A FRAMEWORK FOR FLEXIBLE SCHEDULING IN REAL-TIME MIDDLEWARE

Alexandros Zerzelidis

Submitted for the degree of Doctor of Philosophy

The University of York
Department of Computer Science

November 2007

Abstract

The traditional vehicle for the deployment of a real-time system has been a real-time operating system (RTOS). In recent years another programming approach has increasingly found its way into the real-time systems domain: the use of middleware. Examples are the so called pervasive systems (embedded, interactive but not mobile), and ubiquitous systems (embedded, interactive and mobile), e.g. hand-held devices. These tend to be dynamic systems that often exhibit a need for flexible scheduling because of their operating requirements or their execution environment. Thus, today there is a true need in many real-time applications for more flexible scheduling than what is currently the state-of-practice. By flexible scheduling we mean the ability of the program execution platform to provide a range of scheduling policies, all the way from hard real-time to soft real-time policies, from which an application can choose one most suited to its needs. Furthermore, some applications may need to be scheduled by one policy while others may need a different policy, e.g. fixed priority or earliest deadline first (EDF) for hard real-time tasks, least slack time first (LST) or shortest remaining time for soft real-time tasks. It would be difficult for the middleware to expect this functionality from the RTOS. This would require a fine balance to be struck in the RTOS between flexibility and usability, and many years will probably pass until such approaches become mainstream and usable.

This thesis maintains that this flexibility can be introduced into the middleware. It presents a viable solution to introducing flexible scheduling in real-time program execution middleware in the form of a flexible scheduling framework. Such a framework allows use of the same program execution middleware for a variety of applications – soft, firm and hard. In particular, the framework allows different scheduling policies to co-exist in the system and their tasks to share common resources. The thesis describes the framework's protocol, examines the different types of scheduling policies that can be supported, tests its correctness through the use of a model checker and evaluates the proposed framework by measuring its execution cost overhead. The framework is deemed appropriate for the types of real-time applications that need the services of flexible scheduling.

Contents

List of Figures

List of Tables

Acknowledgements

I wish to express my deepest gratitude to my supervisor, Professor Andy Wellings, for his continual and essential support throughout the course of my research. His guidance and encouragement have truly defined for me what a supervisor should be like, and without them this work would never have been completed. I would also like to sincerely thank Professor Alan Burns for the crucial help he offered me throughout a series of discussions on the protocol, and for the understanding he showed as Head of the Department through the darker moments of my research life. I am also grateful to Dr. Neil Audsley, who has been my assessor for the best part of my research effort and whose comments during our meetings were always constructive and encouraging.

I would like to express my gratitude to the Greek State Scholarships Foundation (I.K.Y.) for funding me for the best part of my research, to Microsoft Research (in particular Dr. Fabien Petitcolas) for providing me with a one-year grant, and to the University of York's Student Financial Support Unit for extending valuable financial support. Without them this research work would be impossible.

I would further like to thank all the staff and fellow research students, past and present, in the Real-Time Systems Group for creating an ideal working environment where I always felt "at home" (evident from the *long* hours which I have spent in my office during all these years). I would like to thank Osmar Marchi dos Santos for our discussions and his help with model checking. Also, thanks are due to Rob Davis and Ian Broster for being always willing to hear my questions and provide me with answers. My thanks also go to Erik Hu, Jagun Kwon, Andrew Borg, Armando Aguilar-Soto and Adam Betts for our relaxing conversations and useful discussions, and to Attila Zabos for our discussions throughout many of the long ours spent in the office and for his help.

In York I have had the pleasure of making good friends who have helped me with their support throughout these years. Amongst them I would like to acknowledge Nikolaos Nasios and Georgios Despotou. Also, Dimitris, Tzeni, Giannis and Margaritis for their endless hospitality. Back home in Greece my friends have also offered me immeasurable support during my studies, as they

have always done. In particular, I wish to thank Florentios, Stergios, Kostas, Komnenos, Dimitris and Grigoris for being always ready and willing to help me.

My deepest gratitude goes to Elpida, who has ceaselessly and selflessly stood by my side all these years, making my life brighter and happier.

Finally, I wish to deeply thank my family, Xenophon, Elpida and Maria, for the encouragement and love with which they have surrounded me during these years and all my life.

Declaration

Certain parts of this thesis have appeared in previously published papers; specifically the following references:

Zerzelidis A, Wellings A J. 2006. Getting More Flexible Scheduling in the RTSJ. In Proceedings 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06), IEEE Computer Society, pp.3-10.

Zerzelidis A, Wellings A J. 2006. Model-based verification of a framework for flexible scheduling in the real-time specification for Java. In Proceedings 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'06), vol. 177. ACM Press, pp.20-29.

Zerzelidis A, Burns A, Wellings A J. 2007. Correcting the EDF protocol in Ada 2005. ACM Ada Lett, Vol. XXVII Issue 2 (Aug. 2007).

Chapter 1

Introduction

A *real-time system* is one where correctness of a computation is dependent upon its completion time. The term "real-time" denotes the need for such systems to respond to real-world events taking place in real-world time, in a predictable fashion. Producing a response either too early or too late is equally erroneous. Furthermore, a *hard real-time system* is one where incorrect completion time, e.g. failure to meet a deadline, is considered to be an incorrect result with serious or even disastrous consequences for the system and its environment. In contrast, a *soft real-time system* is tolerant of incorrect completion times. Such a result, however undesirable, can still be tolerated by a soft real-time system.

The traditional vehicle for the deployment of a real-time system is a real-time operating system. With this approach a real-time application is developed only for a particular operating system. However, in recent years another programming approach has increasingly found its way into the real-time systems domain: the use of middleware. A *middleware system* is a piece of software that functions as a conversion or translation layer between two

different types of applications, in a way that *always* allows an application of the first type to successfully communicate with any application of the second type. In other words, it allows applications of one type to be *portable* with respect to another type. There are various types of middleware systems, some examples being web browsers, database middleware, program execution middleware (also known as universal computing middleware (UCM)) and distributed computing middleware [CDE 2007a]. The appeal of the middleware concept to the real-time community is based on two main reasons: the need for portable real-time software and the need for distributed real-time computing. Program execution middleware covers the first need while distributed computing middleware satisfies the second. This has been done through the development of real-time versions for both middleware types.

Real-time program execution middleware is a type of middleware that sits between a real-time operating system (RTOS) and an application program, and enables the application to run on any RTOS/hardware platform without modification, while providing real-time guarantees for the execution of the application. Such middleware is responsible for providing all the functionality needed to a real-time application that would normally be found in the operating system. Examples of this functionality include scheduling of real-time tasks, resource sharing, memory management and memory access, asynchronous event handling and asynchronous transfer of control. This is extremely important as it completely, or at least to a very large extent, removes the cumbersome process of porting the application. Whereas in desktop environments porting is less of an issue, because of the dominance of the x86 architecture, the problem has remained significant for real-time systems. The main reason behind this is the fact that most of today's real-time systems are embedded systems, specifically consumer electronics. It is often the case with such systems that many different host architectures are used, either simultaneously or because of the evolution in a product's lifecycle. Code portability in such systems is particularly important, since it allows for faster time-to-market and lower production costs. For example, if, during development of an embedded system, implementation decisions dictate that a more powerful platform should be used, then, with portable code, transition to a new CPU would be almost trivial, since code would (ideally) not have to be re-written. This also applies to the development of new versions/generations of embedded systems where code stays almost the same, while the hardware is updated. Therefore, real-time program execution

middleware have an important role to play. An example of this kind of middleware is the Real-Time Specification for Java (RTSJ).

Distributed real-time middleware is a type of middleware that enables software components, possibly written in multiple computer languages and running on multiple computers, to work together. This is an important and growing segment of the IT market. Examples include distributed control of large-scale telecom switching systems and autonomous vehicles over wireless links. Distributed real-time systems can run the range of requirements from hard real-time, e.g. avionics mission computing, to soft, yet stringent, real-time requirements, e.g. telecommunication call processing. An example of such middleware is real-time CORBA.

The two types of middleware come together to form what is known in recent years as *architecture-neutral* real-time systems [CDE 2007b]. An architecture-neutral real-time system is a real-time system whose target architecture is unknown at system design time. The main focus of architecture-neutral real-time systems typically is embedded systems, including so called pervasive systems (embedded, interactive but not mobile), and ubiquitous systems (embedded, interactive and mobile), e.g. hand-held devices. However, their characteristics are also close to those of Internet applications, for which we do not a priori know the target architecture. Such real-time systems have unknown target architecture because:

- they have to be executable on the widest range of architectures possible in order to increase their portability;
- their lifetime is expected to be greater than ten years and, therefore, they have to be immune to technology obsolescence;
- their site of execution may vary.

As we can see, the two types of middleware discussed above are both essential to the realisation of architecture-neutral real-time systems. As an example of a ubiquitous system we can think of a mobile phone user entering a zone where a particular application provides him with certain information in real time. In this case the application does not have a priori knowledge of every possible mobile phone it might run on.

1.1 Real-time scheduling at the middleware level

Middleware systems, as described above, tend to be dynamic systems that often exhibit a need for flexible scheduling, either because of their operating requirements or because of their execution environment. Thus, today there is a true need in many real-time applications for more flexible scheduling than what is currently the state-of-practice. By *flexible scheduling* we mean the ability of the program execution platform, be it the operating system or some higher level middleware software, to provide a range of scheduling policies, all the way from hard real-time to soft real-time policies, from which an application can choose the one most suited to its needs, i.e. a hard-real time application will be using hard real-time policies whereas a soft real-time application will be using soft real-time policies. Scheduling is at the very core of every real-time system and is the subject matter of this thesis. More particularly, the thesis presents a viable solution to introducing flexible scheduling in real-time program execution middleware. Thus, the scope of this thesis is not about the scheduling algorithms themselves, but rather about a framework that allows use of the same program execution platform, in our case a middleware platform, for a variety of applications – soft, firm, or hard.

