and Mac OS, as well as operating systems targeted at server usage, such as Novell NetWare [23].

A description of non-preemptive scheduling in real-time systems, as well as an introduction to scheduling with deferred preemption along with corresponding response time analysis of tasks can be found in [11].

1.3.2 Earlier deferred preemption techniques

In [14], a *rate-monotonic with delayed preemption (RMDP)* scheduling scheme is presented by Gopalakrishnan and Parulkar. Compared to traditional rate-monotonic scheduling, RMDP reduces the number of context switches (due to strict preemption) and system calls (for locking shared data). One of the two preemption policies proposed for RMDP is *delayed preemption*, in which the computation time C_i for a task is divided into fixed size quanta c_i, with preemption of the running task delayed until the end of its current quanta. [14] provide the accompanying utilization based analysis and simulation results, and show an increased utilization of up to 8% compared to traditional rate-monotonic scheduling with context switch overheads.

Unlike [14], which introduces preemption points at fixed intervals corresponding to the quanta c_i, our approach allows to insert preemption points at arbitrary intervals convenient for the tasks. These locations can be selected based for example

on resource access control (surrounding critical sections), or minimal overhead of context switches due to memory cache misses.

In [22], Simonson and Patel investigate the optimal placement of preemption points with respect to memory cache behaviour. Through analysis of the usage of variables in memory by tasks, and the resulting cache behaviour in terms of cache hits and misses, they estimate the cost of context switches by preemptions when located throughout the task code. By optimizing a selection of preemption point locations they were able to get up to a 10% performance improvement by reduced context switch overhead in simulations, compared to traditional cache management techniques.

Zhou and Petrov [27] also investigate the cost of preemption points based on a compile-time analysis of the usage of processor registers, the *register liveness*. They introduced hardware support for preemption points, where the instruction locations of the selected preemption point locations are loaded in a hardware comparison register. The CPU then compares the loaded value with the instruction pointer for every executed instruction, automatically generating a trap to the preemption point handler at a preemption point location. This hardware based solution removes the need for additional code in program code along with associated overhead.

In [5], an implementation of delayed preemption in the LitmusRT research kernel is described, using a technique similar to the one we will describe in Chapter 7. Through the use of a shared flag between kernel and user space there is no need for expensive system calls between non-preemptive sections to check whether a preemption is required.

1.3.3 Swift mode changes

In [17, 18], M. Holenderski, R.J. Bril and J.J. Lukkien describe the advantages of the combination of *mode changes* and FPDS. In a resource constrained system composed of several scalable components, every component (e.g. a video encoder) is able to operate in one of multiple modes where each mode defines a *quality level* of the results it provides. Not all components can run in their highest quality modes at the same time, for example due to memory or CPU usage constraints. During runtime the system may decide to reallocate the resources between components, such that some components need to transition to a lower quality mode to free up resources for other components transitioning to a higher quality mode. It is shown how FPDS improves on the latency of a mode change, caused by mode change overhead, compared to a system scheduled by FPPS, because no resource access control is required (See Section 2.2.3).

1.3.4 FPDS worst-case response time analysis

R.J. Bril, J.J. Lukkien and W.F.J. Verhaegh [6, 7] correct the existing worst-case response time analysis for FPDS in [8, 12], under arbitrary phasing and deadlines smaller or equal to periods. They observe that the critical instance is not limited to the first job, but that the worst case response time of task τ_i may occur for an arbitrary job within an i-level active period. They provide an exact analysis, which is not uniform (i.e. the analysis for the lowest priority task differs from the analysis for other tasks) and a pessimistic analysis, which is uniform.

1.4 Approach

In our research, design and implementation leading up to, and achieving the goals of this Master thesis project, we followed an *incremental* approach. Early on in

the project, the lack of in-depth, up to date and accurate documentation of the design and implementation of RTAI was identified as a risk. While investigating the RTAI code base and surrounding information, we found that the (sparse) existing documentation on the design was often no longer reflecting the current versions of RTAI, and in many cases contradictory to other documentation and the source code itself. Because the implementation of FPDS was one of the main deliverables of the project, and a solid overview of the existing design was not available, we decided to design and implement FPDS in several iterations. The problem was divided into sub problems which were tackled one by one.

First, we investigated and described some FPDS task model alternatives, and how they could be mapped into an existing real-time operating system (RTAI/Linux) with support for periodic tasks. Because FPDS is a generalisation of FPNS, non-preemptive task support was then added to RTAI as a first step, and tested. This design and implementation was refined and extended, resulting in a basic FPDS implementation which was subsequently tested and benchmarked. This was followed by a more efficient implementation of FPDS, on which we conducted comparative measurements. Finally, we investigated some applications and extensions of FPDS as future work.

1.5 Contributions

The contributions of this project are:

- A discussion of the design alternatives of an FPDS extension in RTAI/Linux;
- A functional, extendable, maintainable and efficient implementation of FPDS in RTAI, with support for optional preemption points and a support library for real-time FPDS programs, which is freely available at `http://wiki.wikked.net/wiki/FPDS`;
- An evaluation of the implementation in terms of overhead and other key performance indicators (see Section 6.5);
- A discussion of how to do monitoring and enforcement of task budgets in an FPDS system.

Part of this work, i.e. the design and implementation of the basic FPDS extension in RTAI, resulted in a paper that was published in the proceedings of the OSPERT 2009 workshop held in conjunction with the Euromicro Conference on Real-Time Systems (ECRTS) conference in Dublin [3].

1.6 Overview

In Chapter 2 we recapitulate the theory of fixed-priority scheduling, and introduce the terms and notation we will use in the remainder of the document. Chapter 3 introduces RTAI/Linux and describes its architecture and some relevant implementation details. The rest of the structure of this document largely follows the approach we described in Section 1.4. First, some design considerations for our FPDS extension, including our assumptions, important design aspects and a discussion of the alternatives in mapping an FPDS task model onto a real-time operating system are discussed in Chapter 4. The design and implementation of our first step - non-preemptive task support - towards realising FPDS is described in Chapter 5, followed by the design, implementation and evaluation of basic FPDS in Chapter 6. Chapter 7 compares these results with a more efficient FPDS implementation. Some thoughts about extensions and applications of FPDS as future work are outlined in Chapter 8, followed by the conclusions of this project.

The work described in Chapters 2-6 is also described in our paper that was published in the proceedings of OSPERT 2009 [3].

Chapter 2

Fixed priority real-time scheduling

In this chapter we will lay out the foundation for the project, by providing a short summary of the existing basic theoretic concepts and corresponding definitions of real-time scheduling that are used in the rest of this thesis.

2.1 Basic theoretic task model

A *periodic task* performs a recurring job, which is released (i.e. made available) with a fixed period. Each *job* is an instance of a periodic task and has a *deadline* at which point in time the job must have completed execution. Often, but not always, this deadline is set to equal the release time of the next task instance, such that the job has completed before the subsequent job of the same task is released. In this thesis, a task with priority i will be referred to as τ_i, and the jth instance of task τ_i is written as $\iota_{i,j}$. Periodic task τ_i has period T_i, and *phasing* Φ_i, which is equal to the release time of the first job, $r_{i,0}$. The deadline of task τ_i relative to each job's release time is denoted as D_i, and the corresponding absolute deadline of job $\iota_{i,j}$ is $d_{i,j} = r_{i,j} + D_i$.

At some point in time after the release of a job $\iota_{i,j}$, the system's scheduler will assign the CPU resource to the task: it is *activated*[1]. The absolute time of the first activation of a job is denoted as $s_{i,j}$, which will occur after the job's release time: $r_{i,j} \leq s_{i,j}$. The goal of a real-time scheduler is to ensure that the job will finish at a time $f_{i,j}$ which is at most equal to the job's deadline: $f_{i,j} \leq d_{i,j}$. The difference between the finishing time and the release time is called the *response time* $R_{i,j} = f_{i,j} - r_{i,j}$. If the computation time of a task is known, it is represented by C_i^τ for a single job. Sometimes the computation time of a task varies or is only known to be bounded by an interval, in which case best-case and worst-case computation times of tasks are denoted by BC_i^τ and WC_i^τ.

The timing properties of a task τ_i are thus defined by the tuple [13]:

$$\tau_i = (T_i, C_i^\tau, D_i, \Phi_i) \tag{2.1}$$

An *aperiodic task* has similar properties as a periodic task, but it is not reoccurring with a known period. *Sporadic tasks* are special cases of periodic tasks, which have a guaranteed minimal inter-arrival time T_i.

[1] In this thesis we distinguish between *arrival time* or *release time* (the time when a job becomes available) and *start time* (the time when a job starts execution).

Task	Prio	T	Φ	C^τ
τ_1	1	9	8	2
τ_2	2	14	1	5

Table 2.1: Example task set of Figure 2.1

When multiple tasks are present in a system, they may want to make use of the same resources simultaneously, i.e. *share* them. If this is the case, and additionally tasks have timeliness constraints, for instance in a real-time environment, there needs to be a mechanism for resource sharing in place in order to ensure that all tasks can meet their constraints. One such shared resource is the CPU, and it is the primary contended resource in this thesis, which focuses on shared resource arbitration by *scheduling*.

2.2 Scheduling

Scheduling is the process of deciding how to commit resources between a variety of possible tasks by *time sharing*. In particular, a CPU scheduler determines at a given time which of the available tasks will be executing on a shared CPU. Many different strategies can be used to schedule tasks on shared processors, and they can be classified in several ways. Some possible classifications are:

- *static* scheduling based on fixed parameters versus *dynamic* scheduling based on parameters changing during runtime
- *offline* scheduling, i.e. a schedule that can be calculated on the entire task set before actual task activation, versus *online* scheduling where decisions are taken upon job releases and completions
- *preemptive* scheduling where jobs can be interrupted by higher priority jobs, versus *non-preemptive* scheduling where a job can run until completion or voluntary yielding

This thesis focuses on *static* (fixed-priority) scheduling only [13]. Fixed-priority scheduling is the most common scheduling algorithm in real-time systems, because it is simple to implement and its behaviour is well understood and backed by theory. Each task has an associated, fixed *priority* that is predetermined during development of the system and in principle does not change. A total order between priorities of tasks exists. In fixed-priority scheduling, as opposed to dynamic scheduling algorithms, the scheduler does not itself try to determine an optimal order of scheduling of tasks such that they (ideally) all meet their deadlines. Instead it simply tries to make sure that of all tasks that are ready to run, the task with the highest (fixed) priority has control of the CPU.

A system in which a task can be interrupted during its execution to assign a shared resource to a task having a higher priority, is called a *preemptive* system, and the action of interrupting a lower priority task for a higher priority one is called *preemption*. In a *non-preemptive* system this is not possible, and higher priority tasks being released have to wait until a lower priority task finishes or suspends execution, at which point the highest priority waiting task will receive control of the CPU.

Figure 2.1 shows a scheduled example task set of two tasks, with the task parameters shown in Table 2.2. In this scheduling example, low priority job $\iota_{2,1}$ is released first at time $t = 1$. Due to the period of the timer, interrupting the system every 2

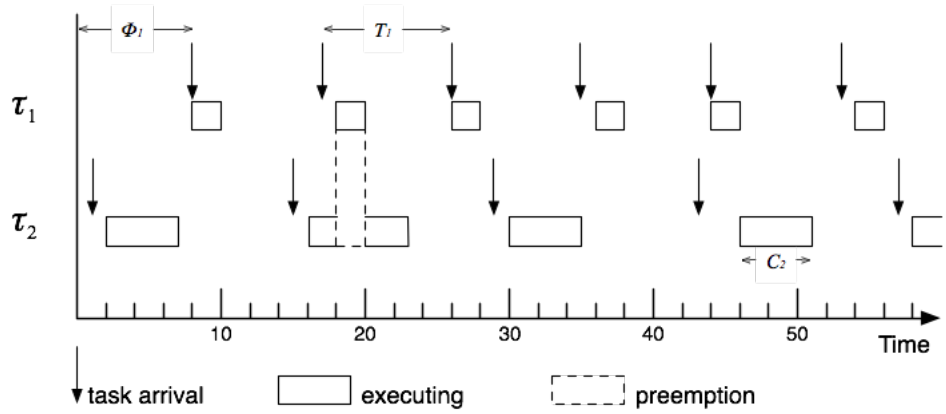

Figure 2.1: Example of a schedule with two tasks, and corresponding properties

time units, the job is activated one time unit later, when the timer interrupt fires and wakes up the new job at time $t = 2$. $\iota_{2,1}$ runs to completion without interference in 5 time units. At time $t = 8$, the first job of task τ_1 is released. Because this coincides with a timer interrupt, job $\iota_{1,1}$ is activated immediately and completes in 2 time units. At time $t = 15$, the period of task τ_2 has expired since its previous release, and job $\iota_{2,2}$ is released. One time unit later it is activated from the timer interrupt, and runs for 2 time units. In the mean time however, higher priority job $\iota_{1,2}$ has been released and gets activated at time $t = 18$, thereby preempting job $\iota_{2,2}$ in the process for $C_1^\tau = 2$ time units.

2.2.1 Fixed-Priority Preemptive Scheduling (FPPS)

Fixed-Priority Preemptive Scheduling, here after abbreviated as FPPS, ensures that at any moment in time, the highest priority task which is ready for execution is active, i.e. is given processor time. When during the execution of a task another, higher priority task is released, the executing task will immediately be preempted in favor of the higher priority task.

Figure 2.1 shows an example of fixed-priority scheduling with preemption, but also the notion of *activation jitter*, i.e. the delayed activation of tasks. Due to the design decision in most implementations of waking up task only during a (periodic) timer interrupt, tasks are often not activated immediately, but only when the next timer interrupt fires.

2.2.2 Fixed-Priority Non-preemptive Scheduling (FPNS)

A system implementing fixed-priority non-preemptive scheduling, or FPNS, never preempts tasks for higher priority tasks. Once a certain job is activated it will run until completion, unless it voluntarily gives up control of the CPU, or needs to wait on a blocking operation, e.g. a semaphore.

2.2.3 Fixed-Priority Scheduling with Deferred Preemption (FPDS)

A generalisation of FPNS is fixed-priority scheduling with *deferred preemption*. Instead of one uninterruptible job which can not be preempted, a job is divided into

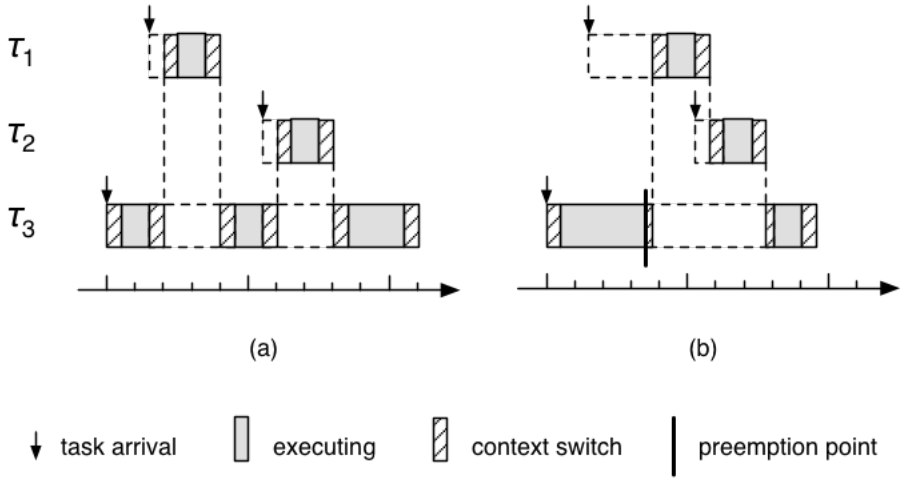

Figure 2.2: FPPS and FPDS scheduling compared

subjobs. Between subsequent subjobs there are *preemption points* where the job can optionally be preempted in favour of a higher priority task when necessary. FPNS is an instance of FPDS in which a job consists of only one subjob. Similarly, FPPS is in instance of FPDS with arbitrarily small subjobs. Like FPNS, FPDS is a form of *cooperative scheduling* [2].