The effort of this thesis to cater for flexible scheduling in middleware is based on two main design constraints. The first is to cater for the state-of-the-art in scheduling. This means being able to support both well-known practices and novel approaches, e.g. dual-priority scheduling [Davis and Wellings 1995]. The second is to take into account the dependency of the middleware on the development platform (operating system and hardware). Attention has been paid to how easy it is going to be for a particular design to be implemented on a variety of platforms. The importance of covering a wide range of platforms lies in addressing the main functionality requirement of a program execution middleware, which is to provide code portability. Code portability increases as the number of supported host OS/hardware architectures increases. Therefore, the effort has been to take as much advantage of the set of common features found across different operating systems as possible, while at the same time requiring no OS changes. Not only does this make the job of porting the middleware easier, it also helps in keeping its performance as uniform as possible across different platforms. This, in a sense, means making the *middleware implementation* as portable as possible.

It has been noted that most real-time operating systems today support pre-emptive priority-based dispatching [Ganssle and Barr 2003], e.g. QNX, any flavour of RT-Linux, Integrity OS, Windows CE and many more. Therefore, we consider a fixed-priority policy to be the only policy the middleware can rely upon in any particular host. It is not, however, the only policy we want the middleware to support. Today there is a true need in many applications for more flexible scheduling than what is currently the state-of-practice [Regehr et al. 2000], [Brandt et al. 2003], [Jemander 2005]. Furthermore, some applications may need to be scheduled by one policy while others may need a different policy, e.g. fixed priority or earliest deadline first (EDF) for hard real-time tasks, least slack time first (LST) or shortest remaining time for soft real-time tasks.

This is the reason that state-of-the-art real-time operating systems nowadays support hierarchical scheduling [Aldea et al. 2006], [Jones and Regehr 1999], [Goyal et al. 1996]. This is a logical evolution of existing systems, which already use a very basic form of hierarchical scheduling found in fixed priority scheduling. That is, a fixed-priority scheduler first executes the tasks with the highest priority and from within these tasks, it chooses the oldest runnable task, i.e. FIFO within priorities. The hierarchical nature of fixed-priority scheduling is often not realised, as it is taken for granted. However, support for more sophisticated hierarchical scheduling in the operating system requires a fine balance to be struck between flexibility and usability, and many years will probably pass until such approaches become mainstream and usable. Even then it will be hard to agree on any one such mechanism to be used in all operating systems. In other words, it will be very difficult for such mechanisms to reach the level of acceptance and integration that fixed-priority scheduling has. Therefore, the middleware cannot yet rely on such state-of-the-art mechanisms for providing flexible scheduling. However, the middleware can introduce the extra flexibility in scheduling at its level, relying only on fixed-priority based dispatching from the operating system.

1.2 Thesis proposition

Based on the above, it is our thesis that *a two-level scheduling framework, having a fixed-priority base scheduler and support for other schedulers through dynamic*

manipulation of task[1] priorities, provides an effective framework for flexible scheduling in real-time middleware software.

The above statement implies that the middleware's default scheduler is based on the operating system priority scheduler. This constitutes its base scheduler, which the middleware usually augments with extra functionality. In many middleware systems such a base scheduler is the only one defined and directly supported. However, using the described framework a middleware application can implement its own scheduler, which may have its own notion of execution eligibility. Adopting this approach allows multiple schedulers to interoperate and is sympathetic to the fact that real-time software is costly to develop, maintain, and evolve in the face of changing scheduling or quality of service (QoS) requirements. It must be noted, however, that this thesis is not concerned with issues of fault tolerance, i.e. no mechanism is defined for preventing an application-defined scheduler from misbehaving.

1.3 Thesis structure

Chapter 2 contains a literature survey of current approaches to introducing flexible scheduling in real-time middleware and also in real-time systems in general.

Chapter 3 considers the problem of resource sharing under the prism of flexible scheduling. The chapter, staying true to the design decision of taking as much advantage as possible of the set of common features found across different operating systems, reviews only the most widely known and used priority inversion avoidance algorithms and selects the one most suited for supporting flexible scheduling.

Chapter 4 presents the Flexible Middleware Scheduling Framework (FMSF), a generic framework for task scheduling at the middleware level that allows multiple, diverse, user-defined scheduling policies to co-exist in the system, each one dictating the execution of a particular subset of the task set. The chapter describes the framework, starting from the assumptions made, the

---

[1] A *task* is a finite sequence of instructions to be executed on a single processor and is the smallest schedulable entity in the system. Any particular task can have many instances of execution. Each such instance is called a *job*.

choice of protocol, and concluding with the description of the actual mechanism.

Chapter 5 provides a three-faceted evaluation of the framework. The first part of the evaluation is the description of an application-defined scheduler implemented in the framework. Our choice is an EDF scheduler, which has gained significant support over the past several years as a valuable supplement to fixed-priority scheduling. Then, a verification of the framework's operation is provided, which is achieved by modelling it as a system of timed automata in the UPPAAL model checker. This provides a strong indication that a correct implementation of the framework's protocol is possible, by testing a number of characteristic cases. Finally, we present an examination of the types of different scheduling policies that can be supported by the framework. The extended range of policies covered can satisfy a plethora of application demands and diverse real-time needs. This demonstrates the framework's full scope.

Chapter 6 provides a practical evaluation based on a simulated implementation on the Real-Time Specification for Java (RTSJ). The rational for choosing RTSJ has been the wide acceptance of the Java platform, in general, and the growing relevance of the RTSJ for real-time computing. This guarantees that the evaluation will be relevant to a significantly broad review audience.

Finally, Chapter 7 gives a summary of the contributions contained in this thesis and provides suggestions for continued research.

As a final note it might be worth mentioning that the citation style used in this thesis is derived from the Council of Biology Editors (CBE) Name/Year style, with the exception that square brackets are used instead of parentheses to denote an in-line citation.

Chapter 2

Flexible Scheduling

Flexible scheduling in real-time systems is an emerging field of study with two complementary research directions: the invention of flexible scheduling algorithms and the development of flexible scheduling frameworks. The work presented in this thesis belongs to the latter category. A flexible scheduling framework constitutes a far more general approach, since it provides the developer with a flexible way of choosing amongst different scheduling algorithms. More often than not flexible scheduling algorithms focus on addressing the needs of particular systems by providing specialised solutions. Under a flexible scheduling framework an application can be scheduled by either a hard policy or a flexible policy or a combination of the two. Such a framework can be implemented at the operating system or at the middleware level.

On a basic level, a system can be thought of providing a flexible scheduling framework only if it provides a mechanism for enforcing more than one scheduling policy. There are, generally, three approaches to introducing a new scheduling policy to a system:

Pluggable schedulers – in this approach the system provides a framework into which different schedulers can be plugged at start-up time. This method can be very efficient, but it has the drawback that the loaded scheduling modules can neither be isolated from each other nor from the kernel itself, so a bug in one of them could affect the whole system. The CORBA Dynamic Scheduling [OMG 2005] specification is an example of this approach. Kernel loadable schedulers also fall into this category, an example being the SHaRK kernel [Gai et al. 2001].

Application-defined schedulers – in this approach, a scheduler can be added to the system by an application at any time. In this sense the scheduler is part of the application. The system notifies the application scheduler every time an event occurs that requires a scheduling decision to be taken. The application scheduler then informs the system which thread should execute next. The proposed extensions to real-time POSIX support this approach [Aldea and Harbour 2002].

Implementation-defined schedulers – in this approach, an implementation is allowed to define alternative schedulers. Typically this would require the underlying implementation (operating system or virtual machine) to be modified. The Ada 95 and real-time Java (RTSJ) languages allow this approach.

Furthermore, a system may allow one or multiple schedulers to be active at the same time. This functionality is extremely helpful in creating flexible multi-purpose computing systems that need to handle applications with significantly different scheduling needs. If multiple schedulers are allowed, there has to be some kind of hierarchy amongst them that regulates which scheduler is in control at any one time. Therefore, a second classification for flexible scheduling frameworks is whether they allow hierarchical scheduling or not. Of course, at a very basic level even fixed-priority scheduling can be thought of as being a two-level hierarchical scheme, with the first level being the priority scheduler and the second being a FIFO or round-robin scheduler.

In general, each of the above solutions can be implemented on either the operating system or the middleware level. The purpose of this chapter is to survey the existing literature and practice on flexible scheduling frameworks, and even though the focus of this thesis is universal computing middleware, the survey presents solutions from a much broader range of implementations. Section 2.1 presents related research in real-time middleware. Section 2.2 presents research on operating system facilities. Section 2.3 describes current

and proposed facilities for two major real-time programming languages, Ada and real-time Java (RTSJ).

2.1 Middleware research

A lot of the effort for flexible real-time scheduling has gone into real-time middleware solutions. This is an often preferred and effective solution, since a middleware approach can more easily abstract away the particulars of the operating system scheduler. This section will examine a number of such efforts.