The advantages of FPDS compared to FPPS are:

- Reduced system overhead due to less context switches
- Preemptions happen only at predefined preemption points, which can be placed at locations convenient or efficient for the task
- When preemption points are placed around critical sections, no resource access protocol for access to shared resources is necessary, as only one (sub)task will be executing a "critical section" at any given time.

Due to the finer granularity of (sub)tasks in FPDS, better response times and schedulability for higher priority tasks are achieved compared to FPNS.

Figure 2.2 shows an example of a task set scheduled by both FPPS (a) and FPDS (b). With FPPS, the highest priority task τ_1 immediately preempts task τ_2 on arrival. After its completion, task τ_3 resumes execution, only to be preempted again during the arrival of τ_2. When scheduled with FPDS (b) and after the addition of a preemption point in task τ_3, task τ_1 can only preempt when the first subtask of τ_3 has completed execution and is willing to be preempted. The amount of preemptions and context switches are reduced by one.

When preemption points are appropriately placed, e.g. at the end of a snippet of code that modifies a specific set of data, the overhead of context switches can be smaller than under FPPS with context switches happening at arbitrary locations.

Multiple relations between subjobs of a job can exist; in the simplest case their relationship is sequential and they are all executed completely in succession, without exceptions. However in support of more realistic code paths, subjobs can also form an *acyclic directed graph* corresponding to branches in code.

In some situations, the use of FPDS can also remove the need for a *shared resource access protocol*. FPPS has arbitrary interleaving of tasks, such that multiple

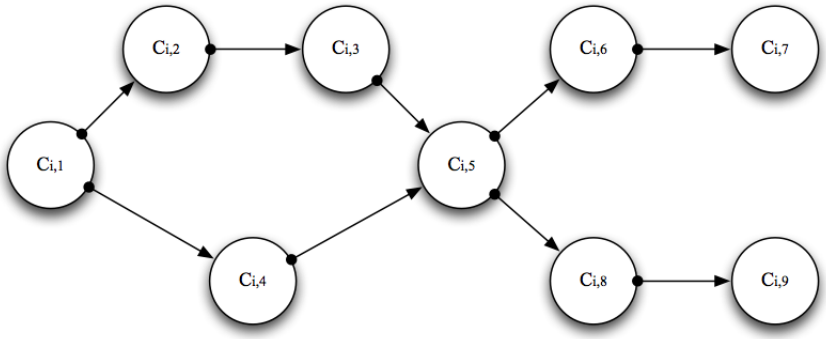

Figure 2.3: FPDS task with a DAG structure

tasks that access a certain resource may need to coordinate their access to maintain integrity of the resource and tasks accessing it. This is typically implemented using explicit *critical sections* that ensure that only one piece of code that shares a certain resource can be executing at any time. Under FPDS, at most one subtask is executing at any moment, so it may be possible to place preemption points such that code accessing a shared resource is fully contained within a single subtask. If FPDS is used like this on a platform with a single CPU, as is the assumption throughout this thesis, then there is no need to additionally guard the "critical section" with other resource access control primitives like *mutexes* or *semaphores*.

In [16], the notion of *optional preemption points* is introduced, allowing a task to check if a higher priority task is waiting, which will preempt the current task upon the next preemption point. For example, if a higher priority task is already pending, then the running (sub)job may decide to finish its work but *adapt its execution path*, and e.g. refrain from initiating a data transfer on an exclusive resource that is expensive to interrupt or restart. Optional preemption points rely on being able to check for pending tasks with low overhead, e.g. without invoking the scheduler.

Task model extensions for FPDS

For FPDS, we need to refine the basic model of Section 2.1.

A job of task τ_i has computation time C_i^τ, and since under FPDS a job $\iota_{i,j}$ is subdivided into K_i subjobs $\iota_{i,j,0}, \iota_{i,j,1}, \ldots, \iota_{i,j,K_i-1}$, the computation time of a subtask k is defined as $C_{i,k}^\tau$. In a sequential relation between subjobs, the computation time of the task simply equals the sum of all subjobs:

$$C_i^\tau = \sum_{0 \leq k < K_i} C_{i,k}^\tau \tag{2.2}$$

Later in this thesis we will refer to the *measured* computation time of a *job* and *subjob* as $C_{i,j}^\iota$ and $C_{i,j,k}^\iota$ respectively.

When the structure of τ_i is not sequential but has the form of a DAG with branches (see Figure 2.3), then C_i^τ is dependent on the path used through the graph, during execution. The worst-case computation time of C_i^τ is the sum of the longest path in the graph, when $C_{i,k}^\tau$ is used as the cost factor.

We define $WC_{i,k}^\tau$ and $BC_{i,k}^\tau$ as the worst-case and best-case computation times of subtask $\tau_{i,k}$ of task τ_i, respectively. Then, the following should hold:

$$BC^{\tau}_{i,k} \leq C^{\iota}_{i,j,k} \leq WC^{\tau}_{i,k} \tag{2.3}$$

where $C^{\iota}_{i,j,k}$ is the *actual execution time* of subjob $\iota_{i,j,k}$. We distinguish this value with $C^{\tau}_{i,k}$ because the latter is theoretical for a given subtask, whereas $C^{\iota}_{i,j,k}$ can be a value measured for a given subjob in a running system.

For the general case of a task τ_i having a subtask structure in the form of a DAG, we define $BCR^{\tau}_{i,k}$ as the best-case computation time of the *remainder* of task τ_i, for all valid paths in the DAG *succeeding* subtask $\tau_{i,k}$, but not including it.

Throughout this thesis, the term *subtask* will be used to denote a part of a task, and like a job is an instance of a task, a subjob is an instance of a subtask. Any job consists of a sequence of subjobs which are executed in succession. Subtasks are part of a task, however any given execution instance of a task (i.e. a job) may not necessarily execute all subtasks, if the subtasks have a non-sequential relation due to branching. Likewise, subjobs unlike jobs do not get *released*; the code of the job itself decides at every preemption point which next subjob will be executed.

A detailed analysis of the response time of tasks scheduled by FPDS can be found in [6, 7].

Chapter 3

RTAI & Linux

The real-time operating system to be extended with FPDS is *RTAI*: Real-Time Application Interface. RTAI is an extension to the Linux kernel, which enhances it with hard real-time scheduling capabilities and primitives for applications to use this. In this chapter we will provide a brief introduction to the architecture and relevant implementation details of both RTAI and the interface with Linux.

3.1 Description

RTAI's goal is to extend Linux with real-time primitives and a scheduler for hard real-time behaviour, which functions as a separate layer next to the standard Linux environment, such that Linux programs can voluntarily choose whether to use RTAI functionality. One of the largest applications of RTAI is machine control, where deterministic timing behaviour is very important. The design of RTAI, being a *hypervisor* [24], is explicitly chosen to avoid intrusive changes to the entire Linux kernel code base, in the interest of maintaining compatibility with Linux which is following a very rapid development cycle.

A rough representation of the architecture of RTAI/Linux is shown in Figure 3.1. At the bottom, right on top of the hardware, a Hardware Abstraction Layer (HAL) is installed by RTAI, which prepares the Linux source code for running under a hypervisor (see the next section). On top of this, RTAI and Linux sideways of each other. All RTAI specific modules are shown as green in the diagram. Making use of the HAL are the modules in RTAI that take care of interrupt handling (the Interrupt Service Routines), scheduling, task creation etc., as well as Inter-Process Communication (IPC) and resource access control. This infrastructure can be accessed by real-time (kernel) tasks via the *API* on top, providing a well-defined interface.

All RTAI modules just describe run in kernel mode (see Section 3.4), where they can only be accessed by code running in kernel mode as well, including real-time programs. Because RTAI also supports hard real-time scheduling for programs running in user mode, an additional layer is needed that provides a user-mode gateway to the RTAI kernel API: the *LXRT* API.

The top layer (in red) represents all programs running in the system, both normal non-real-time tasks running under Linux, as well as (hard) real-time tasks scheduled by RTAI.

Although RTAI does not strictly adhere to this architecture, the diagram presents a non-transitive top-to-bottom *uses* relation, where high level modules make use of modules (immediately) below them, and not vice versa. Modules on the same level cooperate with each other where necessary. RTAI maintains most of its *state* in

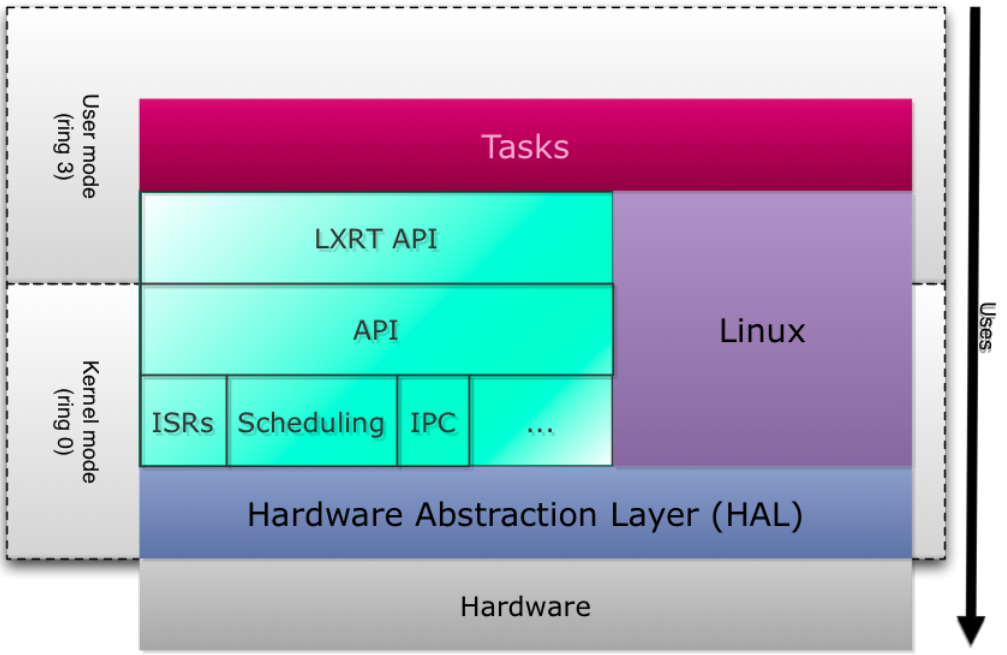

Figure 3.1: Architectural diagram of RTAI/Linux

global data structures such as *linked lists* of *structs*, that can be accessed from anywhere in the code running in kernel mode. (This is not shown in Figure 3.1.)

3.1.1 Hypervisor

RTAI provides hard real-time guarantees alongside the standard Linux operating system by taking full control of external events generated by the hardware. It acts as a *hypervisor* between the hardware and Linux. When RTAI modules are loaded, the *HAL* inserts itself between the hardware interrupts (including the timers) and the normal Linux interrupt handlers, in order to gain full control of interrupt processing. It also intercepts certain function calls made by Linux kernel code to disable and re-enable hardware interrupts (`cli` and `sti`), such that it can alter its state accordingly.

Since RTAI then receives all external events from the hardware, it can directly control how the software reacts to these interrupts, including if and when the rest of the Linux kernel receives them for processing. Using the signals from the timers RTAI does its own scheduling of real-time tasks and is able to provide hard timeliness guarantees.

After RTAI modules are unloaded again, full interrupt control is handed back to the Linux kernel such that interrupts are directly handled by Linux interrupt handlers, as in a normal, unpatched situation.

Although RTAI has support for multiple CPUs, we choose to ignore this capability in the remainder of this document, and assume that our FPDS implementation is running on single-CPU platforms. Systems with multiple CPUs are not yet common in the field of real-time systems, partly because of the complicated analysis of task sets in a multi-CPU platform, for which research results are not as mature yet as for single-CPU systems. Additionally, the advantage of using FPDS as a

resource access control method disappears when multiple subjobs can be running simultaneously, as is the case in a multiple CPU system.

3.1.2 The scheduler

RTAI Linux system follows a *co-scheduling* model: hard real-time tasks are scheduled by the RTAI scheduler, and the remaining idle time is assigned to the normal Linux scheduler for running all other Linux tasks, i.e. Linux is treated as a *background job*. The RTAI scheduler also supports the standard Linux *schedulables* such as (user) process threads and kernel threads, and can additionally schedule RTAI kernel threads. These have low overhead but they cannot use regular OS functions.

The scheduler implementation supports preemption, and ensures that always the highest priority runnable real-time task is executing.

Primitives offered by the RTAI scheduler API include periodic and non-periodic task support, multiplexing of the hardware timer over tasks, suspension of tasks and timed sleeps. Multiple tasks with equal priority are supported but need to use cooperative scheduling techniques (such as the `yield()` function that gives control back to the scheduler) to ensure fair scheduling.

3.1.3 Timers

For multiplexing of tasks RTAI offers a choice between two different timer modes: *periodic* and *oneshot*. In periodic mode the timer is programmed once with a certain frequency, and will generate an interrupt at that rate, activating the scheduler and possibly performing a context switch to another task. This mode has a very low setup overhead as the timer only needs to be programmed once, but task activations are only possible at multiples of the timer period. The alternative mode is to program the timer once for every iteration, with the timer generating only a single interrupt when the time expires (one shot). This allows arbitrary timings, but comes with an additional overhead cost of reprogramming the timer after every timer interrupt. Depending on the specifics of the task set, the programmer can select the more optimal mode to use, trading programming overhead of one-shot timers against activation jitter and interrupt overhead of periodic timers.

3.2 Implementation of the basic theoretic model

Since we are interested in implementing FPDS either inside or on top of RTAI, this section gives a comparison of the basic theoretic model and how it can be applied to the actual existing RTAI implementation.

RTAI directly supports the notion of tasks (τ_i) along with associated priorities. Tasks are instantiated by creating a schedulable (typically a thread) using the regular Linux API, which can then initialize itself as an RTAI task using the RTAI specific API. Priorities are integers, $0 \leq i < 65536$, where 0 is the highest priority. Multiple tasks sharing the same priority are supported by RTAI as well.

As discussed in the previous section, RTAI allows tasks to be scheduled periodically and aperiodically. The start time, and thereby also the phasing Φ_i of a task can be set as an absolute value, or relative to the current time. Through the use of the oneshot timer, RTAI allows arbitrary phasings to be used, although for performance reasons it may be beneficial to use the periodic timer.

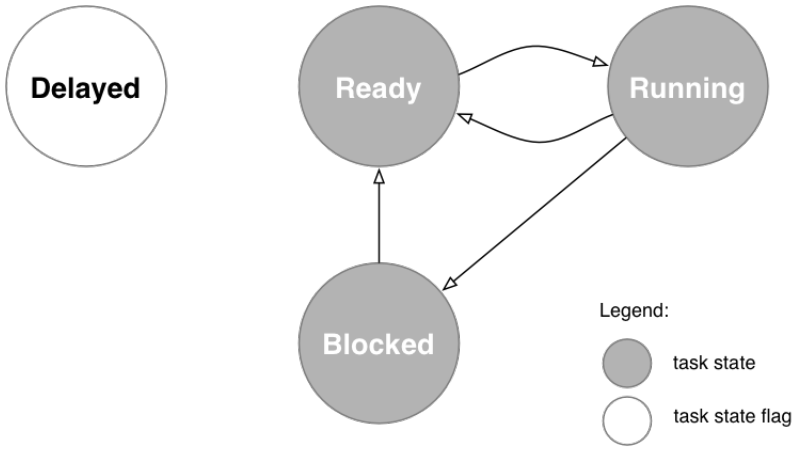

Figure 3.2: RTAI task states and flags

3.3 Tasks in RTAI

Although the terminology of *jobs* is not used in RTAI, all necessary primitives to support periodic tasks with deadlines less than or equal to periods are available. Repetitive tasks are typically represented by a thread executing a repetition, each iteration representing a job. An invocation of the `rt_task_wait_period()` scheduling primitive separates successive jobs. Through a special return value of this function, a task will be informed if it has already missed the time of activation of the next job, i.e. the deadline equal to the period.