Distributed real-time middleware: Prominent among the efforts for distributed real-time systems is the Real-time Common Object Request Broker Architecture (CORBA) Specification by the Object Management Group (OMG) [OMG 2005]. Real-time CORBA (RT-CORBA) aims to offer end-to-end predictability in distributed real-time applications. An RT-CORBA system, being a middleware solution, needs to rely on a base operating system scheduler. Therefore, work in RT-CORBA is of relevance to our work. An RT-CORBA system is based on:

- the real-time operating system
- the real-time object request brokers (ORB)
- the communications transport
- the application(s)

Real-time scheduling in such a system is carried out by the ORB. The ORB relies on the real-time operating system to dispatch threads and to provide mutexes. Scheduling is based on an end-to-end schedulable entity called *distributable thread* (DT) [OMG 2005]. A distributable thread can extend and retract its locus of execution across physical computing nodes by location independent invocations and (optionally) returns. Within each node, the flow of control is equivalent to normal local thread execution. Each DT has a unique system-wide id across all nodes. Moreover, each DT may have one or more execution scheduling parameter elements, e.g. priority, deadline, or importance. Execution of the DT is governed by the scheduling parameter elements, on each node it visits. A DT has only one head of execution, regardless of the number of nodes it spans. The DT interacts with the ORB scheduler of the node it currently runs in at predefined scheduling points (e.g.

application calls, locking, CORBA invocations). An important characteristic of the specification is that it does not define a global scheduling mechanism. Rather, each ORB performs scheduling in its own node, based on scheduling information passed on by the distributable thread.

[OMG 2005] defines two mechanisms for real-time scheduling in a CORBA system:

- a classic fixed-priority scheme
- a dynamic scheduling scheme

Under the fixed-priority scheme, RT-CORBA defines its own priority levels, in order to provide scheduling consistency across different nodes. On each node these priorities map down to the local operating system priorities. A distributable thread carries along with it its RT-CORBA priority when making a remote invocation. The priority is translated to a remote node's local priority and the local operating system schedules the thread according to that priority.

When using the dynamic scheduling scheme, the specification allows an ORB to implement a dynamic scheduling policy through the use of a pluggable scheduler. Each ORB utilises only one scheduler. The dynamic policies described in the specification are *earliest deadline first* (EDF), *least laxity first* (LLF) and *maximize accrued utility* (MAU). The specification does not address interoperation between different dynamic scheduling algorithms running at different nodes. This also means that threads residing at different nodes and scheduled by different schedulers cannot share mutually exclusive resources. Thus, the flexibility offered by the specification is limited to only being able to specify at system start-up which policy each ORB will follow.

A number of scheduling approaches have been proposed for real-time ORB scheduling under the CORBA dynamic scheduling scheme. [Aswathanarayana et al. 2005] presents an endsystem scheduling framework that emphasises on the balanced progress of application computation components. The framework is based on a scheduling abstraction called group scheduling [Frisbie et al. 2004]. It works by grouping computation components and using an arbitrary policy (e.g. round-robin, fixed-priority) to schedule each group. Enforcement of the particular scheduling policy is achieved with the use of dynamic priority changes, which is a well established technique [Burns and Wellings 1997]. The Kokyu framework, [Gill et al. 2002, 2003], is an ORB scheduling service that provides so-called "strategized

scheduling". It consists of pluggable schedulers and configurable dispatching queues. These queues are priority ordered but can have various dispatching policies, e.g. FIFO, earliest deadline, least laxity. The scheduler specifies the number of queues it is going to need and the type of dispatching policy each queue is going to use. Each queue has one local thread servicing it, which is set to the priority of the queue. The above approaches, although able to support different scheduling policies, do not address the issue of resource sharing between threads scheduled by different schedulers.

To address the problem of the interoperation of different dynamic schedulers across different nodes Corsaro et al. [2001] defines a meta-level model that enables inter-working between diverse scheduling policies by translating scheduling properties from one policy to another. By defining the terms property, competitor and scheduler the paper goes on to define an adapter function that translates the properties of competitors according to one scheduler to properties according to another scheduler. Sharing of resources is not specifically addressed. It is left to each ORB's local scheduler to address the problem.

In addition, there have been several efforts to augment RT-CORBA with an end-to-end distributed real-time scheduling service (e.g. [Zhang et al. 2005], [DiPippo et al. 2001], [Kalogeraki et al. 2000], [Wolfe et al. 1999]). Of those, only [DiPippo et al. 2001] provides a distributed protocol for bounding priority inversion, but, on the other hand, it supports only deadline monotonic scheduling.

The *CPU Broker* [Eide et al. 2004] is a CORBA-based processor capacity reservation manager. It resides above a particular host's ORB rather than within the ORB. The broker can mediate between both CORBA and non-CORBA applications and the RTOS, and can implement a variety of different, usually "soft", reservation schemes. Reservations are negotiated on two levels. The top level, called the advocate, receives an initial reservation request from the application. There is at least one advocate per application. Each advocate processes its application's request based on feedback from the application itself and from the system (e.g. consumed CPU time). It then forwards the, possibly modified, reservation request to the second level. The second level implements a system-wide reservation policy. It is responsible for making the final request to the RTOS. As a response to an advocate's request the system-wide policy may re-compute the reservations for all the broker's managed

applications. The CPU broker, unlike our framework, is especially targeted towards soft real-time dynamic systems. The framework proposed in this thesis is able to address a broader set of applications, since it can enforce a broader range of scheduling policies.

Other middleware: An interesting solution to the problem of executing both real-time and non-real-time applications on a single processor is presented in [Deng and Liu 1997]. Their system is based on an EDF operating system scheduler that sets up and schedules different servers. Each real-time application has a dedicated server, while all non-real-time applications are executed by a single total bandwidth server [Spuri and Buttazzo 1996]. A real-time application server can be either a constant bandwidth server [Deng et al. 1996] or a total bandwidth server. Within each server tasks are scheduled according to a particular scheduling algorithm by the server scheduler. The server scheduler for non-real-time applications uses a time-sharing algorithm. The server schedulers of real-time applications are chosen by each real-time application. Hence, this scheme supports two-level hierarchical application-defined scheduling, as does the framework presented in this thesis, with the difference that our approach is based on a fixed-priority scheduler, which is much more common amongst RTOS. Resource contention within a server is resolved according to the locking policy used by the application. Resource contention across different applications is handled by the non-preemptable critical section (NPS) protocol [Mok 1983]. This is a rather crude way of dealing with priority inversion. According to this protocol, whenever a task holds a globally shared resource, it becomes non-preemptable and remains as such until it releases all such resources. In fact, the whole server that executes the locking task becomes non-preemptable. This introduces priority inversion to tasks that are not using the particular resource and would normally preempt the locking task. In contrast, our framework uses a more efficient protocol that bounds priority inversion to the longest critical section of a lower eligibility thread. To address the issue of portability across the vast majority of commercial RTOS, Kuo and Li [1999] translated this framework to one based on a fixed-priority scheduler. In their system all servers are sporadic servers [Sprunt et al. 1989], because of the incompatibility of the constant utilisation and total bandwidth servers with the fixed-priority scheduler.

Another approach for flexible scheduling based on POSIX-compliant RTOS is given by Li et al. [Li et al. 2004]. They have provided a formalized POSIX framework, which aims at supporting various utility accrual (UA)

scheduling algorithms, but can also support non-UA scheduling policies. The heart of the framework is the *meta scheduler*. The meta scheduler runs as a separate POSIX thread and enforces a particular policy by dynamically changing application thread priorities, as in [Burns and Wellings 1997]. Scheduling decisions are dictated by application-defined schedulers that the meta scheduler queries at each scheduling event. Their approach is similar to the approach presented in this thesis, with three major differences. First, the meta scheduler does not implement a particular priority inversion avoidance algorithm. Presumably, each application scheduler implements its own locking policy, although this is not specified. However, this means that resource sharing between different schedulers is not supported. Secondly, the meta scheduler keeps a list of resources that it manages. These resources cannot be used by POSIX threads running outside the meta scheduler. If outside threads could use them, then the operating system would apply its own priority inheritance mechanism, causing threads to change priority without the knowledge of the meta scheduler. Thirdly, the meta scheduler runs as a separate process in the operating system. In contrast, this thesis suggests that scheduling operations should be made in the context of scheduled threads. This latter approach greatly reduces the number of context switches and hence the amount of overhead the application experiences.

Another processor reservation manager, similar to the CPU Broker above, is the Dynamic Quality-of-Service Resource Manager (DQM) [Brandt et al. 1998], [Brandt and Nutt 2002]. This middleware is illustrative of a category of systems that approach the issue of flexibility in scheduling with the notion of "quality of service" (QoS). More specifically, DQM uses and extends the model of QoS levels introduced by Tokuda and Kitayama [1993]. A QoS level describes the amount of CPU an application is entitled to. In general, the higher the level, the higher the CPU time allocation. The DQM is designed to address the soft real-time needs of desktop applications. It provides implementation-defined non-hierarchical scheduling. The resource manager can be initialised with one of four approximation algorithms that select the level at which each application must execute. It is explained that the user can affect the enforced policy by choosing different parameters for the different QoS levels that their application will run at. However, this is nothing more than the equivalent of choosing a priority in a fixed-priority system. Also, the programmer must specifically program her application to use the DQM by calling certain functions. Overall, the approach is less flexible than the framework presented in this thesis because of three reasons: i) CPU allocation

policies are implementation-defined instead of application-defined, ii) applications running under DQM must be specially programmed to interface with the manager, and iii) DQM does not provide any synchronisation mechanism for sharing non-CPU resources.

The concept of servers is used by Kaneko et al. [1996]. The authors use an SGI Challenge multiprocessor system running IRIX and implement a planning-based scheduler on top of the operating system priority scheduler. The planning-based scheduler runs at a very high priority and its task is to direct the execution of hard real-time tasks and of multimedia servers running on different processors (one server per processor). It performs admission control on newly started tasks and dynamically generates a schedule of feasible tasks. Tasks running directly under the planning-based scheduler are guaranteed all required resources, including a processor. These are the hard real-time tasks. When a multimedia task arrives, the planning scheduler has to again access its feasibility. If it is schedulable, a server is created for it on an available processor, or the task is added to an existing server. The server, on its part, schedules multimedia tasks according to one of four allocation policies, which is chosen at start-up time. This approach provides a hierarchical implementation-defined scheduling scheme. Scheduling flexibility is very low, since the programmer cannot choose the scheduling policy for hard real-time tasks and has a choice of only four policies for multimedia tasks. Resource sharing is handled by the planning-based scheduler, but in a very restricting way: tasks are added to the schedule only if all required resources are available. Furthermore, there is no specified safe way of sharing resources between hard real-time tasks and multimedia tasks.

2.2 Flexible scheduling in real-time operating systems

Although some specialist operating systems support different scheduling approaches, the vast majority of commercial-off-the-shelf (COTS) real-time operating systems support fixed-priority preemptive scheduling. However, the idea of flexible scheduling has also been applied in many, mainly experimental, RTOS.

One solution is presented by Wang and Lin [1999] in RED-Linux [Wang and Lin 1998], in which a two-level scheduler is used. The central idea in this approach is the use of four scheduling attributes: priority, start and finish

times, and execution time budget. These attributes are meant to capture the basic minimum set of common characteristics across different scheduling algorithms. However, the authors identified this set by examining only three scheduling paradigms. Therefore, although this mechanism can support some scheduling algorithms, there may be others that cannot be implemented, if they are based on parameters different from those chosen. The framework itself is based on a lower kernel-level scheduler, called the *dispatcher*, and an upper level scheduler, called the *allocator*. The allocator can be implemented in either the kernel space or the user space, depending on the implementation characteristics. An application passes its scheduling characteristics to the allocator. The application can also, at runtime, instruct the allocator as to which policy to enforce. This, essentially, constitutes application-defined scheduling. The allocator enforces a particular policy by specifying to the dispatcher which scheduling attributes to use when making a scheduling decision. The attributes that participate in the scheduling decision collectively constitute the *effective priority* of the task. For example, in order to enforce EDF scheduling the allocator specifies the finish time as the effective priority. The scheduling attributes together with the information about a task's effective priority are passed to the dispatcher through a special API. The dispatcher inspects the values of the scheduling attributes that make up the effective priority, chooses one job from the ready queue and dispatches it. Apart from the limited number of supported scheduling algorithms, this solution does not address the implementation of priority inversion avoidance protocols for shared resources. Finally, it is not an easily portable solution, since it is implemented on top of RED-Linux, which is a specially modified version of Linux.

A different approach is followed by Ford and Susarla [1996]. They define CPU Inheritance Scheduling where the application defined schedulers are threads which donate the CPU to other threads. This donation is achieved through the kernel dispatcher, which only implements thread blocking, unblocking and CPU donation. The dispatcher is not a thread, rather it executes in the context of the running thread. Furthermore, it is not a standard RTOS dispatcher, e.g. it does not have any notion of thread priorities. It is each scheduler's responsibility to provide the necessary context for the policy it is implementing. All schedulers are running as user-level threads that instruct the dispatcher to donate CPU time to one of their threads through a `schedule` operation. Scheduler threads can also be scheduled by other schedulers. Thus, a scheduler hierarchy of arbitrary depth is formed with child

schedulers "inheriting" CPU time from their parent scheduler. Only the scheduler at the root of this hierarchy has all the CPU time to donate. In this approach the only method used to avoid priority inversion is a special form of priority inheritance, which may be a limitation for some application-defined policies. On the other hand, the approach theoretically allows any type of scheduler to be implemented and also allows multiple schedulers to co-exist in the system.

Another solution commonly found in RTOS is pluggable schedulers. The application scheduling algorithms are modules to be included or linked with the kernel. With this mechanism the functions exported by the modules are invoked from the kernel at every scheduling point. In RT-Linux [Yodaiken 1999] three scheduling modules are included: the standard fixed-priority scheduler, an earliest-deadline-first scheduler and a rate monotonic scheduler, although it is not clear whether multiple schedulers can co-exist in a system. In addition, there is no protocol defined to handle priority inversion in a case where arbitrary schedulers co-exist in RT-Linux. Vassal [Candea and Jones 1998], a modification of the Windows NT 4.0 kernel, can support a second scheduler, apart from the default system scheduler, which can be dynamically loaded. This scheduler can dynamically choose to schedule its threads or let the native scheduler perform the task. However, there is again the issue of resource sharing between threads scheduled by different schedulers. S.Ha.R.K [Gai et al. 2001] is a third pluggable approach. Multiple scheduling modules can be loaded at system start-up and, thus, a hierarchy of schedulers can be created. The first module to be loaded is the highest scheduler in the hierarchy, the second is the next highest and so on. Whenever a task becomes runnable at a scheduler higher than the scheduler of the running task, the latter gets preempted. Hence, S.Ha.R.K. supports a two-level hierarchical scheduling scheme with priority scheduling at the root of the hierarchy and an arbitrary number of pluggable schedulers at the second level. The approach is more flexible than the previous two in the sense that pluggable locking policy modules can be loaded as well at start-up, thus providing integrated support for a number of priority inversion avoidance policies, e.g. priority inheritance, priority ceiling emulation [Sha et al. 1990], stack resource policy [Baker 1991] etc.

Application-defined scheduling is offered in MaRTE OS [Aldea and Harbour 2001, 2002, 2004b]. An application-defined scheduler can be implemented as either a separate scheduler thread or its functionality can be

executed within the context of a scheduled thread. Furthermore, in the former case a scheduler thread can either run in user space or, for increased efficiency, in kernel space. To simplify the effort of implementing application-defined policies the concept of a task's "urgency" is introduced. The urgency of a task is its priority according to a particular scheduling policy. So, for example, for EDF scheduling the urgency is the absolute deadline. This is reminiscent of the notion of effective priority in RED-Linux. The idea is that when priority queues are ordered by urgency the dispatcher automatically selects the most eligible task. However, this notion of urgency does not suffice for policies where task urgencies change dynamically during a task's release. Rearranging the queue under constant changes in urgency values would probably be too costly. For the sharing of resources the approach supports the standard POSIX protocols (priority inheritance, priority ceiling emulation), and also offers an urgency inheritance protocol and the stack resource policy, which, as is shown in Chapter 5 of this thesis, supports a variety of scheduling policies. A potential weakness compared to the framework in this thesis is that application scheduled threads do not run in clear separation of system scheduled threads. A thread scheduled by an application scheduler can execute at the same priority level as system scheduled threads, either in FIFO order or round-robin. This can more easily lead to erroneous situations where an application scheduled thread executes at the wrong priority level and gets preempted by system scheduled threads. In contrast, the flexible middleware scheduling framework in this thesis defines bands of execution where threads under a particular application-defined scheduler run. The only way to run at the same priority as system scheduled threads is if a band thread locks a resource also used by system threads.

The European FIRST project [FIRST 2005] is a joint effort to add scheduling flexibility to the RTOS. The project has produced the FIRST Scheduling Framework (FSF) [Aldea et al. 2006], whose key abstraction is the *service contract*. The establishment of service contracts between the applications and the underlying scheduler is meant to capture the scheduling requirements of each application and are independent of the scheduling policy used. The result of an accepted contract is the creation of a *server* particular to the application. Various server schemes can be used, e.g. constant bandwidth servers on top of an EDF scheduler, or sporadic servers on top of fixed priorities. Application threads are run directly by the server. Alternatively, the FSF can be extended to a two-level hierarchical scheduling architecture, according to which the lower level scheduler is the scheduler that takes care of

the service contracts, and each of the top-level schedulers runs within a particular FSF server and schedules application threads according to its particular policy. At the moment, FSF provides top-level schedulers only as part of its implementation, i.e. top-level schedulers are implementation-defined. The reason for this, as explained in [Aldea et al. 2006], is that it is simpler than having a specific API for application schedulers. However, this makes the framework less flexible than if it was supporting application-defined scheduling. Another issue regarding FSF is the fact that there is no run-time mechanism for mutual exclusion in shared resources. Two reasons are given for this: the first is upward compatibility of legacy code that is using regular mutexes, and the second is that enforcing worst case execution time for critical sections is computationally expensive. Thus, there is no standard way of sharing resources between threads scheduled by different schedulers. A further problem is the fact that there is no standard way of providing FSF functionality. As explained in [Aldea et al. 2006], "each FSF implementation would have to be tailored to a specific RTOS". Therefore, the FSF, at the moment, does not promote portability. However, an effort is underway [Aldea and Harbour 2004] to incorporate the new mechanisms into the POSIX real-time operating system standards [IEEE 2004]. In this respect, it is very helpful that due to the use of servers the framework is independent of the underlying operating system scheduling policy. This has been demonstrated by implementing the framework on two RTOS, MaRTE OS [Aldea and Harbour 2001] and SHaRK [Gai et al. 2001].

2.3 Flexible scheduling in real-time programming languages

In this section we will examine the approach taken and the proposals made for two major real-time languages, Ada [Ada-Europe 2007] and real-time Java [Bollella et al. 2000], [Belliardi et al. 2006]. Ada is, perhaps, the best known real-time language, having being specifically designed for building mission-critical military systems in the mid-seventies. Java is a relatively late entry into the genre, but shows much promise and has amassed great support for real-time development over the last ten years. It is, thus, worth looking at how these two popular real-time programming tools promote the development of flexibly scheduled systems.