In each task control block (TCB) various properties and state variables are maintained, including a 16 bit integer variable representing the running state of the task. Three of these bits are used to represent mutually exclusive running states (*ready*, *running*, *blocked*). The remaining bits are used as boolean flags that are not necessarily mutually exclusive, such as the flag *delayed* (waiting for the next task period), which can be set at the same time as *ready* in the RTAI implementation. This implies that testing the *ready* state is not sufficient for determining the readiness of a task. See Figure 3.2 for a (simplified) overview of the task states relevant for our work.

3.3.1 Implementation of periodic tasks

A common method to implement periodic tasks in RTAI makes use of the `rt_wait_period()` primitive which suspends the calling task until the end of its period. The imple-

mentation of a periodic task typically reflects the following pseudocode:

```
function task_thread() {
    # task initialization
    rt_init_task()
    rt_make_periodic(period, phase)
    rt_make_hard_realtime()

    # task / jobs loop
    j = 0
    while (true) {
        # Invoke job instance j
        task_job(j)
        # Wait until the next periodic task activation
        rt_wait_period()
        j++
    }
}

function task_job(instance) {
    # Perform job
}
```

3.4 Privilege separation and system calls

Like many operating systems, Linux makes use of the *privilege ring* features of modern CPUs, where the kernel runs in the most privileged ring 0 and has full access to all hardware, whereas the applications run in a less privileged ring with restricted access to hardware. These are commonly called *kernel mode* and *user mode*, respectively [26]. RTAI, inserting itself between the Linux kernel and the hardware, also runs in ring 0 (kernel mode) and thus has full control of the hardware.

The kernel also reserves part of the physical memory for itself, which is protected from access by user mode programs. This memory space is called *kernel space*, and in Linux is always present in physical memory. In contrast, the (remaining) memory used by normal programs is *user space*, and can be *paged out* to (slow) external storage when available physical memory is low. Each Linux process runs in its own *address space* and accesses its memory using *virtual addresses* which have an indirect mapping to real, physical memory. This provides protection between processes, because they can't access each other's memory.

When a user mode program wants to invoke a kernel routine or needs access to data in kernel memory, it cannot call or access it directly, but needs to perform a *system call* (or *syscall*). On behalf of the calling process, the system will change mode from user mode to kernel mode, execute the requested kernel routine, and return to user mode. The kernel will copy any required data in memory from user space to kernel space or vice versa, if required. The extra time this process costs is considerable *overhead*.

3.4.1 Implementation details

RTAI's implementation of system calls from user mode programs to its in-kernel API routines is based on the system call implementation of the Linux kernel. In this section we will describe what happens when a user mode real-time task executes an RTAI syscall, as this process and its overhead is relevant in the FPDS implementation that will be described in the upcoming chapters.

Real-time programs that run in user mode link to a *dynamic library* called `liblxrt` to access RTAI functionality. For every RTAI API primitive that resides in kernel mode, a corresponding *wrapper function* with the same semantics exists in the `liblxrt` library. Every one of these functions does essentially the same thing: it packs all function arguments into a single `struct`, and passes it along with the total byte length of all parameters and a numeric identifier of the desired (in-kernel) API primitive to the `rtai_lxrt()` function, which will invoke the system call. After this function finishes, the return value of the system call will be unpacked and passed back to the real-time program.

Function `rtai_lxrt()` on its turn encodes its passed arguments (i.e. the RTAI API call number and the byte size of its arguments) into a single 32 bit integer. It then calls the Linux function `syscall()` with Linux system call number `RTAI_SYSCALL_NR`. All RTAI API calls are thus executed by wrapping them in a single Linux system call invocation, which executes a kernel mode routine installed by RTAI.

The implementation of the Linux `syscall()` function differs per CPU architecture and CPU model. On the *i386* architecture that we used for this project, the actual switch from user mode into kernel mode is generally implemented using a *software interrupt*, `int 80h`. When this instruction is executed, the CPU acts as if a normal hardware interrupt arrived with hexadecimal number 80, and internally changes the privilege mode from ring 3 to ring 0. It then invokes the Interrupt Service Routine (named `system_call()` in Linux) for interrupt 80h that was installed by the operating system.

Before executing the software interrupt, the arguments of the system call, including the requested system call number, are placed in the *processor registers*. The first action of the ISR is therefore to store this information on the kernel stack. It then stores the contents of all other CPU registers, for later restoration after the system call finishes. After some validity checks on e.g. the the system call number argument, the appropriate system call kernel function is invoked, which normally executes the system call. The RTAI LXRT specific system call function acts as an additional layer of dispatching: it decodes the first argument passed which includes the RTAI API call number, and calls the desired in-kernel RTAI API routine. The requested RTAI API action is then executed in kernel mode.

After the routine finishes, the same path is followed in reverse order, back from kernel to user mode. Return values are passed on between functions, and put in the CPU registers for the privilege mode switch, until it is eventually passed as the return value of the original `liblxrt` function back to the real-time application.

The majority of the overhead incurred in this process is present inside the actual mode switch inside the software interrupt, executed inside the CPU itself. This can last over $1\mu s$, depending on the CPU model. The rest of the overhead is caused by the various iterations of copying and encoding arguments between functions and the mode switch, which are relatively slow memory operations.

A more detailed description of the implementation of system calls in Linux can be found in [4].

3.5 Scheduler implementation

In order to provide some context for the design decisions and implementation considerations that will follow, we briefly describe the implementation of the existing RTAI scheduler.

RTAI maintains a *ready queue* per CPU, as a *priority queue* of tasks that are ready to run (i.e., *released*), sorted by task priority. A task remains on the ready queue when it is executing. Periodic tasks are maintained with release times of their

next job in a separate data structure, the so-called *timed tasks*. This data structure can be an ordered linked list or a red-black tree. If at any moment the current time passes the release time of the head element of the timed tasks list, the scheduler migrates this task to the ready queue of the current CPU. In practice this does not happen instantly but only upon the first subsequent invocation of the scheduler, e.g. through the timer interrupt, and therefore having a maximum latency equal to the period of the timer. The scheduler then selects the head element from the ready priority queue for execution, which is the highest priority task ready to run. The currently running task will be preempted by the newly selected task if it is different. The scheduler ensures that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute, and therefore it is a FPPS scheduler.

When a job finishes and invokes the `rt_wait_period()` primitive, the task is migrated back from the ready queue to the timed tasks list, waiting for the expiration of the period.

The implementation of the scheduler is split over two main scheduler functions, which are invoked from different contexts, but follow a more or less similar structure. It remains somewhat unclear why the functionality common to both functions is not factored out. The function `rt_timer_handler()` is called from within the timer interrupt service routine, and is therefore time-triggered. The other function, `rt_schedule()` is event-triggered, and performs scheduling when this is requested from within a system call. Each of the scheduler functions performs the following main steps:

1. Determination of the current time
2. Migration of runnable tasks from the timed tasks queue to the ready queue
3. Selection of the highest priority task from the ready queue
4. Context switch to the newly selected task if it is different from the currently running task

After a new task is selected, the scheduler decides on a context switch function to use, depending on the type of tasks (kernel or user space) being switched in and out. The context switch is then performed immediately by a call to this function.

These steps are shown as a flow diagram in Figure 3.5.

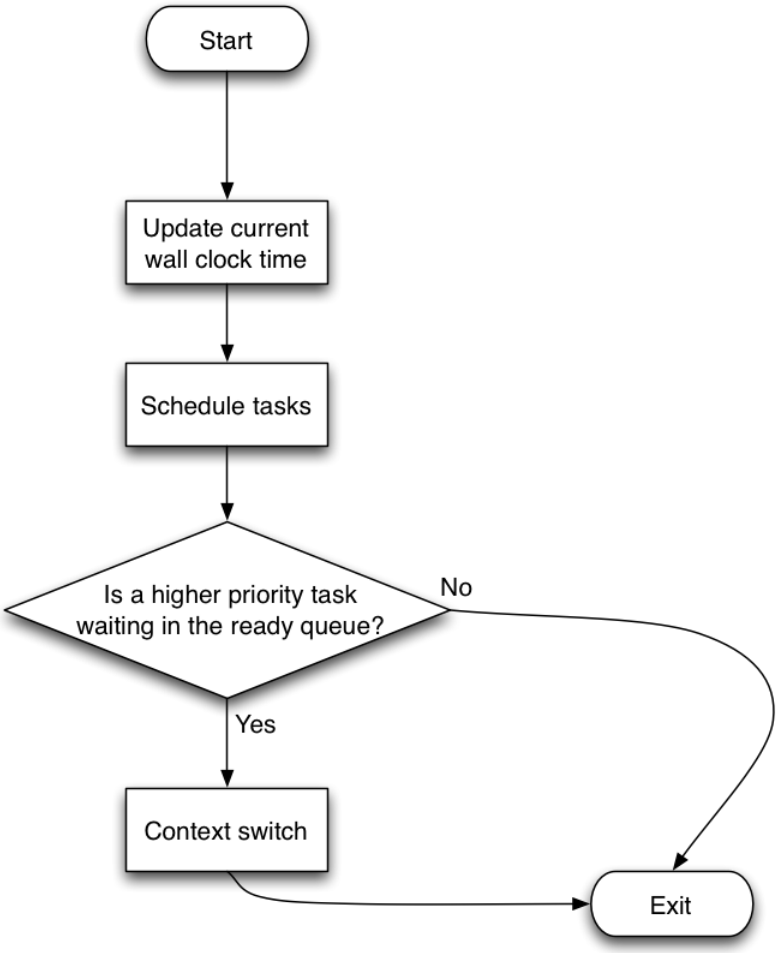

Figure 3.3: Flow diagram of the standard RTAI FPPS scheduler logic

Chapter 4

Design considerations

In this chapter we will discuss our design considerations leading up to our implementation of FPDS in RTAI. First, we list a number of assumptions we've made in our design, followed by a discussion of the possible task models for FPDS. We conclude with a number of design aspects that we consider important and that will be used later to check our implementation.

4.1 Assumptions

We will base our design of the implementation of FPDS in RTAI/Linux on a number of *assumptions*. In situations where these assumptions do not hold, our design and implementation of FPDS may not be optimal.

First, we assume that real-time applications that want to use FPDS in RTAI/Linux, will run in *user mode*. Although RTAI is also able to schedule tasks in kernel mode, we think that the functionality provided by the Linux kernel to normal user mode processes, as well as the isolation between user mode processes in different address spaces are strong advantages of this RTOS platform over others, despite the overhead introduced by these features. When using tasks scheduled in kernel mode, many of these advantages are not available, and then smaller and simpler real-time operating systems could be used instead, which do not use privilege separation. Therefore we will focus on real-time applications running in user mode in this thesis, although kernel tasks will be supported by our implementation as well.

Secondly, we assume that the overhead of context switches of FPPS, when compared to FPDS, is caused primarily by the preemption by other tasks, and not by interrupts. Although the execution of an interrupt service routine constitutes a context switch as well, it is minimal when compared to a process switch. An Interrupt Service Routine (ISR) is explicitly written to be small, efficient and non-intrusive: it does not require a memory address space switch. It operates on small amounts of memory only and executes its work as quickly as possible to not interfere with the rest of the system. Therefore there can be less overhead caused by memory cache misses and Translation Lookaside Buffer (TLB) flushes compared to regular tasks. For an overview of system overheads in context switches, see [15].

Because of the need of the system to keep *wall clock time*, we assume that in typical cases, we cannot disable the timer interrupt during the execution of non-preemptive (sub)tasks. In RTAI/Linux and most other operating systems, wall clock time is maintained using a periodic timer that fires at a steady rate.

4.2 Task model selection

Before discussing any implementation details of FPDS in RTAI, decisions have to be made about the *task model* to be implemented. A variety of options exist, the advantages and disadvantages of which will be discussed here.

4.2.1 Subtasks as normal system tasks

Since FPDS is a generalisation of both FPPS and FPNS by dividing a job into arbitrary length subjobs, one possibility for implementing FPDS is to follow the same train of thought: instead of creating a single task in the system, a task can be created for every subjob, or subtask, which should then be grouped in some way to maintain the notion of a single task with corresponding scheduling behaviour. The advantages of this approach lie in the fact that the system and its scheduler have full information about the existence of all subtasks, along with their (worst case) computation times and other parameters, and by the simplicity by which this method can be implemented in the case of subtasks with sequential order.

After the completion of a subjob, control is returned to the system scheduler, which can then schedule the next subjob. If the subtasks are in a sequential relation to each other, the scheduler can follow a simple ordering of system tasks, for instance using a fixed linked list, or by using synchronisation primitives such as semaphores in which case no source modifications to the scheduler are needed.

There are many disadvantages to this approach however. Because essentially every subtask becomes a separate task as it is in a regular FPNS system, subtasks are very isolated from each other, disrupting the normal code flow and context of a task. Subtasks will probably have to be separate functions, and run in different execution contexts (i.e. separate threads or processes), making it hard to access shared data in a convenient way. Although it *may* not be impossible to largely hide it from the programmer, the isolation between subtasks and the additional burden induced by it is likely to reduce the incentive to create preemption points within a task.

Another important observation to make is that a strong *precedence relation* exists between all subtasks of a task. Generally, ignoring timing constraints, regular tasks can be scheduled in any order, i.e. with arbitrary interleavings. This is sometimes restricted by shared access to resources or data where one task is blocked while another task is using a shared resource in a critical section, or when a task is waiting for the results from another task before it can start execution. The remaining freedom of possible interleavings of tasks is utilized by a (real-time) scheduler to ensure that certain criteria (such as meeting deadlines) are met. However this freedom does not exist for subtasks of a task: the order of execution of subtasks is fixed, and the scheduler is, with respect to subtasks, essentially little more than a *dispatcher*. This is a result from the fact that there is no additional scheduling freedom in the task set with subtasks compared to the original FPNS task set.

In the simple case where all subtasks of a task have a strict sequential order, the interleaving of subtasks within a certain task is fixed when ignoring other tasks. This is no longer necessarily the case with a task where subtasks have dependencies defined by a directed acyclic subgraph, where the order (and selection) of subjobs may be different for every job run. But again, it is not the scheduler which determines this; the order of FPDS subjobs is determined by the *program flow*, i.e. decisions made within the code of the task. Therefore, in order for the system scheduler to be able to activate the correct subtask, the result of this decision needs to be transfered from the task to the system scheduler, when it becomes available. Depending on the specific implementation, this could incur a large amount of extra overhead, if for instance a system call is required to pass this information.

At the end of a subjob, control is always returned to the system scheduler which then dispatches the next subjob. Consequently, at every preemption point a *context switch* occurs without exception. This implies that efficient *optional preemption points*, as we will implement in Chapter 7, are not possible in this model. This defeats a large part of the advantage of FPDS: the reduced overhead of preemptions.

4.2.2 Subtasks divided by yield points

Many real-time and non-real-time operating systems already implement a primitive that can be used almost directly for FPDS: a `yield()` function. When a currently running task τ_i calls `yield()`, it signals the kernel that it voluntarily releases control of the CPU, such that the scheduler can choose to activate other tasks before it decides to return control to the original task, according to the scheduler algorithm. When used in a non-preemptive (FPNS) system, a `yield()` function thus implements a preemption point in the definition of FPDS: because the scheduler is invoked, higher priority tasks $\tau_{0..i-1}$ that have become available since the previous preemption will be activated by the scheduler before the original task is reactivated. If no higher priority tasks are available the scheduler will immediately pass back control to original task τ_i. If other tasks with equal priority are ready, they are given precedence.

This approach has some advantages when compared to the intrusiveness of the method in Section 4.2.1, and has the potential for lower overhead. With preemption yield points a task does not need to be split up in multiple subtasks with separate functions and separate threads. The design and structure of a task are not as heavily influenced by the needs of the FPDS implementation; preemption points can easily be added and removed by placing a single function call at appropriate locations. This causes a lower burden on the programmer, and is therefore more likely to be used effectively in practice. Additionally, automated tools can generate preemption points transparently to the programmer in the background if desired, guided by other primitives and cues in the source code such as critical sections.

Some of the added overhead of the subtasks scheduling of Section 4.2.1 can also be avoided. There is no need for the system scheduler to do scheduling of subtasks within a task, as the execution will simply follow the code flow of the task over subtask boundaries, the preemption points. The transfer of branching information to the system scheduler to select the next subtask is also unnecessary. Furthermore, the system does not need to store the potentially large amount of information that is normally maintained for each *schedulable*. If there are many subtasks this can imply a large memory saving.

This model does not necessitate a context switch at the end of each subtask, and thereby opens the opportunity for *optional preemption points*, where the task is only preempted if other, higher priority tasks are ready for execution. If the implementation of the real-time operating system is such that it is possible to determine this from the execution context of the running task in user space, then optional preemption points can be implemented efficiently.

The biggest drawback to implementing FPDS using yield functions is however that the system has no knowledge of subtasks and their properties, and therefore cannot do any analysis on them to make improved scheduling decisions compared to FPNS. It is consequently also not possible in this model to support a mixture of FPNS *and* FPNS subtasks within a task.

Although this model supports substantial efficiency gains over the one of Section 4.2.1, a potentially larger improvement can be made with optional preemption points, as the substantial cost of context switches at every subtask boundary can be avoided in many cases.

4.2.3 Preemption yield points with subtask properties

The two solutions presented above are rather different to one another, and complement each other in their advantages and disadvantages. The model with full system tasks of Section 4.2.1 trades efficiency and implementation flexibility for more information at the system level, whereas the model of Section 4.2.2 does the opposite. It is possible to alter and improve on these models to reduce or eliminate some of their disadvantages, essentially bringing both models closer to each other. For example, it could be advantageous to make the system subtasks of section 4.2.1 more lightweight, by creating special purpose subtasks in the system, only keeping the essential data required for scheduling subtasks, as well as more lightweight context switches and scheduling routines. However because we feel that the other proposed model in its purity more closely fits the needs and intentions of FPDS, we will work from that direction.

The primary drawback of the method using yield points is the lack of any information about the structure and properties of all subtasks available to the system scheduler, and/or other components in the system. The goal is thus to build upon a plain yield function implementation to make this information available in an efficient manner without the unnecessary overhead that creating system (sub)tasks entails.

When looking at the structure of a task and the relation between its subtasks, we see that the order relation of these subtasks does not change over the course of the lifetime of the task, whether they follow a sequential order or a directed acyclic graph. An example of such a directed acyclic graph is shown in Figure 2.3. At task creation this graph[1] can be supplied and stored as a property of the task, in the task's own memory address space for processing at preemption points, or even in the Task Control Block (TCB) if access to it is required by the kernel. Each node in the graph represents a subtask, and each edge is a possible transition to another subtask. For every node, at least the following attributes should be present:

- The subtask identifier (e.g. $\tau_{i,j}$, or simply j)
- The subtask computation time $C_{i,j}$

In addition to this some analysis on the task can be done offline to facilitate an efficient implementation of the scheduler in an FPDS system. For example, the *worst case computation time* of the graph WC_i^τ can be computed and stored for later use. Because every subtask node is the root of a subgraph, the worst case response time $WC_{i,k}^\tau$ of the remainder of the task at that subgraph can be stored as an attribute of the subtask node. This can be done offline or at task creation time before the task's deadlines can be missed due to scheduler/preemption overhead.

4.3 Design aspects

While designing the implementation of our chosen FPDS task model, we have a number of aspects that lead our design choices. First of all, we want our implementation to remain *compatible*; our extensions should be conservative and have no effect on the existing functionality. Any FPDS tasks will need to explicitly indicate desired FPDS scheduling behaviour. *Efficiency* is important because overhead should be kept minimal in order to maximize the schedulability of task sets. Therefore we aim for an FPDS design which introduces as little run-time and memory overhead as possible. Due to the need of keeping time, we do not disable interrupts

[1] As a sequential ordering is a special case of a directed acyclic graph and we intend to support both forms in this implementation, we will only mention the generic form from now on.

during FPDS tasks, so the overhead of *interrupt handling* should be considered carefully as well. Because we want to be able to integrate our extensions with future versions of the platform, our extensions should be *maintainable*, and written in an *extendable* way, with flexibility for future extensions in mind.

Chapter 5

FPNS

The process of implementing FPDS in RTAI/Linux was done in several stages. Because the existing scheduler in RTAI is an FPPS implementation with no direct support for non-preemptive tasks, the first stage consisted of a proof of concept attempt at implementing FPNS in RTAI. The following stages then built upon this result to achieve FPDS scheduling in RTAI in accordance with the task model and important design aspects described above.

The existing scheduler implementation in RTAI is FPPS: it makes sure that at every moment in time, the highest priority task that is in *ready* or *running* state has control of the CPU. In contrast, FPNS only ensures that the highest priority ready task is started upon a job finishing, or upon the arrival of a task whenever the CPU is idle. For extending the FPPS scheduler in RTAI with support for FPNS, the following extensions need to be made:

- Tasks, or individual jobs, need a method to indicate to the scheduler that they need to be run non-preemptively, as opposed to other tasks which may want to maintain the default behaviour.
- The scheduler needs to be modified such that any scheduling and context switch activity is deferred until a running non-preemptive job finishes.

Alternatively, arrangements can be made such that at no moment in time a ready task exists that can preempt the currently running FPNS task, resulting in a schedule that displays FPNS behaviour, despite the scheduler being an FPPS implementation. Both strategies will be explored.

Our FPNS implementation is non-preemptive only with respect to other tasks; i.e. a task will not be *preempted* by another task, but can be *interrupted* by an interrupt handler such as the timer ISR.

5.1 Implementation using existing primitives

Usage of the existing RTAI primitives for influencing scheduling behaviour to achieve FPNS would naturally be beneficial for maintainability of our implementation. When only supported API interfaces are used with the desired result, there is no need for modifications to the RTAI core source. This would reduce the probability of problems with future incompatibilities and merge conflicts with the associated maintenance overhead: as long as the API of RTAI primitives to programs utilizing them stays backwards compatible, the implementation will remain in working state.

Modifying the *ready* state of non-preemptible tasks

When investigating the RTAI scheduler primitives that are explicitly exported to user programs and documented for such use [21], we find several different methods to effectively achieve FPNS behaviour in our system. One feasible method involves making sure that all other tasks are taken out of the *ready* state while a designated FPNS task is running. Using RTAI primitives, this could for example be done by suspending all higher priority tasks during the execution of a job, using `rt_suspend()` and `rt_resume()`, or by using mutual exclusion primitives such as semaphores [21]. This is clearly not a very practical approach however: each task requires knowledge of the existence of every other higher priority task in the system to be able to suspend and resume it. Furthermore, since only one task can be suspended or resumed at a time, this method has a large amount of overhead of at least $2N$ system calls per FPNS task τ_t, where $N = \#_{\tau_i}(i < t)$, the number of higher priority tasks in the system. The modifications to the schedulability of a task and changes to the ready queue also imply that the system scheduler is rerun at every invocation of these primitives, further increasing the overhead, and making this method very unattractive. Finally, a mechanism needs to be put in place ensuring that newly arriving tasks that are higher priority than any currently running FPNS task cannot cause interference, as they have not been suspended prior to the execution of this task.

Task priority changes

A better option is offered by the ability of RTAI tasks to change their *base priority*. At the start of a job the priority can be raised to a value which is equal to the maximum priority supported by the system, or higher than any current or future tasks: the *maximum* of all task priorities, plus one. This ensures that the current FPNS task will not be preempted by a higher priority task as by definition, a higher priority task does not exist for the duration of the executing job. After the job finishes, the base priority should be lowered again to its original value. In terms of overhead the advantages of this method over suspension of higher priority tasks are clear: for each FPNS job only a single pair of priority increase and decrease system calls is required, which is constant $\mathcal{O}(1)$ overhead. There are however drawbacks to this approach as well. The modification of base priorities of tasks during job execution conflicts with the theory of Fixed Priority Preemptive Scheduling and may cause problems with priority based protocols, particularly in the area of resource access control protocols which deal with priority inversion [25].

Locking the scheduler

Although it is not found in any public documentation, it turns out that RTAI also has the primitives `rt_sched_lock()` and `rt_sched_unlock()` available to user programs, which could be used in a similar way to the temporary priority changes described above, but without the corresponding drawbacks that priority modifications present. When the scheduler is locked during the execution of a FPNS job, no rescheduling will occur and consequently no preemption will take place. However, with the lack of documentation of these primitives, future support and backwards compatibility of this solution is not guaranteed [21].

5.2 RTAI kernel modifications

Concluding the previous section, existing scheduler primitives in RTAI can be used to implement FPNS with varying levels of overhead and tradeoffs in practicality

and interference with other scheduling protocols. As an alternative, the notion of a non-preemptible task can be moved into the RTAI kernel proper, allowing for modified scheduling behaviour according to FPNS, without introducing extra overhead during the running of a task as induced by the API primitives mentioned. Looking ahead to our goal of implementing FPDS, this also allows more fine grained modifications to the scheduler itself, such that optional preemption points become possible in an efficient manner: rather than trying to disable the scheduler during an FPNS job, or influencing its decisions by modifying essential task parameters such as priorities, the scheduler would become aware of non-preemptible or deferred preemptible tasks and support such a schedule with intelligent decisions and primitives. It does however come at the cost of implementation and maintenance complexity. Without availability of documentation of the design and implementation of the RTAI scheduler, creating these extensions is more difficult and time consuming than using the well documented API. And because the RTAI scheduler design and implementation is not stable, as opposed to the API, continuous effort will need to be spent on maintaining these extensions with updated RTAI source code, unless these extensions can be integrated into the RTAI distribution. Therefore we aim for a patch with a small number of changes to few places in the existing source code.

An important observation is that with respect to FPPS, scheduling decisions are only made differently during the execution of a non-preemptive task. Preemption of any task must be initiated by one of the scheduling functions, which means that one possible implementation of FPNS would be to alter the decisions made by the scheduler if and only if a FPNS task is currently executing. This implies that our modifications will be *conservative* if they change scheduling behaviour during the execution of non-preemptive tasks only.

API changes for non-preemptible tasks

We extended the API with a new primitive named `rt_set_preemptible()` that accepts a boolean parameter indicating whether the calling task should be preemptible, or not. This value will then be saved inside the task's control block (TCB) where it can be referenced by the scheduler when making scheduling decisions. This *preemptible* flag inside the TCB only needs to be set once, e.g. during the creation of the task and not at every job execution, such that there is no additional overhead introduced by this solution.

Deferral of scheduling during non-preemptible task execution

During the execution of a (FPNS) task, interference from other, higher priority tasks is only possible if the scheduler is invoked through one of the following ways:

- From within the timer ISR
- From, or as a result of a system call by the current task

The first case is mainly used for the release of jobs - after the timer expires, either periodically or one-shot, the scheduler should check whether any (periodic) tasks should be set *ready*, and then select the highest priority one for execution. The second case applies when a task does a system call which alters the state of the system in such a way that the schedule *may* be affected, and thus the scheduler should be called to determine this. With pure FPNS and unlike FPDS, all scheduling work can be deferred until the currently running task finishes execution. Lacking any preemption points, under no circumstances should the current FPNS task be preempted. Therefore, the condition of a currently running FPNS task should be

Task	Prio	T	Φ	C^τ
τ_1	1	100	0	0
τ_2	2	1000	0	500

Table 5.1: Task set used for testing FPNS scheduling behaviour

detected as early on in the scheduler as possible, such that the remaining scheduling work can be deferred until later, and the task can resume execution as soon as possible, keeping the overhead of the execution interruption small.

Execution of the scheduler should be skipped at the beginning, if the following conditions hold for the currently running task:

- The *preemptible* boolean variable is unset, indicating that this is a non-preemptive task
- The *delayed* task state flag not set, indicating that the job has not finished
- The *ready* task state flag is set, indicating that the job is ready to execute

We have added tests for this condition to the beginning of both scheduler functions `rt_schedule()` and `rt_timer_handler()`, which resulted in the desired FPNS scheduling for non-preemptible tasks. For preemptive tasks, which have the default value of *true* in the *preemptible* variable of the TCB, the scheduling behaviour is not modified, such that the existing FPPS functionality remains unaffected.

5.3 Verification

For testing the results of our implementation in the previous section, we developed a testing scheme that verifies whether the interleaving of tasks scheduled by our implementation indeed follows a FPNS schedule.

We defined the task set described in Table 5.3. A low priority task τ_2 with a low period T_2 and relatively high computation time C_2^τ is set up to receive interference from a high priority task τ_1, which has a ten times higher period T_1 and essentially zero computation time C_1^τ. The sole purpose of the higher priority task is to interrupt the jobs of the lower priority task in order to verify whether the lower priority task will be preempted.

Preemption of tasks is normally transparent to the tasks themselves. However, RTAI has support for signaling tasks when they are about to be switched in. Tasks can install a *signal handler*, which is invoked from the context of the task, but with interrupts disabled. For each task we created signal handlers which append a combination of their task identifier (the task's priority) and the job number to a globally allocated integer array on every context switch:

$$IL(\iota_{i,j}) = i \times 10000 + j \tag{5.1}$$

Using the contents of this array, we can determine the interleaving of jobs over context switches during the test run. In a FPPS schedule, a single low priority job $\iota_{2,j}$ should be preempted by high priority jobs $\iota_{1,j}, \iota_{1,j+1}, \ldots$ multiple times, and therefore be appended to the integer array multiple times, surrounding the preempting jobs of task τ_1. In contrast, in a FPNS scheduled system a job $\iota_{1,j}$ should appear exactly once, as a running job should never be preempted.

As a baseline test, we first ran the test under both the original, unmodified kernel, and under the modified kernel but with non-preemptiveness *disabled*. Both tests should result in a FPPS schedule, and indeed we found identical results for both tests:

```
Interleaving was as follows:
0 0 0 0 0 0 0 0 10001 20000 10002 20000 10003 20000 10004 20000 10005
20000 10006 20000 10007 20000 10008 10009 10010 20001 10011 20001
10012 20001 10013 20001 10014 20001 10015 20001 10016 20001 10017
20001 10018 10019 10020 20002 10021 20002 10022 20002 10023 20002
10024 20002 10025 20002 10026 20002 10027 20002 10028 10029 10030
20003 10031 20003 10032 20003 10033 20003 10034 20003 10035 20003
10036 20003 10037 20003 10038 10039 10040 20004 ...
```

This interleaving follows our expectations: the low priority task τ_2 is preempted several times by high priority jobs within the same low priority job execution.

With preemption disabled for the low priority task in our FPNS system, after our modifications, we should not be seeing the same low priority job being context switched in more than once:

```
Interleaving was as follows:
0 0 0 0 0 0 0 0 10003 10006 10007 10008 10009 10010 10011 10012 20001
10013 10015 10016 10017 10018 10019 10020 10021 10022 20002 10023
10025 10026 10027 10028 10029 10030 10031 10032 20003 10033 10036
10037 10038 10039 10040 10041 10042 20004 10043 10046 10047 10048
10049 10050 10051 10052 20005 10053 10055 10056 10057 10058 10059
10060 10061 10062 20006 10063 10065 10066 10067 ...
```

In this result we find that indeed the low priority task is not preempted and each job is running to completion - each low priority job appears exactly once. High priority jobs that are arriving during the execution of the non-preemptible low priority task are queued up and are scheduled immediately after the low priority job finishes. This implies that the high priority jobs miss their deadlines due to their relatively high period, but this does not matter for the purpose of this test.