Ada recently underwent a major revision and several new features have been included in the new reference manual [Ada-Europe 2007]. Ada supports one type of scheduling, fixed-priority scheduling. However, new dispatching policies have been added, which dictate the way tasks are ordered in a priority queue. Apart from FIFO ordering, which can be either preemptive or non-preemptive, Ada now offers round-robin [Burns et al. 2003] and earliest deadline first dispatching [Burns et al. 2004]. All these policies are implementation-defined, therefore, as we can see the language supports a two-level hierarchical implementation-defined scheduling scheme, where the low-level scheduler is fixed-priority based and the second level is defined by the priority queue order. FIFO and round-robin dispatching both use the priority ceiling emulation protocol to handle resource contention. Earliest deadline first dispatching enforces priority inversion avoidance through the preemption level protocol (PLP) [Burns et al. 2004], which is a special version of the stack resource policy [Baker 1991]. The mechanics of the PLP, though, are the same as those of the priority ceiling protocol. Therefore, Ada uses the same priority inversion avoidance mechanism across all supported policies, which allows for great flexibility in designing a complex real-time system with components running under different scheduling policies. All in all, it can be said that although the number and types of policies are restricted, the existence in Ada of all the basic real-time scheduling policies under one scheme and the ability to use these policies in a cooperative manner is extremely useful to the real-time systems developer and ensures the continuation of the language's relevance in the real-time domain.

During the process of defining the new specification of the Ada language an interesting proposition was made for the endowment of Ada with application-defined scheduling capabilities [Aldea and Harbour 2003], [Aldea et al. 2004]. Application schedulers implement primitive operations that are invoked by the system when a scheduling event occurs, and in their reply they can specify which task to suspend and which to execute. This is approach is similar to the framework of this thesis. One difference is that application schedulers under this framework can exchange information between them, which can lead to the construction of cooperative schedulers. On the other hand, this framework is based on a previous proposal by the same authors [Aldea and Harbour 2002] for application-defined scheduling in POSIX. Since the Ada runtime system is not designed as middleware, OS support is needed and the implementation of their application-defined scheduling for Ada presupposes the existence of their application-defined facilities in the

operating system, or the ability of the operating system to accept kernel-loadable modules. This fact reduces the framework's portability. Furthermore, application schedulers can only use the stack resource policy (SRP) for handling priority inversion and cannot define their own protocol. What is more, the SRP does not cater for tasks suspension while locking. Of course, this is not a problem with Ada, since such behaviour is not allowed. It does mean, however, that the framework is not generally applicable.

The RTSJ adopts the implementation-defined schedulers approach (although it also tries to provide a framework for the implementation to follow) and allows for applications to determine dynamically whether the real-time JVM on which it is executing has a particular scheduler. Unfortunately, this is the least portable approach, as an application cannot rely on any particular implementation-defined scheduler being supported. The only scheduler an application can rely on being present is the `PriorityScheduler`. The work reported in this thesis only assumes the presence of the priority scheduler and that priority changes have an immediate effect. An attempt has been made [Feizabadi et al. 2003] to support a utility accrual scheduler in the RTSJ but this required a non standard interface and was not generalized. Similarly, although JTime by TimeSys supports multiple schedulers, this has been achieved in an ad hoc manner [Dibble and Wellings 2004].

2.4 Summary

A number of solutions, across different application domains, have been proposed over the years to address the issue of flexibility in the scheduling of real-time applications. While all have been useful in demonstrating the feasibility of a flexible scheduling framework, they are not able to offer an adequate solution for the domain of real-time program execution middleware. More specifically, all of the reviewed approaches exhibited one or more of the following limitations:

- inability to support arbitrary scheduling policies
- inability to support multiple co-existing schedulers
- inability to allow the application to select the preferred scheduling policy at runtime (application-defined scheduling)

- lack of portability, e.g. by being based on a particular operating system
- inability to facilitate sharing of non-CPU resources between threads scheduled by different schedulers
- inability to allow applications not using the framework to co-exist and share resources with applications that are using it
- inability to allow applications define their own resource sharing protocol for their own use
- inability to bound execution eligibility inversion as efficiently as possible, i.e. to one instance

As a minimum, an effective solution should address all of the above issues. Such a solution has to be general enough to be able to express the scheduling needs of the widest possible range of applications, while guaranteeing that the approach will be portable to the largest possible number of platforms. The following chapters elaborate on a framework for flexible scheduling in real-time program execution middleware that can host an arbitrary number of arbitrary scheduling policies based on the de facto operating system real-time scheduling standard of a fixed-priority scheduler. The framework can be used to create cooperating real-time applications with diverse scheduling needs.

Chapter 3

Resource sharing for flexible scheduling

The introductory chapter has set forth our thesis and given justification for it. We have argued in favour of a middleware framework that will provide an easy way for applications to supply their own preferred policy for scheduling their tasks. We have suggested that supporting multiple and diverse scheduling policies in the middleware will be significantly helpful to programmers. However, it would be even more useful to allow tasks running under these policies to also share resources in producing a combined functionality. In other words, we want to *allow tasks scheduled by different schedulers to access the same resources* without fear of undermining the correctness of the system. It needs to be stressed, though, that this work addresses only single processor systems and, therefore, the protocols examined in this chapter are strictly single processor protocols.

First and foremost, this requirement poses the need for expressing resource usage eligibility in a way which is independent of execution eligibility.

Ideally, the framework should be able to use one metric, shared and understood by all the different schedulers, to adjudicate requests for access to resources. Secondly, as with all cases of resource sharing, it presents the problems of *deadlock* and *unbounded eligibility inversion* (a special case of which is unbounded *priority* inversion). These terms are well-defined in the literature but for completeness we give their definitions below.

> *"A deadlock is a situation where two or more processes are unable to proceed because each is waiting for one of the others to unlock a resource"*

> *"Eligibility inversion is the situation where a lower-eligibility task executes while some ready higher-eligibility task waits"*

> *"Unbounded eligibility inversion is the scenario where a low eligibility task holds a shared resource that is required by a high eligibility task, causing the execution of the high task to be blocked[2] until the low task has released the resource, all the while allowing other medium eligibility tasks to preempt it"*

Eligibility inversion is unavoidable once we allow sharing of resources between tasks. Imposing precedence constraints on tasks could produce schedules without eligibility inversion in certain systems (essentially static, off-line schedules), but in the general case, there is no guarantee that the higher eligibility task will always request a shared resource before lower eligibility tasks do. However, real-time systems have to bound the extent of eligibility inversion. Finally, in the interest of the framework being as widely applicable as possible, the resource sharing protocol should be able to retain its correctness when tasks self-suspend with the potential of holding resources.

There exist several protocols that help avoid eligibility inversion and deadlock situations. All of them impose new rules on the scheduling of tasks in light of resource sharing amongst them. However, most of them are experimental protocols with limited applicability, e.g. [Kim and Koh 1995], [Squadrito et al. 1999]. They are meant to address specific needs of a certain class of applications, e.g. soft real-time systems (e.g. [Lipari et al. 2004]), multimedia systems (e.g. [deNiz et al. 2001]), real-time data bases (e.g. [Huang et al. 1992], [Squadrito et al. 1996]), real-time distributed systems (e.g. [Sun and

---

[2] By *blocked* we refer to a task that has been rendered unschedulable due to reasons outside its control, e.g. a resource it wants is already locked. This term is similar to *suspended*. However, we define suspension to be the result of either external factors or of the task itself, i.e. self-suspension.

Liu 1996], [Zhang and Cordes 2002]), and so forth. Moreover, most of them are versions of more established protocols, e.g. [Lam and Ng 2000], [Real and Wellings 1999].

Our design criteria, which have also been laid out in the Introduction, dictate that the framework should utilise well-known and widely used protocols. This is important on more than one level:

- The better-known the protocol, the easier it is going to be to implement it.
- The more general the protocol, the broader the spectrum of scheduling policies it will be able to handle.
- The more widely the protocol is used, the more likely it is going to be that programmers will be familiar with it, and by extension, adopt the framework.

As a result of these criteria, there is also no intent to create a new protocol. On the contrary, our focus is placed on the well-established resource sharing protocols that form the basis for most other protocols. Indeed, one can think of these as archetypes for other protocols. They are:

- Basic Priority Inheritance (PI) [Sha et al. 1990]
- Basic Priority Ceiling Protocol (PCP) [Sha et al. 1990]
- Immediate Priority Ceiling Protocol (IPCP) [Burns and Wellings 2001], also known as Ceiling Priority Protocol [Liu 2000]
- Stack Resource Policy (SRP) [Baker 1991], also known as Stack-Based Preemption-Ceiling Protocol [Liu 2000]

There are also three major variations of the PCP and SRP:

- Stack-Based Priority-Ceiling Protocol (SPCP) [Liu 2000]
- Minimal Stack Resource Policy (MSRP) [Baker 1991]
- Basic Preemption-Ceiling Protocol (BPreCP) [Liu 2000]

Finally, a protocol of significant relevance to our work is the Preemption Level Protocol (PLP) [Burns et al. 2004], which is part of the new Ada standard [Ada-Europe 2007]. Its purpose is to incorporate EDF scheduling in the Ada programming language by using the SRP. This is particularly

interesting, since Ada only specifies a fixed-priority preemptive scheduler. Therefore, PLP allows resources to be shared between the priority scheduler and a second-level scheduling policy.

If possible, our aim is to adopt one of these protocols in our framework, with the smallest possible amount of changes. The following sections examine the protocols in more detail. Section 3.1 briefly discusses the priority-based protocols (PI, PCP, IPCP, SPCP). Section 3.2 presents in detail the preemption-ceiling protocols (SRP, BPreCP, MSRP), their mechanism and relative merits. Section 3.3 examines the relation of priority-ceiling and preemption-ceiling protocols and shows the cases where the former are a special case of the latter. Section 3.4 presents the preemption level protocol (PLP). Finally, Section 3.5 provides a summary of the properties of each protocol and makes some concluding remarks justifying our choice.

3.1 Priority-based resource sharing protocols

As is evident by their names, PI, PCP, IPCP and SPCP all rely on the notion of priority. However, the meaning of priority implied is not that of abstract eligibility. Rather, it refers to a particular task parameter that is used in fixed-priority scheduling and whose *base* value remains unchanged by the system throughout the execution of a task (hence the term fixed-priority). Each one of these protocols affects a task's *active* priority[3], which at first is equal to the base priority and changes when accessing resources. In other words, the protocol mechanisms are tailored for tasks that are scheduled according to the particular metric of fixed priority. Tasks of the same priority are scheduled on a FIFO basis.

Priority-based protocols, being linked to a particular scheduling policy, are unsuitable for use in our framework. Nevertheless, they will be discussed, since they provide insight into the workings of preemption-ceiling protocols.

3.1.1 Basic priority inheritance (PI)

The basic priority inheritance protocol is most commonly known as just priority inheritance. The protocol specifies that

---

[3] When referring to the priority of a task, we will always mean its active priority, unless otherwise noted.

- *the priority of a task accessing a resource is elevated to the priority of the highest task waiting for that resource.*

The positive aspects of the protocol are that it bounds priority inversion and it does not need to know task resource demands in advance. However, of all the priority protocols, priority inheritance is the only one susceptible to deadlock and, additionally, its bound on priority inversion is not the smallest possible.

Although the initial protocol refers to fixed-priority scheduling, it has been translated to other scheduling policies, such as EDF [Stankovic et al. 1998]. In fact, the protocol can easily be applied to every scheduling policy with job-level static eligibilities[4]. That is, policies for which a task's eligibility stays unchanged for the duration of the task's release. Indeed, we can say that priority inheritance is a particular application of the more general *eligibility inheritance protocol*. So, for example, with EDF scheduling the protocol would be an "absolute deadline inheritance protocol".

The fact that the protocol does not avoid deadlocks and needs different versions for use with different policies makes it unsuitable for our framework. Nevertheless, it can be a valuable tool in certain situations and for certain classes of systems, as in the case of the Mars Rover detailed in [Jones 1997]. A testimony to that is the fact that PI is available in most real-time operating systems and languages like POSIX [IEEE 2004], Ada [Ada-Europe 2007], and the RTSJ [Bollella et al. 2000], [Belliardi et al. 2006]. More importantly, as we will see in subsequent sections, eligibility inheritance (or priority inheritance, in the case of priority-based protocols) is an integral part of other protocols.

3.1.2 Priority-ceiling protocols

These are protocols that are also relevant to fixed-priority scheduling. The PCP protocol uses the notion of *priority ceiling* introduced in [Sha et al. 1990]. For all priority-ceiling protocols, the priority ceiling of a resource is the highest base priority amongst the tasks that may use it. This has the implication that, in contrast to PI, task resource needs must be known before its release. Also

[4] When we want to refer to the metric for task eligibility used by an arbitrary scheduling policy, we will use the term "execution eligibility", or simply "eligibility".

in [Sha et al. 1990] the notion of a *system ceiling*[5] is introduced, which is the highest ceiling amongst the locked resources. Based on [Sha et al. 1990] the PCP can be defined as follows:

> **Priority Ceiling Protocol (PCP):** *A task is granted a request for a resource only if i) the resource is free and ii) the active priority of the task is greater than the system ceiling, or it is equal to the system ceiling and the task is locking the resource whose ceiling is equal to the system ceiling. Otherwise, the task is said to be blocked on the system ceiling by the task that is locking the resource that set the system ceiling. When a task blocks another task in this manner, it inherits the priority of the blocked task.*

The SPCP is presented in [Liu 2000] and can be defined as follows:

> **Stack-based Priority Ceiling Protocol (SPCP):** *A task can start execution only if its active priority is greater than the system ceiling. Otherwise, the task is said to be blocked on the system ceiling by the task that is locking the resource that set the system ceiling.*

The PCP and SPCP are similar in that they both make use of the system ceiling. They elevate the system ceiling to the highest ceiling amongst the locked resources. Their difference is that, if a task does not have a higher priority than the system ceiling, the SPCP blocks it upon its release, while the PCP blocks the task upon its request of a resource. As will be shown in Section 3.3, this helps SPCP produce a schedule that allows tasks to share a run-time stack. A second difference of PCP and SPCP is that the former applies priority inheritance to a task holding a resource when a higher priority task blocks upon requesting the same resource. The SPCP does not need to apply priority inheritance, as will be seen in Section 3.3.

The IPCP, on the other hand, does not use a system ceiling. It is defined as follows [Liu 2000]:

> **Immediate Priority Ceiling Protocol (IPCP):** *Every task executes at its base priority when it does not lock a resource. The priority of each task locking a resource is equal to the highest priority ceiling of all the resources locked by the task.*

---

[5] Sha et al. [1990] do not use the term "system ceiling", but rather they refer to the semaphore with the highest priority ceiling of all the semaphores currently locked. The system ceiling, as defined here, is a simplification of that, found in [Liu 2000].

Even though the notion of a system ceiling is absent from the IPCP, it produces the same schedule as SPCP through a different technique, which is to elevate the priority of the locking task to the ceiling of the locked resource as soon as the resource is locked.

Priority-ceiling protocols are powerful programming tools that have been extensively researched. Several variations of the PCP protocol exist, some examples being [Squadrito et al. 1999], [Jiang and Cheng 2001], and [Chen et al. 2004]. Versions have also been devised for multiprocessor systems, e.g. [Rajkumar et al. 1988], [Rajkumar 1991], [Chen and Tripathi 1994], [Gai et al. 2001b]. IPCP, together with priority inheritance, is widely used as the priority-ceiling protocol of choice in real-time operating systems and languages, e.g. POSIX [IEEE 2004], Ada [Ada-Europe 2007] and the RTSJ [Bollella et al. 2000], [Belliardi et al. 2006][6]. Their advantages and shortcomings are well documented and for an entry-level discussion on the relative merits of PI and IPCP one can look at [Yodaiken 2002] and [Locke 2002].

Despite their extensive use and applicability, however, they still present us with one major drawback, which, as in the case of priority inheritance, is their application to only fixed-priority scheduling. There have been attempts to use BPI and PCP with deadlines for the Earliest Deadline First (EDF) scheduling algorithm [Chen and Lin 1990], [Spuri and Stankovic 1994], [Stankovic et al. 1998], but they can not be generalized to other scheduling algorithms. Therefore, these protocols are also unsuitable for achieving safe resource usage across policies, for we must be able to express resource usage eligibility in a way which is independent of any particular scheduling algorithm.

Priority-ceiling protocols will be discussed in more detail in Section 3.3, where their mechanism is analysed and their relation to the preemption-ceiling protocols displayed.

3.2 Preemption-ceiling protocols

The key characteristic of the SRP, BPreCP and MSRP protocols, and one that has particular bearing on our work, is the fact that they are specifically designed to detach adjudication of resource sharing from any kind of eligibility

[6] In fact, the RTSJ uses a version of IPCP that also enforces priority inheritance.

metric. These protocols assume preemptive dispatching based on any type of eligibility metric[7]. Thus, they are independent of any particular scheduling policy. Their mechanism depends on a new task parameter called *preemption level*, which was defined in [Baker 1991]. Below is a definition based on Baker's paper.

Definition 3.1: *The* preemption level $\pi(\tau)$ *of a task* τ *is a static parameter that specifies the resource usage eligibility of* τ.

Of the three, SRP and MSRP were introduced first and explained in [Baker 1991]. BPreCP is presented in [Liu 2000]. The three protocols are similar, and later on, in Sections 3.2.3 and 3.2.4, we are going to demonstrate their differences and explain their relative merits.

Baker [1991] explains that the SRP is an extension of PCP. In Section 3.3 we elaborate on this notion of extension. Baker explains that the PCP is extended in three ways that can be applied independently or together:

1. It introduces *multi-unit resources*. Effectively, it treats the ceiling of a resource as a function of the number of free units currently available. This leads to dynamic resource ceilings.
2. It *separates* the execution eligibility of a task from its resource usage eligibility, the latter being expressed through the *preemption level* parameter.
3. It allows *sharing of runtime stacks*.

Each task has its own static preemption level. Preemption levels are separate from eligibilities (which can be static or dynamic), and are assigned according to the following rule, which has been taken from [Baker 1991] and been slightly modified for clarity.

Rule 3.1: *If a task* τ_1 *has higher execution eligibility than another task* τ_2 *even if released later than* τ_2 *then it must also have a higher preemption level than* τ_2

[7] Chapter 5 will more fully address the range of different scheduling policies compatible with the notion of preemption levels. In this chapter we will assume that any type of eligibility metric is compatible.

In other words, *in situations where no locking takes place, preemption levels concur with execution eligibilities as to which task should run next.* The above allocation rule is meant to provide an optimum preemption level assignment, so that tasks will suffer the minimum possible execution eligibility inversion [Baker 1991]. To understand the rule better, let us assume the lower eligibility task τ_1 is released at t_1. If we take the higher eligibility task τ_2 released at $t_2 > t_1$, such that $t_2 - t_1 \approx 0$, then effectively we can consider that, in the extreme case, τ_2 is released simultaneously with τ_1 ($t_1 = t_2$). Also without loss of generality, we can take $t_1 = 0$. Given that $p(\tau_1, t_1) = p(\tau_1, t_2) < p(\tau_2, t_1) = p(\tau_2, t_2)$, the rule specifies that $\pi(\tau_1) < \pi(\tau_2)$. In other words, *if two tasks were to be both released at time t=0, the one with the higher execution eligibility should also have a higher preemption level.* As an example of such an assignment, Baker explains that in the case of EDF preemption levels should be assigned monotonically according to relative deadlines. That is, a task with a shorter relative deadline has greater preemption level than a task with a longer relative deadline. We can understand this if we consider two EDF tasks being released at $t=0$. If at t task τ_2 has a shorter relative deadline than τ_1, then it also has a closer absolute deadline, i.e. greater eligibility. Therefore, its preemption level should be greater.