In the output, some instances of the high priority task are missing, e.g. 10004 and 10005, i.e. jobs $\iota_{1,4}$ and $\iota_{1,5}$. The explanation for this is that jobs $\iota_{1,3}$ and $\iota_{1,4}$ missed their deadline, which caused function `rt_task_wait_period()` to return immediately, without resulting in any context switch.

The context switches to task 0 at the beginning are the result of the (lowest priority) *initializing task* being context switched in before the test starts; it is not part of the test itself. The *first* job of every task is also missing in the interleaving, because according to the RTAI API documentation, the first context switch into a task is not signaled by RTAI. The reason for this is unknown.

Chapter 6

FPDS

Following the description of FPNS in the previous section, we move forward to the realisation of a FPDS scheduler, by building upon this design and implementation. An FPDS implementation as outlined in section 4.2.1, where subtasks are modeled as non-preemptive tasks with preemption points in between, has the following important differences compared to FPNS:

- A job is not entirely non-preemptive anymore; it may be preempted at predefined preemption points, for which a scheduler primitive needs to exist.
- The scheduling of newly arrived jobs can no longer be postponed until the end of a non-preemptive job execution, as during a (optional) preemption point information is required about the availability of higher priority ready jobs.

Figure 6 shows a time diagram of the FPDS design that will be described in this chapter. Low priority FPDS task τ_2 is executing when at time $t = 2$ a higher priority job of task τ_1 arrives. This job is "woken up" by the scheduler executing inside the periodic timer Interrupt Service Routine (ISR). The scheduler detects that a lower priority but non-preemptive FPDS subjob is executing, and is not at a preemption point. It will notify the lower priority task that a higher priority task is waiting to preempt it. τ_2 resumes execution after the ISR finishes, until it reaches its next preemption point at time $t = 7$. There, it receives the notification by the scheduler of a higher priority waiting task, and decides to *yield* to the higher priority task, by executing a corresponding API primitive. The kernel invokes the scheduler again, but detects that the executing task is at a preemption point, and this time allows preemption by the higher priority task τ_1. After this job finishes, τ_2 can start with its next subjob at time $t = 11$.

6.1 Scheduler modifications

There are several ways to approach handling interrupts which occur during the execution of nonpreemptive subjob. First, the interrupt may be recorded with all scheduling postponed until the scheduler invocation from `yield()` at the next preemption point.

Alternatively, all tasks which have arrived can be moved from the pending queue to the ready queue directly (as is the case under FPPS), with only the context switch postponed until the next preemption point. This has the advantage that there is opportunity for optional preemption points to be implemented, if the information about the availability of a higher priority, waiting task can be communicated in an efficient manner to the currently running FPDS task for use at the next optional preemption point. These two alternatives can be described as *pull* versus *push*

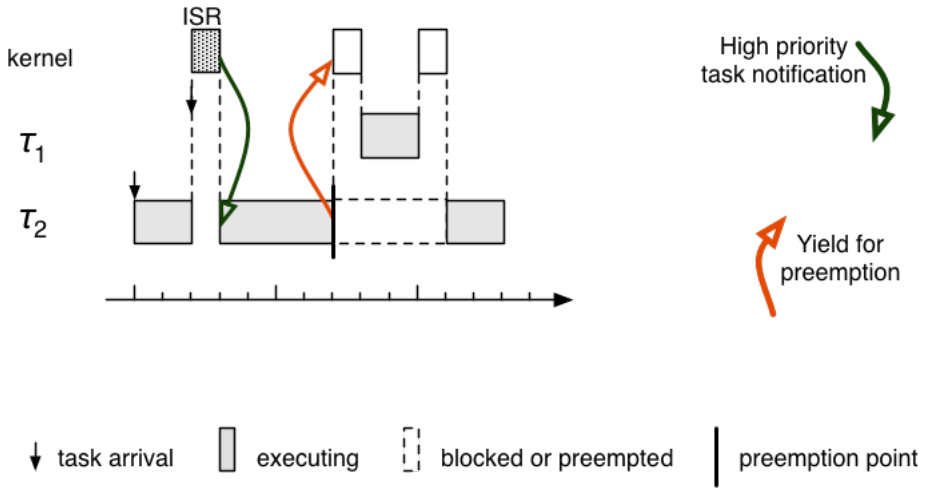

Figure 6.1: Time schedule of preemption in our FPDS design

models respectively. They represent a tradeoff, and the most efficient model will most likely depend on both the task sets used, and the period of the timer.

On our platform we could not measure the difference between these two alternatives; any difference in efficiency between the two approaches was lost in the noise of our experiment. Therefore we opted to go with the last alternative, as this would not require extensive rewriting of the existing scheduler logic in RTAI, and thereby fit our requirements of *maintainability* and our extensions being *conservative*. The efficiency differences between these approaches may however be relevant on other platforms, as described in [16], based on [10].

Working from the FPNS implementation of section 5, the needed modifications to the scheduling functions `rt_schedule()` and `rt_timer_handler()` for FPDS behaviour are small. Unlike the FPNS case, the execution of the scheduler can not be deferred until the completion of a FPDS (sub)task if we want to use the push mechanisms just described, as the scheduler needs to finish its run to acquire this information. Instead of avoiding the execution of the scheduler completely, for FPDS we *defer* the invocation of a context switch to a new task if the currently running task is not at a preemption point, and notify the running task of a waiting higher priority job.

The existing test clause for preemptibility of the currently task that we introduced at the start of the scheduler functions for FPNS is therefore moved to a section in the scheduler code between parts 3 and 4 as described in Section 3.5, see Figure 6.1. There, the scheduler has decided that a new task should be running, but has not started the context switch yet. At this point we set a newly introduced TCB integer variable `should_yield` to *true*, indicating that the current task should allow itself to be preempted at the next preemption point. This variable is reset to *false* whenever the task is next context switched back in.

With these modifications, during a timer interrupt or explicit scheduler invocation amidst a running FPDS task, the scheduler will be executed and wake up any timed tasks. If a higher priority task is waiting at the head of the ready queue, a corresponding notification will be delivered and the scheduler function will exit without performing a context switch.

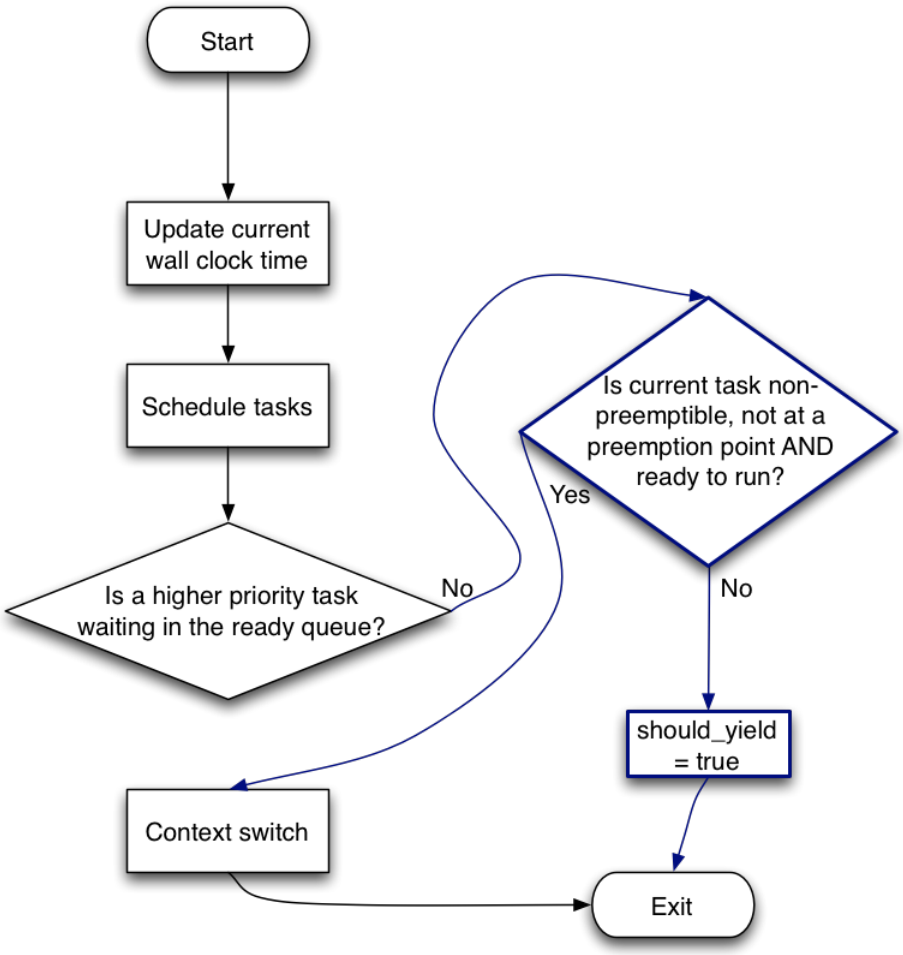

Figure 6.2: Flow diagram of the FPDS scheduler logic

This is still not sufficient for achieving FPDS behaviour however, as the scheduler will not allow preemptions even during a preemption point. To achieve correct preemption behaviour in this situation, we extended the `if` clause with a test for the `RT_FPDS_YIELDING` task state flag that was introduced. During the execution of the scheduler at a preemption point it is set to *true*, causing the scheduler to execute as in a FPPS system, and perform a context switch to the new, higher priority task as desired.

These modifications to the scheduler are described in the form of a flow diagram in Figure 6.1. The blue figures and arrows represent changes compared to the original scheduler implementation, which was described by Figure 3.5.

Kernel implementation of preemption points

It would appear that using a standard `yield()` type function, as present within RTAI and many other operating systems, would suffice for implementing a preemption point. Upon investigation of RTAI's yield function (`rt_task_yield()`) it turned out however that it could not be used for this purpose unmodified. This

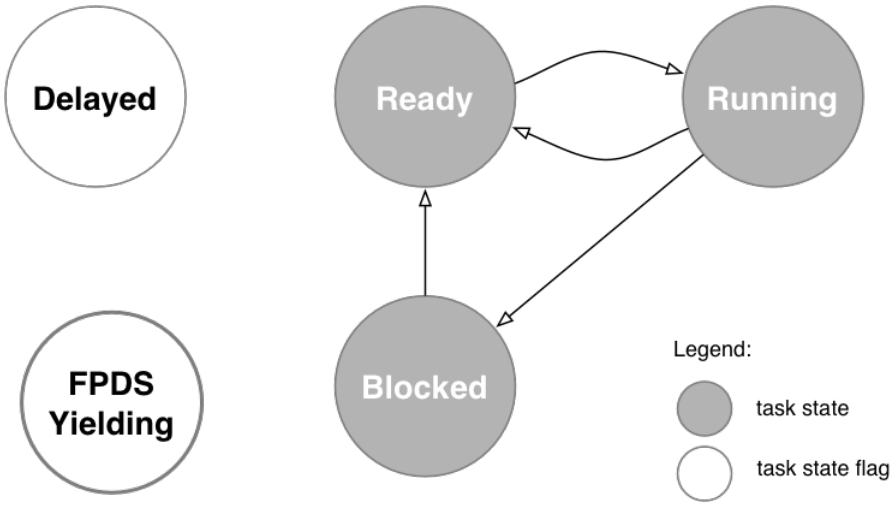

Figure 6.3: RTAI tasks state extended with RT_FPDS_YIELDING bit flag

function is only intended for use with round-robin scheduling between tasks having equal priority, because under the default FPPS scheduler of RTAI there is no reason why a higher priority, ready task would not already have preempted the current, lower priority task. However with the non-preemptive tasks in FPDS, a higher priority job may have arrived but not been context switched in, so checking the ready queue for *equal* priority processes is not sufficient. An unconditional call to scheduler function `rt_schedule()` should have the desired effect, as it can move newly arrived tasks to the ready queue, and invoke a preemption if necessary. However, the modified scheduler will evaluate the currently running task as non-preemptive, and avoid a context switch. To indicate that the scheduler is being called from a preemption point and a higher priority task is allowed to preempt, we introduce a new bit flag `RT_FPDS_YIELDING` to the task state variable in the TCB, that is set before the invocation of the scheduler to inform it about this condition. The flag is then reset again after the scheduler execution finishes.

Due to the different aim of our FPDS yield function in comparison to the original yield function in RTAI we decided not to modify the existing function, but create a new one specific for FPDS use instead: `rt_fpds_yield()`. The FPDS yield function is simpler and more efficient than the regular yield function, consisting of just an invocation of the scheduler function wrapped between the modification of task state flags. This also removed the need to modify the existing code which could introduce unexpected regressions with existing programs, and have a bigger dependency on the existing code base, implying greater overhead in maintaining these modifications in the future.

The following abstract code describes the implementation of the newly intro-

duced `rt_fpds_yield()` API primitive:

```
function task_thread() {
    # Indicate to the scheduler that we are executing a preemption point
    current_task.state.FPDS_YIELDING = true
    # Invoke the scheduler
    rt_schedule()
    # Reset the yielding flag
    current_task.state.FPDS_YIELDING = false
}
```

6.2 Notification of waiting higher priority tasks

With FPDS, part of the scheduling decisions is moved from the central scheduler at system level to the user application level. Although the system scheduler remains responsible for the administration of task arrival and scheduling of the order of tasks, the decision of *when* to activate a task now lies partially within an application, in the case of a required preemption. In order to be able to make this decision in an efficient way, the currently running FPDS application should either be informed by the scheduler when a higher priority job has arrived and is waiting for preemption, or be able to use an efficient primitive to ask the kernel to make this decision upon invocation. These solutions follow push and pull models, respectively.

The latter alternative, pulling (or polling), is effectively already implemented by the `rt_fdps_yield()` function introduced in the previous section. By invocation of this primitive the scheduler is called with the `RT_FPDS_YIELDING` task state flag set, allowing the scheduler to wake up timed tasks and move them to the ready queue, and then context switch to a higher priority task if one is available (see Section 3.5). In case there is no higher priority task waiting, the scheduler will return without action to the calling task. This is indeed our desired FPDS behaviour, but it comes at a cost of a system call including a full scheduler invocation at every preemption point.

Because we want tasks to be able to learn about the presence of waiting higher priority tasks without necessarily causing a preemption, we implemented a simple method of reading the status of the *should_yield* flag that is stored in the TCB and written to by the timer interrupt handler. Because the TCB is stored in kernel-memory, inaccessible by normal user-land tasks, we introduced a new system call `rt_should_yield()`, which returns a boolean with the value of the TCB *should_yield* variable. This represents a first step towards support for *optional preemption points*, although this implementation is not the most efficient one: for every inspection of the presence of waiting tasks, a system call is necessary. A more efficient implementation that avoids this problem will be introduced in the next chapter.

6.3 User-land support

To aid the use of FPDS by real-time programs, a small FPDS specific library has been written that abstracts the use of the API primitives offered by RTAI, and can make commonly used functions available that are helpful to FPDS applications. Initially, this library merely consists of functions for FPDS task initialization and an implementation of optional preemption points. The expectation is that it will be extended when FPDS is employed along with support for task deadline monitoring and/or enforcement, or in the context of *mode changes* [17], where a task will have a non-sequential subtask structure to support different modes.

The new library was named `libfpds` and was written in C as a shared library, with a dependency on the RTAI `liblxrt` library for access to the RTAI API.

6.3.1 FPDS task initialization

A task wishing to be scheduled using FPDS can make use of the `fpds_task_init()` function. It expects one argument: a pointer to a *struct* of type `fpds_context`, which is used for keeping state information local to the thread/task. Since the library must be able to support multiple threads/tasks within a single program to which it is linked, it cannot keep its state in its own static memory on the heap, as it would be shared by all threads using it. We decided to solve this problem by having each task manage its own FPDS context instance, by allocating and deallocating it itself, and passing it onto `libfpds` functions that require access to it. Function `fpds_task_init()` takes care of initialisation of the FPDS context instance, which is opaque to the task itself.