Since the SRP is an extension of PCP it follows that resources have ceilings. These, however, are based on preemption levels. In all three protocols, resource ceilings for multi-unit resources are computed according to the following rule:

> **Definition 3.2.a**: *The* ceiling of a multi-unit resource *is set to the maximum preemption level amongst the tasks that can block on the resource, given its current number of free units.* [Baker 1991]

We can see that the above definition leads to dynamic ceilings that change according to the number of free units a resource has at any particular time. A specific case is single-unit resources, which according to the above definition, have static resource ceilings that are set in a way analogous to PCP, albeit using preemption levels instead of priorities.

> **Definition 3.2.b**: *The ceiling* $\lceil r \rceil$ *of a single-unit resource* r *is set to the maximum preemption level amongst the tasks that use it.* [Baker 1991]

All three protocols can work with both multi-unit and single-unit resources, because, as we will see, the definition of the resource ceiling is orthogonal to the protocol mechanism. In other words, as long as a resource has a ceiling value at every point in time, the protocols are not affected by the way the value was assigned. To understand this let us consider any resource r at any time t, and any task τ_b whose request of r would block at time t. What the preemption-ceiling protocols expect from the definition of the ceiling of resource r, is the inequality $\pi(\tau_b) \leq \lceil r \rceil$. Conversely, if τ_g is any task whose request of r at time t would be granted, the protocols expect $\pi(\tau_g) > \lceil r \rceil$ to hold. Both Definitions 3.2.a and 3.2.b satisfy this.

The SRP and BPreCP protocols will be the focus of our examination of preemption-ceiling protocols, since they constitute the two main approaches to the use of preemption levels. MSRP is merely an extension of the SRP and will be briefly presented later in this section.

3.2.1 Stack Resource Policy (SRP)

Based on the definitions for preemption levels and resource ceilings [Baker 1991] describes the SRP with the following rule.

> **Stack Resource Policy (SRP):** *A task can start execution only if it is the oldest, highest execution eligibility, pending request, and its preemption level is higher than the ceiling of each locked resource.*

According to Baker, the usage of preemption levels in the SRP means that *a task can preempt another task only if it has a higher preemption level.* However, we must not mistake preemption levels as an extra metric for task dispatching. This rule describes a necessary condition for preemption, but not a sufficient one. It should be read as "a released task with lower preemption level can definitely not preempt the running task". This is more of an ascertainment stemming from the preemption level assignment rule, rather than a rule in itself. Indeed, if preemption levels are assigned according to Rule 3.1, a lower preemption level at the point of release automatically means that the released task has lower eligibility than the running task.

To make the SRP rule easier to apply, the notion of a *system ceiling* is introduced [Baker 1991]:

Definition 3.3: *The* system ceiling $\overline{\pi}$ *is the highest ceiling amongst the locked resources:* $\overline{\pi} = \max\{\lceil r \rceil \mid r = locked\}$

As we can see, the execution eligibility rule is made up of two parts, which are:

SRP Eligibility Test (SRP_ET): *A task can start execution only if it is the oldest, highest execution eligibility, pending request.*

SRP Preemption Level Test (SRP_PLT): *A task can start execution only if its preemption level is higher than the system ceiling.*

The eligibility test represents a dispatching point. Two observations can be made on the SRP_ET. First, the fact that the SRP_ET describes only one such dispatching point, namely a task's release, means that the SRP considers tasks to be non-suspending. Secondly, we can see that what the SRP_ET implies is that the eligibility of a newly released task τ_n must be compared to the eligibility of all the tasks previously released. What is important here is the meaning of "pending". A pending request is a released task that, for whatever reason, is still waiting to be granted the CPU. It might be that its execution eligibility was not high enough to preempt the running task, or that it was blocked by the SRP_PLT due to mutual exclusion. Therefore, in order for task τ_n to start execution, it must have higher execution eligibility than all the tasks that were released before it and are currently pending. At first one might think that any task τ_p that is currently pending must have lower execution eligibility than the running task τ_r. However, it is clear that the SRP rule can fail either because of the execution test or because of the preemption level test. Therefore, it is perfectly feasible that the pending task τ_p has greater execution eligibility than the running task τ_r, but failed the preemption level test due to resource sharing. Hence, to successfully perform the ET part of the SRP, we must keep track of the highest execution eligibility amongst the tasks that have already been released in the system. The test should compare against this eligibility and not that of the running task. And since the eligibility of runnable tasks is, by default, being taken into consideration, it is the eligibility of tasks blocked on the system ceiling that should also be included.

This notion of checking against an execution eligibility different from that of the running task is reminiscent of another protocol we have already seen. It is exactly what the *eligibility inheritance protocol* achieves, as explained in Section

3.1.1. When the running task inherits the eligibility of the highest eligibility pending task, checking against its active priority is the same as checking against the highest eligibility task. In other words, what is implied is that, for the SRP to be correct, the eligibility test should behave in a way analogous to dispatching under the effects of eligibility inheritance. And the easiest way to achieve that is to actually apply the eligibility inheritance rule. In the case of SRP, eligibility inheritance should take place on the running task, when a newly released task blocks due to the system ceiling. However, Baker [1991] does not explicitly specify eligibility inheritance taking place. Nor does he provide any other way of achieving the same result. Therefore, we must assume that eligibility inheritance is an implicit part of the SRP. The inheritance rule can be specified as:

> **SRP Eligibility Inheritance (SRP_EI)**: *When a task blocks due to the system ceiling, the blocking task inherits the highest execution eligibility of all the blocked tasks, if this is greater than its own current eligibility. Upon unlocking, the eligibility of the blocking task returns to the value it had at the time of locking the resource.*

The inherited eligibility is called the *active eligibility* of the blocking task. Having defined the SRP_EI rule it is immediately obvious that its addition does not affect the SRP rule. The reason is that the rule applies to newly released tasks, whereas eligibility inheritance has meaning only for tasks already running and holding resources. [Liu 2000] combines the SRP_ET, SRP_PLT and SRP_EI to present the SRP protocol more formally, but for priority-driven systems. Liu [2000] names this version the Stack-Based Preemption-Ceiling Protocol and it is given below, but translated for any execution eligibility metric.

Stack-Based Preemption-Ceiling Protocol (SBPCP)

1. *Update of the system ceiling*: Whenever all the resources are free, the system ceiling is $\overline{\pi} = \varnothing$. The system ceiling is updated each time a resource is allocated or freed.
2. *Scheduling rule*: After a task is released, it is blocked from starting execution until its preemption level is higher than the current system ceiling. At any time t, tasks that are not blocked are scheduled preemptively according to their assigned eligibilities.
3. *Allocation rule*: Whenever a task τ requests a resource r, it is allocated the resource.

4. *Eligibility inheritance rule*: When some task is blocked from starting, the blocking job inherits the highest eligibility of all blocked tasks. The eligibility of the blocking task drops to its previous value when the reason for blocking the other task is removed.

It is interesting to note here that although Baker [1991] does not explicitly mention an inheritance rule, it is clear from the use of the word "pending" in the eligibility test. It is worth mentioning the following quote from [Baker 1991]:

"Note also that the SRP preemption test (i.e. the SRP_PLT) has the effect of imposing priority inheritance (that is, an executing job that is holding a resource resists preemption as though it inherits the priority of any jobs that might need that resource), though the effect is accomplished without modifying the formal priority of the job, *p(J)*."

Baker attributes the application of eligibility inheritance (to which he refers to as priority inheritance) to the preemption level test, which is to say that by merely applying the test we have the effects of eligibility inheritance. This is not correct. It is true that the SRP rule enforces eligibility inheritance, yet not through the SRP_PLT. The use of the word "pending" in the eligibility test implies the existence of the effects of eligibility inheritance. Whether these effects are achieved by modifying the active eligibility of the task is a matter of implementation. In order to make our point clear we can say that the SRP_ET test needs the enforcement of SRP_EI in order to produce the correct results, but it is the SRP_PLT, when it fails, that triggers the application of SRP_EI. In this sense, eligibility inheritance is a distinct part of the SRP rule, as are SRP_ET and SRP_PLT, however it is implicit.

The SRP has been proven to be deadlock-free [Baker 1991], with the assumption that tasks do not suspend while locking. In Section 3.2.3 we prove that the SPR is not deadlock-free when tasks do suspend while locking. The stack-sharing properties of SRP are important in embedded systems, where memory is limited and precious [Gai 2005], [Hänninen et al. 2006]. In certain cases the SRP allows storage savings of up to 90% [Baker 1991]. In addition, the SRP ensures that, once a task has started executing, it cannot block. Also, as we will see in Section 3.2.4, the SRP allows better response times for the highest eligibility task, in contrast to BPreCP, as can be seen in Figures 3.3.e and 3.3.f. This is because by applying the SRP_PLT rule at their release, the

SRP blocks middle eligibility tasks before they execute. This way it minimises interference on the low eligibility locking task, which in turn means that it will finish its critical section sooner and allow the high task to start its execution sooner.