The initialising task is also set to be non-preemptible, using the API primitive `rt_set_preemptible()` that was introduced in Section 5.2. From thereon, the task will either need to move out of *ready* state (e.g. using the primitive `rt_wait_period`), or execute a preemption point to give control to other tasks.

6.3.2 Preemption point

In Section 6.1 all necessary kernel infrastructure to support preemption points has been established, but what remains is a preemption point *primitive* that is simple to use by programs utilising FPDS. A preemption point should not need to consist of more than one line of code at most, e.g. a function call, for it not to be too intrusive to existing task code. We introduce a new function `fpds_pp()` for this purpose.

A preemption point serves two purposes:

- Detection whether there are any waiting higher priority tasks
- Yielding to higher priority tasks (to allow preemption)

Likely these two purposes will be combined in most cases, i.e. a yield to higher priority tasks will be done if higher priority waiting tasks are detected, but this may not always be the case. It can be useful for tasks to learn about the presence of higher priority tasks without immediately transferring control to them. For this reason we introduce a boolean argument `allow_yield` to function `fpds_pp()` that indicates the wanted behaviour, i.e. whether the preemption point is allowed to immediately *yield* before returning if suggested by the kernel, or not. Furthermore, the function will return a boolean that represents the presence of waiting higher priority tasks, i.e. the value of the `rt_should_yield()` system call. The task can use this return value, if interested, to determine if a preemption had occurred during the preemption point (if `allow_yield` is *true*), or if preemption will occur next time it will yield (if `allow_yield` is *false*).

A corresponding flow diagram of the user-land implementation of a preemption point is shown in Figure 6.3.2.

6.4 Verification

We would like to verify the correctness of our FPDS implementation in a similar way as we verified FPNS scheduling in Section 5.3. Our earlier method of recording the *interleaving* of jobs throughout context switches is no longer sufficient however,

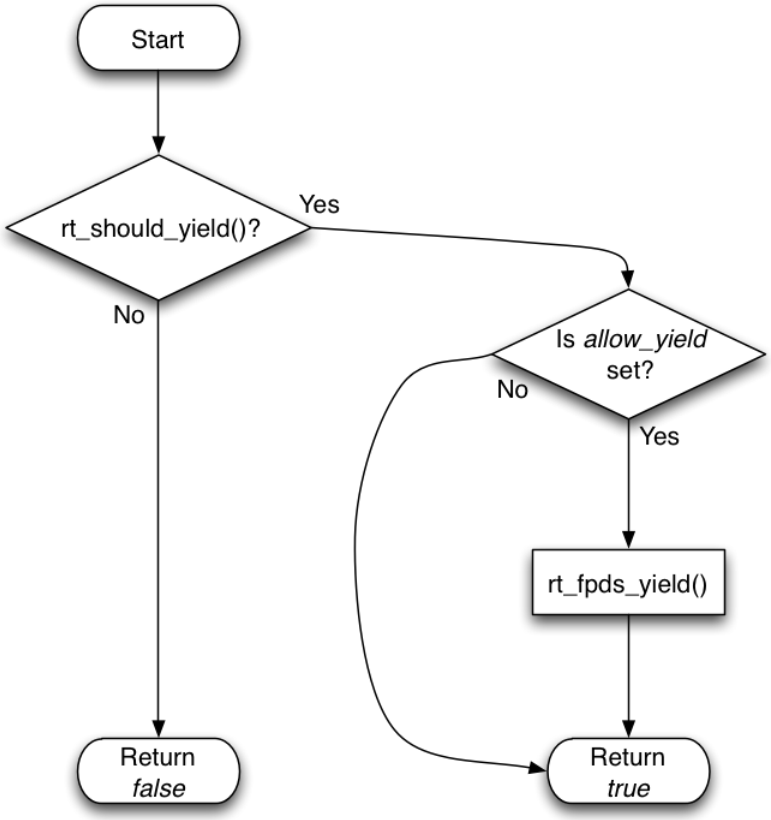

Figure 6.4: Flow diagram of a preemption point

as unlike an FPNS job, a FPDS job can be preempted. The resulting interleaving would therefore be difficult or impossible to distinguish from an FPPS schedule.

The difference between FPPS and FPDS is that under FPPS preemptions are allowed at arbitrary moments, whereas a FPDS job should only be preempted while it is executing a preemption point. In our testing, we attempt to find occurrences of preemptions where this is violated.

The test methodology of Section 5.3 was modified for use of FPDS scheduling. The low priority task τ_2 was changed into an FPDS task, by splitting it into $K = 500$ equally sized subtasks, $\tau_{2,0}, \tau_{2,1}, \ldots, \tau_{2,K-1}$ with $C^{\tau}_{2,k} = C^{\tau}_2/K$. Between every pair of subtasks a preemption point was placed.

Because the high priority task τ_1 will be attempting to preempt the low priority FPDS task, we modified it to test whether τ_2 was executing a preemption point while τ_1 is assigned the CPU. For the purpose of testing, we introduced a global integer variable `at_pp`, which is maintained at 1 when τ_2 is allowed to be preempted (i.e., at a preemption point, and before the first and after the last instruction of the task), and at 0 everywhere else. Task τ_1 was changed to assert whether `at_pp` equals 1 during its execution, displaying an error message and terminating the test program if this is not the case.

```
Interleaving was as follows:
0 0 0 0 0 0 0 0 10000 20000 10001 20000 10002 20000 10003 20000 10004
20000 10005 20000 10006 10007 10008 10009 10010 20001 10011 20001
```

```
10012 20001 10013 20001 10014 20001 10015 20001 10016 10017 10018
10019 10020 20002 10021 20002 10022 20002 10023 20002 10024 20002
10025 20002 10026 10027 10028 10029 10030 20003 10031 20003 10032
20003 10033 20003 10034 20003 10035 20003 10036 10037 10038 10039
10040 20004 10041 20004 10042 20004 10043 20004 ...
```

In this test, clearly the low priority task τ_2 was preempted by higher priority task τ_1, and no jobs missed their deadlines. The high priority task did not detect any preemptions of τ_2 outside a preemption point, i.e. `at_pp` was 1 consistently throughout the test. This result thus provides a strong indication that this task set was scheduled correctly using a FPDS scheduling scheme.

6.5 Key performance indicators

Our implementation should be checked for the following elements, which relate to the design aspects mentioned in Section 4.3:

- The interrupt latency for FPDS. This is intrinsic to FPDS, i.e. there is additional blocking due to lower priority tasks. It has been dealt with in the analysis in [6, 7];
- The additional *run-time* overhead due to additional code to be executed. This will be measured in Section 6.6;
- The additional *space* requirements due to additional data structures and flags. Our current implementation introduces only two integer variables to the TCB, so the space overhead is minimal;
- The number of *added*, *changed*, and *deleted* lines of code (excluding comments) compared to the original RTAI version. Our extension adds only 106 lines and modifies 3 lines of code, with no lines being removed;
- The *compatibility* of our implementation. Because our extensions are conservative, i.e. they don't change any behaviour when there are no non-preemptive tasks present, compatibility is preserved. This will also be verified by our measurements in Section 6.6.1.

6.6 Measurements

We performed a number of experiments to measure the additional overhead of our extensions compared to the existing FPPS scheduler implementation. The hardware used for these tests was an Intel Pentium 4 PC, with 3 Ghz CPU, running Linux 2.6.24 with (modified) RTAI 3.6-cv.

To avoid factoring in the overhead of the actual *measurement process* in the results, we have taken the commonly used approach of measuring a varying amount of iterations of the subject over a timed interval, and averaging the results.

6.6.1 Scheduling overhead

The goal of our first experiment is to measure the overhead of our implementation extensions for existing real-time task sets, which are scheduled by the standard RTAI scheduler, i.e. following FPPS. For non-FPDS task sets, scheduling behaviour has not been changed by our conservative implementation, but our modifications may have introduced additional execution overhead.

As our example task set we created a program with one non-periodic, low priority, long running FPPS task τ_l, and one high priority periodic FPPS task τ_h. τ_l consists of a `for` loop with a parameterized number of iterations m, to emulate a task with computation time C_l^τ. The computation time of the high priority task, C_h^τ, was 0; the only purpose of this empty task is to allow for measurement of overhead of scheduling by presenting an alternative, higher priority task to the scheduler. T_h was kept equal to the period of the timer, such that a new high priority job is released at every scheduler invocation from the timer event handler. The following pseudocode represents the definition of both tasks:

```
function high_job(instance) {
    loop_task(0)
}

function low_job(instance) {
    loop_task(50000000)
}

function loop_task(n) {
    for (i = 0; i < n; i++)
       # noop
}
```

Since, from the perspective of an FPPS task, the only modified code that is executed is in the scheduler functions, we measured the response time of task τ_l under both the original and the modified FPDS real-time RTAI kernel. We varied the period of the timer interrupt, and thereby the frequency of scheduler invocations encapsulated by the timer event handler. The results are shown in Figure 6.5.

As expected, there is no visible difference in the overhead of the scheduler in the modified code compared to the original, unmodified RTAI kernel. For an FPPS task set the added overhead is restricted to a single `if` statement in the scheduler, which references 3 variables and evaluates to *false*. This overhead is unsubstantial and lost in the noise of our measurements. We conclude that there is no significant overhead for FPPS task sets introduced by our FPDS extensions.

6.6.2 Preemption point overhead

With an important aspect of FPDS being the placement of preemption points in task code between subtasks, the overhead introduced by these preemption points is potentially significant. Depending on the frequency of preemption points, this could add a substantial amount of additional computation time to the FPDS task.

We measured the overhead of preemption points by creating a long-running, non-periodic task τ_l with fixed computation time C_l^τ implemented by a `for` loop with $m = 100M$ iterations, and scheduled it under both FPPS and FPDS. The division into subtasks of task τ_l has been implemented by invoking a preemption point every n iterations, which is varied during the course of this experiment, resulting in $\lceil m/n \rceil$ preemption point invocations.

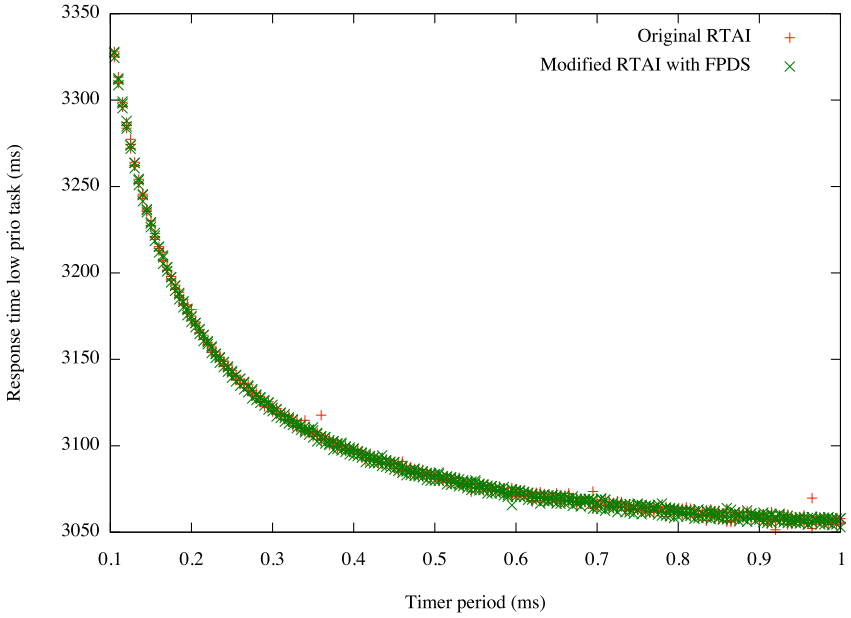

Figure 6.5: Overhead of the modified kernel for FPPS task sets

The following abstract code describes the definition of τ_l:

```
function low_job(instance, pp_sample) {
    for (i = 0; i < 1000000000; i++) {
        if (mod(i, pp_sample) == 0) {
            yield_count += fpds_pp(1)    # Preemption point
        }
    }
}
```

For the FPPS test the same task was used, except that every n iteration interval only a counter variable was increased, instead of the invocation of a preemption point. This was done to emulate the same low priority task as closely as possible in the context of FPPS:

```
function low_job(instance, pp_sample) {
    for (i = 0; i < 1000000000; i++) {
        if (mod(i, pp_sample) == 0) {
            yield_count += 1
        }
    }
}
```

The response time R_l was measured under varying intervals of n for both FPPS and FPDS task sets. The results are plotted in Figure 6.6.

Clearly the preemption points introduced in the lower priority task introduce overhead which does not exist in a FPPS system. The extra overhead amounts to about 440 μs per preemption point invocation, which corresponds well with a measured value of 434 μs per general RTAI system call overhead which we obtained in separate testing. This suggests that the overhead of a preemption point

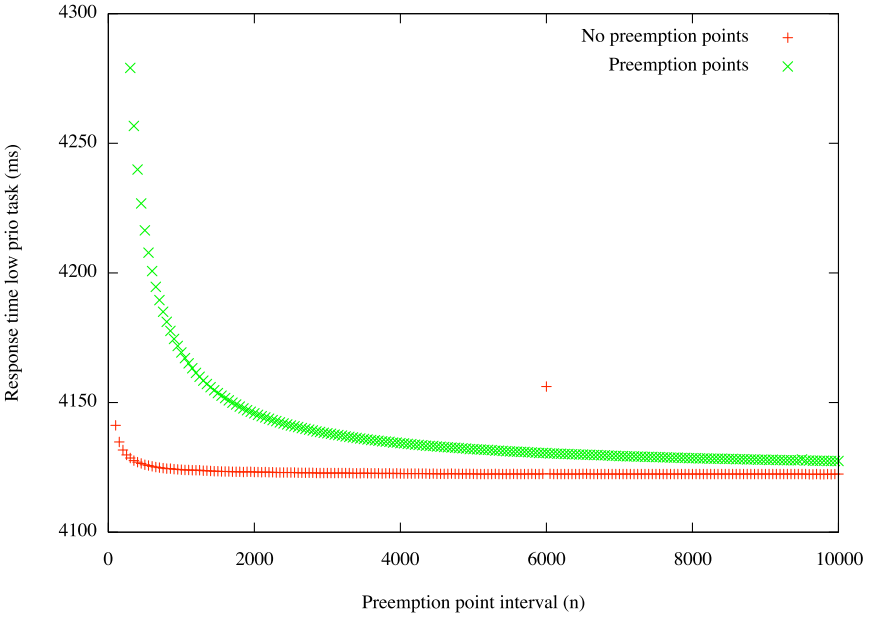

Figure 6.6: Overhead of preemption points

is primary induced by the `rt_should_yield()` system call in the preemption point implementation, which is invoked unconditionally.

We see one anomaly in the test, around $n = 6000$ in the graph, where the response time of FPPS is significantly higher than expected. The exact reason for this unfortunately remains unclear.

6.6.3 An example FPDS task set

Whereas the previous experiments focussed on measuring the overhead of the individual extensions and primitives added for our FPDS implementation, we performed an experiment to compare the worst case response time of a task set under FPPS and FPDS as well. The task set of the previous experiment was extended with a high priority task τ_h with a non-zero computation time C_h^τ (see the pseudocode below). For this experiment we varied the period of the high priority task. To keep the workload of the low priority task constant, we fixed the interval n of preemption points to a value (5000) frequent enough to allow preemption by τ_h without it missing any deadlines under all values of $T_h \geq 1.2ms$ under test. The response

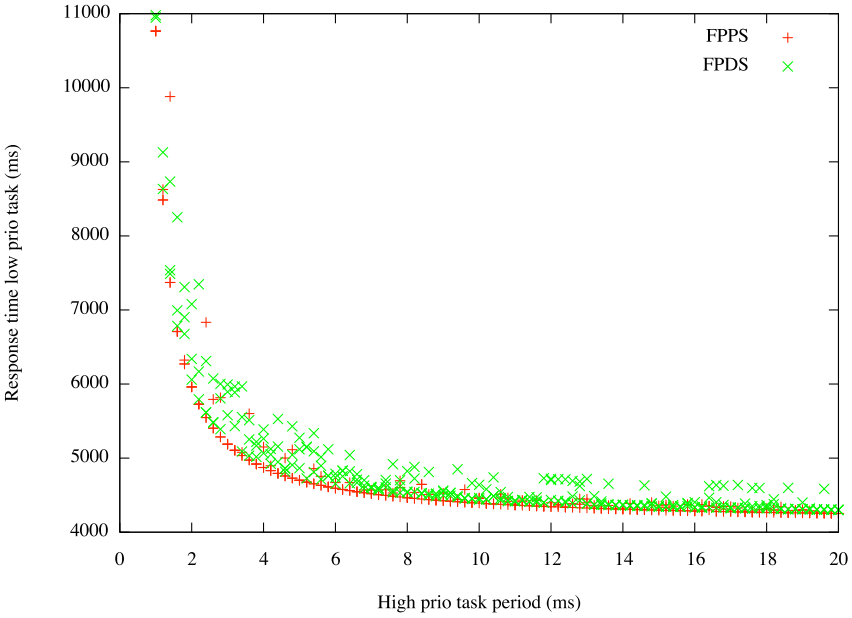

Figure 6.7: A task set scheduled by FPPS and FPDS

time of the low priority task (R_l) is plotted in Figure 6.7.

```
function high_job(instance) {
    loop_task(100000)    # ca. 1.2 ms on the test machine
}

function low_job(instance) {
    for (i = 0; i < 1000000000; i++) {
        if (mod(i, 5000) == 0) {
            yield_count += fpds_pp(1)    # Preemption point
        }
    }
}
```

The relative overhead appears to depend on the frequency of high priority task releases and the resulting preemptions in preemption points, as the number of preemption points invoked in the duration of the test is constant. The results show an increase in the response time of τ_l for FPDS of at most 17% with a mean around 3%. The large discrepancy of the results can probably be attributed to the unpredictable interference from interrupts and the resulting invalidation of caches. Considering the relatively low mean overhead of FPDS, we would like to identify the factors which contribute to the high variation of the task response time, and investigate how these factors can be eliminated (see Chapter 9).

Chapter 7

Optional preemption points

The implementation of FPDS that we arrived on in the previous chapter was functional, but not really optimized for efficiency. Especially the implementation of preemption points is wasting resources by unconditionally executing a system call which, for the real-time Linux tasks in user mode, requires two privilege mode changes (from user into kernel mode, and back). We may be able to avoid this. When preemption points cause significant slowdowns of the execution of tasks, and decrease the utilisation of the system, this may cause programmers to reduce their usage and make the division of FPDS tasks into subtasks less fine grained, which again reduces the schedulability of task sets and the utilisation of a system. It is therefore in our interest to make sure that preemption points are implemented in an efficient and nonintrusive way.

In this chapter we will investigate the different alternatives to implement optional preemption points, and show how our newly chosen alternative implementation performs better than the one of the previous chapter.

7.1 Push notification of waiting higher priority tasks

If the presence of waiting higher priority tasks, as detected from the interrupting timer handler, can be pushed down to the running task, then the largest source of preemption point overhead can be removed. Timer events are executed even during the execution of an FPNS/FPDS task, because (sub)tasks are non-preemptive only with respect to other tasks, not interruption by interrupt handlers.

For the implementation of notification from the scheduler inside the timer interrupt handler, we recognized the following options:

- RTAI's signal handling system
- A settable flag in the TCB, which can be polled during a preemption point using a system call
- A settable flag which can be examined by the task from user space without a system call, utilising a memory page shared by the currently running process and the kernel

The solution using RTAI's signal handling system was quickly rejected upon examination; although tasks' own signal handlers are executed within the process space of the installing task, they run in a different execution context which implies that two context switches are required for the delivery of a new task's arrival. Because one of the stated advantages of FPDS is reduction of the overhead of context

switches [11,16], this solution with two extra context switches per task arrival event appears counter-productive in reducing the overhead compared to FPPS.

The second option is effectively the one that was implemented in the previous chapter. It involves the modification of a boolean *should_yield* that is added to the TCB. When during the execution of the scheduler it is determined that the currently running FPDS task should (but cannot) be preempted, this variable in the task's TCB will be set to *true*. When the task later reaches a preemption point, it can read this value through a system call without having to invoke the scheduler, and decide whether to perform a yield based on this information. The scheduler makes sure the flag is reset upon a context switch to the current task. Although this approach is quite elegant in terms of simplicity, it is not very efficient due to the implied system call overhead.

Alternatively this *should_yield* flag could be placed in special reserved memory page that is mapped inside FPDS tasks' process space as a read-only page, but writeable by the kernel. This recent approach, with the common name *virtual system call (vsyscall)* is now used by several mainstream operating systems to allow for information transfer from kernel to user space without incurring system call overhead. A typical example is the vsyscall version of the often called Unix system call `gettimeofday()`. Every timer interrupt the kernel updates the current time in this read-only memory page that is mapped to every process' address space, where it can directly be read without a system call by a userspace function also placed in this memory page. This same approach could be used for the *should_yield* variable, which would mean that the need for a system call at every preemption point is removed. This method is favourable in terms of run-time overhead compared to the other two methods; however it is also the method with the highest implementation complexity, as well as the greatest dependency on the existing code base. This is due to the procedures required to set up a shared, read-only memory page mapped into every process with dynamic shared objects (library code) inside.

While investigating this option we found that recent Linux kernels have some infrastructure to support this scheme, however these implementations were architecture specific, and not universally supported. We found several different implementations between both architectures and recent kernel versions, suggesting that a more final and stable cross platform implementation is still being worked on. Moreover, this infrastructure is handled within the Linux kernel proper, and not by RTAI. Using this infrastructure from RTAI would increase the cross dependencies between the Linux source code base and the RTAI patches, further complicating the maintainability of our extensions. For these reasons we decided that this method was not very attractive.

7.1.1 Alternative design

There are two specific properties of our real-time environment which we can make use of to arrive at a simpler implementation that has the same performance benefits as the one described above, but without the maintainability and complexity drawbacks. Instead of setting up a special memory page to be written to by the kernel, and mapped into the address space of interested processes, the regular, allocated memory of a process could be used as well. The kernel, which the RTAI hypervisor is part of, has full (write) access to all memory, and would therefore be able to update a variable in user process memory space as well. In general this is however not unproblematic:

- A different process' memory address space may be loaded when the kernel needs to update a process' variable.

- The relevant memory page may be *paged out* into swap space, and therefore not present in physical memory.
- Care needs to be taken that no access control measures are circumvented.

In our case, the first does not apply. Whenever the kernel wants to update the `should_yield` variable to *true*, indicating that a higher task is waiting to preempt a lower priority FPDS task, this is during the (interruption of the) actual execution of the task of which the `should_yield` variable should be set. This implies that the memory address space of this process is guaranteed to be loaded at that point. The setting of the `should_yield` flag is done either by the execution of the scheduler from the timer interrupt handler, or from a system call that (indirectly) invokes `rt_schedule()`. Neither of these cases is problematic.

The kernel also needs to reset the flag again after a context switch has occurred, before the task resumes execution again and a new high priority task can introduce interference. The location of this assignment can conveniently be chosen at a moment when the process's address space is loaded. In our implementation we chose to reset the flag in the scheduler directly after the context switch back has completed, i.e. when the original task is context switched back in after preemption by one or more tasks, but before it resumes its actual execution.

The condition of an attempted write by the kernel to a process's memory page that is paged out (a *page fault*) is also not as likely to occur, since real-time tasks need to *lock* their memory pages into physical memory to maintain real-time behaviour. In RTAI/Linux, real-time programs are recommended to use the `mlockall()` system call to achieve this, in which case a page fault can no longer occur. Even if this is unwanted for some reason, it is sufficient to lock the memory page containing the `should_yield` variable into physical memory, before registering it with the RTAI kernel.

Finally, the use of the in-kernel macro `put_user()`, which checks whether the given address is within the user space boundary of the memory address space shared by the kernel and the process, takes care of the problem of access control. Because this macro enforces the registered address of the `should_yield` variable to be confined to the process's address space, there is no risk of violating either the memory of the kernel, or any other process.

Implementation

We implemented this solution by introducing two new RTAI system calls, `rt_assign_var()` and `rt_release_var()`. These allow an RTAI task to register certain predefined variables on a given process memory address for access by the kernel, until they are released again. `rt_assign_var()` expects three arguments: an identifier for the variable that will be registered with the kernel (currently, only `RT_VAR_SHOULD_YIELD` is supported), a pointer to the variable in the process's address space, and the size of the variable, in bytes. System call `rt_release_var()` only expects the variable identifier and will unregister it if it was previously set. Extra variables that need to be shared between the RTAI kernel and the task can be introduced in a similar way by using different variable identifier values, without having to introduce new RTAI system calls.

Comparing this solution to the *vsyscall* system described in Section 7.1, we find that it is simpler to implement, and has better maintainability properties because the interface with the Linux kernel is minimized. Our implementation has no need for a special memory page that is mapped into every (real-time) process, because already existing heap memory inside processes's address space can be used, such that the extra memory usage is restricted to a single integer pointer. But most importantly, in our implementation we can also support data transfer from user

space to kernel space, unlike the vsyscall solution in Linux, where processes can only read data previously stored by the kernel in the read-only mapped memory page. We will make use of this advantage in Chapter 8.

7.2 User-land support

The support library `libfpds` that was introduced in Section 6.3 has been extended to support the new style optional preemption points as well. One of both mechanisms can be selected at compile time by defining `FPDS_FAST_SHOULD_YIELD` preprocessor variable (which is the default). If defined, the new mechanism that is described here will be compiled in, otherwise the old style is selected.

The new optional preemption points make use of a `should_yield` variable in the address space of the real-time process, but we would like to abstract from this, and hide these details from the programmer. Since the `should_yield` flag is task/thread specific, it is added to the `fpds_context` struct, and initialized to *false* in function `fpds_task_init()`. To register this variable with the kernel, we introduce two thin wrappers around the RTAI API functions `rt_assign_var()` and `rt_release_var()` functions: `fpds_assign_should_yield()` and `fpds_release_should_yield()`. The former takes an integer pointer argument that will be passed to the kernel; the latter releases any previously registered variable.

Function `fpds_task_init()` is extended to include a call to `fpds_assign_should_yield()` as well, using the `should_yield` integer variable from the supplied `fpds_context` instance. Additionally, it performs a `mlock()` on the memory page containing the variable `should_yield`, for the reasons described in Section 7.1.1. Cleanup of this variable registration is automatic upon task destruction, but can be done manually by the task using the `fpds_release_should_yield()` function.

Because the `should_yield` variable needs to be inspected during a preemption point, the function `fpds_pp` now needs access to the `fpds_context` instance. The prototype of this function is therefore extended to require this instance to be passed as a `const` pointer. In `fpds_pp()`, the unconditional system call to `should_yield()` is now replaced by a very fast memory read of the `fpds_context->should_yield` variable, and a system call (to `rt_fpds_yield()`) is only required when the task indeed needs to yield for higher priority jobs.

7.3 Comparative measurements

Through some new measurements we compared the performance of the new implementation of optional preemption points to the previous, system call based implementation.

The performance improvement is expected to result from the replacement of system call based polling by a direct memory read. In Section 6.6.2, the overhead of executing preemption points over varying granularity was measured. We repeated this test to compare the response time of a task scheduled under FPPS, and the two FPDS implementations. The results are shown in Figure 7.1.

From the differences in response time shown in this graph we can conclude that our new FPDS preemption point implementation has a significant performance advantage compared to the previous FPDS implementation. The response times are similar to the FPPS scheduled task set, which does not have preemption points but executes a counter increment instead.

For the strange anomaly in the results, where the response time of the test program is significantly *better* than the mean we do not have a conclusive explanation; our best guess is that in the particular test instance the wall clock time was not

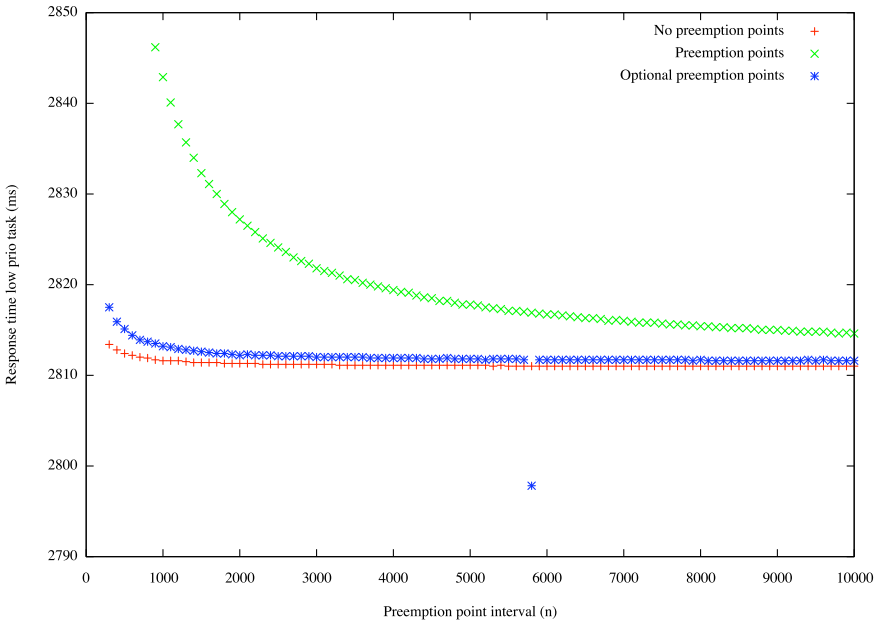

Figure 7.1: Comparison of preemption point implementation overhead

properly updated by the periodic timer handler due to e.g. a delayed interrupt, which affected the measurements.

Chapter 8

Future work

In the previous chapters, we have arrived at a functional and efficient implementation of FPDS. We will now have a look at how the structure of an FPDS task can be made explicit, and an example of how this can be exploited: we explore a possible design for support of *monitoring and enforcement* of task budgets.

8.1 Implementing subtask structure awareness

In our discussion about the task model in Section 4.2.3, we have seen that the structure of an FPDS task can be regarded as a directed acyclic graph of subtasks, and mentioned the possibility of making this structure apparent to the system. In this section we will discuss some of the potential applications of this idea.

In Section 4.2.3 we proposed a task model for FPDS that preserves the use of a single task in the system to model a FPDS task consisting of individual subtasks, but introduces a data structure describing the properties of, and relationship between these subtasks. In this section we will briefly describe some aspects that should be considered when designing and implementing this in RTAI.

It is expected that tasks initialize their structure of subtasks at task creation time. Each node in the DAG represents a subtask, which can be linked to one or more neighbors. When an FPDS task is initializing, it should supply the required information about the structure of the graph, and for each node any properties of subtasks that are considered useful. These subtask properties could include, for example:

- The worst-case computation time
- The best-case computation time
- The preemptibility, to mix preemptive and non-preemptive subtasks
- The *mode* to which the subtask belongs, in a system supporting "mode changes" (see Section 1.3.3)

Additionally, the nodes in the graph could be used to store the results of analysis on the task structure, such as the worst-case computation time of all paths in the subgraph for a given node. If this information does not change during the execution of jobs, it can reduce the overhead at preemption points and during scheduling by the kernel.

The DAG will be inspected throughout the execution of tasks, so it is recommended to select a graph data structure implementation that avoids fragmentation of individual nodes over multiple memory pages. A compact memory representation

would avoid memory cache misses and cache line trashing while other task data is being accessed between preemption points.

At least the following primitives should be created to operate on the task structure information:

- Primitives to initialize the properties of a single node (subtask)
- Primitives to form the graph structure of the task
- Primitives to iterate over the graph, e.g. during task execution at preemption points
- Primitives for cleanup of the data structure

8.2 Monitoring and enforcement

The introduction of non-preemptiveness into a real-world system poses a risk to the reliability of the entire system because non-preemptive tasks that get corrupted or are faulty can prevent even higher priority tasks from receiving sufficient processor time. A single task that slips into an endless loop can cause starvation of the entire system, for example. For this reason the *monitoring* of execution times of (sub)tasks, and the corresponding *enforcement* of the designed worst-case computation times is of increased importance in an FPDS scheduled system. Because a task is divided into non-preemptive subtasks of which we can keep individual properties and statistics, we have the opportunity to keep tighter control compared to a FPNS system where only information about entire tasks is available.

An FPDS task that violates its stated worst-case computation times causes interference to other tasks. With FPDS, higher priority tasks can preempt an FPDS task, only at a preemption point. Lower priority tasks need to wait until the complete FPDS task finishes.

Let τ_i be an FPDS task, consisting of K_i subtasks, $\tau_{i,0}, \tau_{i,1}, \ldots, \tau_{i,K_i-1}$, forming a DAG. Assume that a schedule is known to be feasible when all FPDS subjobs execute within their stated maximum computation times:

$$\forall_{i,j,k} \left(C^{\iota}_{i,j,k} \leq WC^{\tau}_{i,k} \right) \tag{8.1}$$

We will analyse the cases for interference with higher and lower priority tasks individually.

For *lower* priority tasks, a job $\iota_{i,j}$ can only cause additional interference if its total execution time $C^{\iota}_{i,j}$ exceeds the worst-case computation time of the task:

$$C^{\iota}_{i,j} > WC^{\tau}_{i} \tag{8.2}$$

This does not differ from the situation with FPPS or FPNS tasks; however it can be *detected* earlier if *minimum* computation times $BC^{\tau}_{i,k}$ are available for all subtasks. When the minimum remaining computation time of a task in addition to the past execution time of the running job at the moment a subjob exceeds its worst-case computation time is larger than the worst-case execution time of the entire task, it will cause problems. Let $\iota_{i,j,k}$ be a subjob that exceeds its maximum computation time $WC^{\tau}_{i,k}$. If it holds that:

$$\sum_{k'=0}^{k} C^{\iota}_{i,j,k'} + BCR^{\tau}_{i,k} > WC^{\tau}_{i} \tag{8.3}$$

Then, the minimum computation times of the remaining subjobs guarantee that the job cannot "catch up" using any gain time of subjobs executing in less than their maximum computation times.

Higher priority tasks can preempt between every subjob. From [7] we know that the maximum blocking factor B_h^τ of a task τ_h is:

$$B_h^\tau = \begin{cases} \max_{j>h} \max_{0 \le k < K_j} WC_{j,k}^\tau & \text{for } h < n \\ 0 & \text{for } h = n \end{cases} \tag{8.4}$$

where τ_n is the lowest priority task.

If, with a running job $\iota_{i,j}$ we don't want to increase the blocking factor of a higher priority task, we need to make sure that:

$$C_{i,j,k}^\iota \le \min_{h<i} B_h^\tau \tag{8.5}$$

Seeing as $\forall_{h<i} (B_h^\tau \ge B_i^\tau)$, we can rewrite Equation 8.5 as:

$$C_{i,j,k}^\iota \le B_{i-1}^\tau \tag{8.6}$$

which, according to (8.4) equals:

$$C_{i,j,k}^\iota \le \max_{l \ge i} \max_{0 \le k < K_l} WC_{l,k}^\tau \tag{8.7}$$

In words, the execution time of any subjob of task τ_i should not exceed the worst-case computation time of any subtask of τ_i, or any lower priority subtasks.

Obviously, these thresholds are very pessimistic. They only detect whether the original fixed-priority schedule, which is assumed to be calculated "offline" as being feasible, will be violated, regardless of whether all tasks in the system might still be able to meet their deadlines despite a task overrunning.

8.2.1 Design considerations

RTAI is designed as a thin layer providing fixed-priority scheduling, and does not do any online analysis of schedule feasibility. Our goal is thus to design an extension of RTAI with minimal, efficient support for monitoring and enforcement in the context of FPDS scheduling, that only attempts to interfere when the original offline schedule is not met.

The purpose of enforcement is protection against corrupt or faulty tasks, which implies that enforcement cannot be done by tasks themselves: in a situation where enforcement is necessary, the relevant enforcement routines might never get executed. A possible location to do enforcement checking would be inside the timer interrupt handler. This code is guaranteed to be executed despite aberrant behaviour of normal tasks, and additionally this means that enforcement can coincide with the release of new tasks.

The fact that enforcement will be done from inside an ISR does however mean that its code needs to be as short and predictable as possible. It should not have to do extensive calculations or analysis, and should avoid touching memory as much as possible to prevent memory cache trashing.

While enforcement needs to be done by the RTAI kernel, *monitoring*, and any computation time analysis to be used by the kernel can partially be supported by the task itself - for example, abstracted away inside a preemption point, with all information about the subtask structure available.

Monitoring interference with other tasks

As discussed, an FPDS task can only cause additional interference with the scheduling of *lower priority* tasks if the execution time of the job exceeds the worst-case computation time of the entire task, as with FPPS and FPNS (see Equation 8.2). For the monitoring and enforcement of worst-case computation times against interference with lower priority tasks, the kernel needs the following data about the executing job (see Equations 8.2 and 8.3):

- The execution time since the start of the job
- WC_i^τ
- Optionally, $BCR_{i,k}^\tau$

WC_i^τ can be calculated by the task from the individual values $WC_{i,j}^\tau$ of all subtasks, by calculating the maximum sum of all $WC_{i,j}^\tau$ that form a valid execution path in the subtask DAG. Ideally this is calculated either offline or during task initialisation, and this value can be published to the kernel at the start of the (first) job.

The current execution time of the running job can easily be maintained by the kernel, by storing a timestamp in the task's TCB upon a context-switch to the task, and comparing it with the current time during the enforcement check, or when preempting the task. Alternatively, the execution time can be monitored by the task itself inside the preemption points in a similar way, upon the start of a subjob and at the end of a subjob, before the task yields to higher priority tasks. To implement this efficiently in RTAI, avoiding system calls at every preemption point, for example on Intel CPUs, the `rdtsc` instruction could be used to get the processor cycle count, converted to nanoseconds, without the need to perform a system call.

If minimum computation times are available per subtask ($BC_{i,k}^\tau$), then the future overrun of an FPDS job could be predicted earlier, at the expense of some additional processing overhead (See Equation 8.3). If the kernel has access to a computed lower bound of computation time of the remainder of the job ($BCR_{i,k}^\tau$), it can determine that it is impossible for the job to meet its stated worst-case computation time, and terminate it early if a lower priority task is already waiting. This value can be calculated at each preemption point, and published to the kernel in a similar way as described in the previous sections.

Note that both the worst-case computation time WC_i^τ and the minimum computation time BC_i^τ can become *pessimistic* during the execution of an FPDS task, due to the selection of branches in the DAG describing the structure of the task, if the job departs from the critical path. Using the infrastructure described in Chapter 7, less pessimistic predictions can be maintained in variables that are published to the kernel, with `rt_assign_var()`. This way, they can be updated inside a branching preemption point and inspected by the kernel without requiring expensive system calls.

Monitoring interference with *higher priority* tasks is more difficult. The kernel is not necessarily invoked at preemption points between subjobs, and is therefore not by itself able to closely monitor the behaviour of every subjob, such as the exact time it starts. Therefore we have a *tradeoff* between implementation efficiency, and features such as monitoring and enforcement.

Measurement of the execution time of individual subjobs could in principal be done by the task itself at preemption points, for example using the `rdtsc` instruction to get a timestamp of the start of the subjob, as discussed above. This information can be efficiently communicated to the kernel during preemption points as well

where it can be used for enforcement by the kernel, which will be discussed in the next section.

Enforcement

Enforcement is a difficult problem to solve, as we cannot rely on any code execution within the task itself for it. A single subjob could overrun, and cause interference with both higher and lower priority processes before the kernel is able to detect this, for example in the ISR of a periodic timer. When the periodic timer fires, the offending task may have been exceeding its budget for longer than the schedule can allow.

It might be possible to solve this problem if a low overhead *oneshot timer* is available that can be programmed using absolute times, for example Intel's HPET [19] in the PC platform. During the arrival of other tasks in the timer interrupt handler, the oneshot timer could be programmed to fire when the currently running job is about to exceed its execution time allowance. Using Equations 8.2 and 8.7, we derive that the timer should fire at the minimum of the absolute times where the task will interfere with lower and higher priority tasks, respectively:

$$\overline{f} = \min \begin{cases} s^{\iota}_{i,j} + WC^{\tau}_{i} \\ s^{\iota}_{i,j,k} + \max_{l \geq i} \max_{0 \leq k < K_l} WC^{\tau}_{l,k} \end{cases} \tag{8.8}$$

$\overline{f}$ is the absolute time when the timer should fire, and $s^{\iota}_{i,j}$ and $s^{\iota}_{i,j,k}$ are the absolute start times of the currently running job and subjob.

The value of $\max_{l \geq i} \max_{0 \leq k < K_l} WC^{\tau}_{l,k}$ can be computed and updated upon initialization of every new task. The task itself can publish the value of $s^{\iota}_{i,j,k}$ to the kernel from within a preemption point without much overhead, and $s^{\iota}_{i,j}$ can be determined by the kernel itself at context switches.

If the subjob finishes in time, it will yield for preemption by the higher priority task(s), at which point the kernel is able to cancel the previously programmed timer and avoid the unnecessary overhead of handling the interrupt, but at the cost of overhead of reprogramming the timer.

The method used for the actual enforcement of task budgets is implementation and policy dependent, and outside the scope of this thesis. The kernel could decide to e.g. kill a misbehaving job, suspend it, or make it non-preemptive with a lower priority than any of the tasks it interferes with. Many alternatives are feasible and the optimal strategy depends heavily on the specifics of the real-time application.

Chapter 9

Conclusion

During this Masters project, we successfully designed and implemented a fully functional FPDS scheduler in an existing operating system that is used in production. We considered a few different task models for FPDS with possible mappings to implementations inside real-time operating systems, keeping specifically the architecture of RTAI in mind. The chosen design, using minimal modifications to the scheduler in the kernel with a support library in user space worked out well: we arrived at an implementation that met all important design aspects that we mentioned in Section 4.3. We have shown that it is feasible to implement FPDS efficiently and with low overhead, and verified this through a series of measurements. Following our initial realisation of FPDS and our first measurements, we identified the unconditional execution of system calls in preemption points as the biggest source of overhead and eliminated it through the implementation of preemption points using shared memory. This optimization work also provided some opportunities for extensions and applications of FPDS, where knowledge of the subtask structure of tasks can be exploited without incurring too much extra overhead. We briefly touched upon this in Chapter 8, where we looked at possibilities for monitoring and enforcement of task budgets.

Unfortunately, unlike our expectations at the start of this project, the surveillance platform for which FPDS was identified as a potential solution did not become available, partly due to the selection of a different real-time platform than RTAI. It would have been interesting to use the platform as a real-world test case of FPDS, to verify whether the alleged problems of context switch overhead and resource access control could indeed be solved by it.

For future work, we expect that the overhead of FPDS can be decreased further by the reduction of interference by interrupt handlers. Although less expensive than a full process switch, the execution of an ISR does constitute a context switch which does interrupt a non-preemptive task. Using one-shot timers it may be possible to reduce or eliminate this interference, because timer interrupts can be restricted to moments of actual task arrivals.

The process of understanding the architecture, design and implementation of RTAI, having relatively little documentation, took longer than was expected at the start of the project. The lack of abstraction, refactoring and consistency inside the RTAI source code compared to e.g. the Linux kernel itself gives the open source project a high barrier for entry by external programmers. We were also surprised to find a lot of unpredictable behaviour while running RTAI, its test suite and our own measurement applications on different PC systems. We suspect that this was caused by some design aspects of the modern PC platform which is targeted more at general performance rather than predictability, as well as some interference between timer handling in recent Linux kernel versions and RTAI, which has not been fully

adapted yet.

Some other lessons learned:

- No availability of documentation of unfamiliar systems is sometimes better than having incorrect documentation.
- Performing measurements and experiments is prone to take longer than expected due to unexpected practical complications, and should be scheduled early.
- The PC platform is optimized for efficiency, not predictability, and is not necessarily the best choice for real-time systems.

Bibliography

[1] RTAI 3.6-cv - The RealTime Application Interface for Linux from DIAPM, 2009.

[2] N.C. Audsley, I.J. Bate, and A. Burns. Putting fixed priority scheduling theory into engineering practice for safety critical applications. In *Proc. 2^{nd} IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 2–10, June 1996.

[3] M. Bergsma, M. Holenderski, R.J. Bril, and J.J. Lukkien. Extending RTAI/Linux with fixed-priority scheduling with deferred preemption. In *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 5–14, 2009.

[4] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd edition.* O'Reilly, 2005.

[5] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE Computer Society, 2008.

[6] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited – with extensions for ECRTS'07 –. Technical Report CS Report 07-11, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven (TU/e), The Netherlands, April 2007.

[7] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems Journal*, 42(1-3):63–119, August 2009.

[8] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Advances in Real-Time Systems*, pages 225–248. Prentice-Hall, 1994.

[9] A. Burns, M. Nicholson, K. Tindell, and N. Zhang. Allocating and scheduling hard real-time tasks on a parallel processing platform. Technical Report YCS-94-238, University of York, UK, 1994.

[10] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[11] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (3^{rd} edition).* Addison-Wesley, 2001.

[12] A. Burns and A.J. Wellings. Restricted tasking models. In *Proc. 8^{th} International Real-Time Ada Workshop*, pages 27–32, 1997.

[13] G.C. Buttazzo. *Hard real-time computing systems - predictable scheduling algorithms and applications (2^{nd} edition)*. Springer, 2005.

[14] R. Gopalakrishnan and Gurudatta M. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. *SIGMETRICS Perform. Eval. Rev.*, 24(1):1–12, 1996.

[15] M. Holenderski. Real-time system overheads: a literature overview. Computer Science Report 08/26, Eindhoven University of Technology, 2008.

[16] M. Holenderski, R. J. Bril, and J. J. Lukkien. Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth. In *ECRTS '08 WiP: Proceedings of the Work in Progress session of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[17] M. Holenderski, R. J. Bril, and J. J. Lukkien. Swift mode changes in memory constrained real-time systems. Technical Report CS-09-08, Eindhoven University of Technology, 2009.

[18] M. Holenderski, R. J. Bril, and J. J. Lukkien. Swift mode changes in memory constrained real-time systems. In *Proc. IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC09)*, 2009.

[19] Intel Corporation. *IA-PC HPET (High Precision Event Timers) Specification*, 1.0a edition, October 2004.

[20] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[21] G. Racciu and P. Mantegazza. *RTAI 3.4 User Manual, rev 0.3*, 2006.

[22] J. Simonson and J.H. Patel. Use of preferred preemption points in cache-based real-time systems. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS)*, pages 316–325, April 1995.

[23] Wikipedia. Computer multitasking — wikipedia, the free encyclopedia, 2009. [Online; accessed 2-June-2009].

[24] Wikipedia. Hypervisor — wikipedia, the free encyclopedia, 2009. [Online; accessed 20-July-2009].

[25] Wikipedia. Priority inversion — wikipedia, the free encyclopedia, 2009. [Online; accessed 22-July-2009].

[26] Wikipedia. Ring (computer security) — wikipedia, the free encyclopedia, 2009. [Online; accessed 25-June-2009].

[27] X. Zhou and P. Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 352–357, New York, NY, USA, 2006. ACM.