Finally, the MSRP is briefly described in this section, since it is just an extension of the SRP:

> **Minimal SRP:** *A task can execute only if it is the oldest, highest execution eligibility pending request and its preemption level is either i) higher than the system ceiling or ii) equal to the system ceiling with the presently available resources sufficient for the task to execute to completion without direct blocking.*

The MSRP was designed to solve the problem of unnecessary blocking present in the SRP. It imposes the minimal blocking necessary to prevent deadlocks and multiple eligibility inversion under conditions of stack sharing. However, this has made the scheduling rule more complicated, as noted in [Baker 1991].

3.2.2 Basic Preemption-Ceiling Protocol (BPreCP)

Having examined the SRP, it is easy to see where the BPreCP differs. The disparity is that the BPreCP enforces its own BPreCP_PLT test at a different point in a task's execution, namely at the point of attempting to lock a resource and not at its release. This has implications when an executing task τ performs nested locking of resources, i.e. it is already locking a resource when the BPreCP_PLT is enforced. Locking a resource means that the system ceiling will have already been set to the preemption ceiling of the locked resource and from the Definitions 3.2.a, 3.2.b and 3.3 it is easy to see that $\bar{\pi} \geq \pi(\tau)$. Therefore, locking a second resource in a nested way would be impossible under the SRP_PLT. For this reason, the preemption level test is defined for the BPreCP as follows:

> **BPreCP Preemption Level Test (BPreCP_PLT):** *A task is granted a request for a resource only if its preemption level is higher than the system ceiling, or if it is the task holding the resource whose preemption ceiling is equal to the system ceiling.*

The eligibility inheritance rule for the BPreCP is the same as for the SRP. However, the eligibility test is different, since with BPreCP scheduling decisions can be made during a task's execution and not just at the point of its release. The eligibility test can be written as follows:

> **BPreCP Eligibility Test (BPreCP_ET):** *A task can execute only if it is the oldest, highest active execution eligibility runnable task, provided that execution eligibility inheritance is being applied to a task holding a resource when a higher eligibility task blocks on the same resource.*

Liu [2000] gives a formal definition of the BPreCP protocol, which is given below, again translated for execution eligibilities rather than just priorities.

Basic Preemption-Ceiling Protocol (BPreCP):

1. *Scheduling rule:*
   a. At its release time t, the active eligibility of every task τ is equal to its base eligibility. The task retains this eligibility except under the condition stated in the eligibility inheritance rule.
   b. Every ready task is scheduled preemptively according to its active eligibility.
2. *Allocation rule:* Whenever a task τ requests resource r, one of the following two conditions occurs:
   a. The resource is locked.
      i. The preemption level of τ is not higher than the system ceiling ($\pi(\tau) \leq \bar{\pi}$), the request is denied and τ blocks.
   b. The resource is free.
      i. If the preemption level of τ is higher than the system ceiling ($\pi(\tau) > \bar{\pi}$), or if τ is the task holding the resource whose preemption ceiling is equal to the system ceiling ($\lceil r \rceil = \bar{\pi}$), r is allocated to τ.
      ii. If the preemption level of τ is not higher than the system ceiling ($\pi(\tau) \leq \bar{\pi}$), the request is denied and τ blocks.
3. *Eligibility inheritance rule:* When τ becomes blocked, the task τ_b that blocks τ inherits its active eligibility. τ_b executes with its inherited eligibility until the time when it releases every resource whose preemption ceiling is equal to or higher than the preemption level of τ. At that time the eligibility of τ_b returns to the value it had at the time it was granted the resources.

As can be seen, in Liu's definitions as well, the eligibility inheritance rule is basically the same in BPreCP as in the SRP. In the following section we prove that the BPreCP is deadlock free under any circumstances. This, together with its support for arbitrary scheduling policies, makes it, overall, the best candidate for forming the base of the resource sharing protocol in our framework, which will be presented in the next chapter.

3.2.3 Self-suspending tasks

Task self-suspension can happen through one of many ways, an example being the equivalent of a POSIX `sleep()` call. In general, we can distinguish two cases: i) self-suspension while executing outside critical sections, and ii) self-suspension while executing inside a critical section (locking a resource). In this section we examine the effect of both cases on the preemption-ceiling protocols.

Proposition 3.1: Non-blocking task execution, minimum eligibility inversion and stack sharing all presuppose that tasks will not self-suspend. If the system does allow self-suspension, these conditions are negated.

Proof: Let us suppose task τ_1 suspends itself at time t_s, when the system ceiling is $\bar{\pi}_s$. At t_s another task will have to be selected to run, say τ_2. Suppose $p(\tau_2) < p(\tau_1)$. Also suppose that τ_2 locks a resource with a ceiling higher than the system ceiling ($\lceil r \rceil > \bar{\pi}_s$). Then τ_1 wakes up and preempts τ_2. If τ_1 now attempts to lock r it will block either on its lock, in the case of SRP, or on the system ceiling, in the case of BPreCP. Furthermore, this blocking is additional to any initial blocking from lower eligibility tasks, so eligibility inversion is not minimal. Finally, since we are forced to execute a lower eligibility task after a higher eligibility task has already started its execution, stack sharing under SRP is violated. ■

Proposition 3.2: Under SRP, task self-suspension while locking negates deadlock-free execution.

Proof: To prove this, let us consider two tasks τ_1, τ_2, with $p(\tau_2) < p(\tau_1)$, using two resources r_1, r_2 as shown in the Gantt diagram of Figure 3.1.

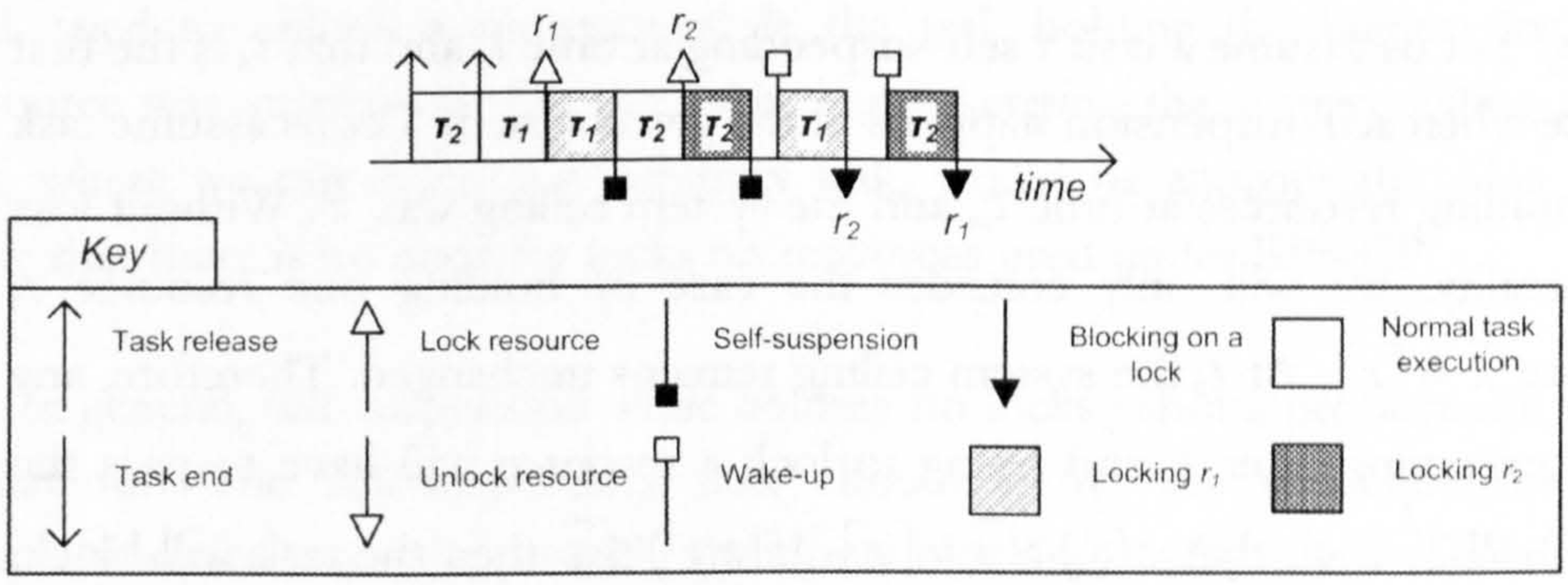


Figure 3.1: Deadlock of self-suspending tasks under SRP

Task τ_2 is released first. Then τ_1 is released and preempts τ_2. It runs and locks r_1. Then it suspends itself and τ_2 is selected to run. τ_2 locks r_2 and then suspends as well. After some time τ_1 wakes up, continuing to hold r_1, and tries to lock r_2. Naturally it cannot and blocks on the lock. After a while τ_2 wakes up and tries to lock r_1. It also blocks on the lock producing a deadlock. ■

This situation is possible because in the SRP the preemption level test takes place once at the point of release. The SRP applies its PLT early (enforcing early task blocking) precisely because it assumes no self-suspension. Once a task starts running there are no more preemption level tests to regulate its use of resources, since it is assumed that everything will be available. If, however, a task is selected to run because some other task suspended while holding a resource, this availability is not a given anymore. This also affects the implementation of shared resources. When locking tasks do not suspend, the SRP is guaranteed to grant every resource request. Therefore, no locks need to be implemented for resources. However, once tasks are allowed to self-suspend, the absence of locks can jeopardise mutual exclusion. This can be seen in Figure 3.1, at the time τ_1 tries to lock r_2. If no lock is present, both τ_1 and τ_2 will be simultaneously using r_2.

In contrast, BPreCP applies its PLT at every attempt to lock a resource. This, together with the fact that the system ceiling changes only upon locking and unlocking, means that it is not possible for a task to lock a resource without having a higher preemption level than the system ceiling, even if some task has previously suspended while holding a lock. This is proved in the following proposition.

Proposition 3.3: Under BPreCP, task self-suspension while locking can not lead to a deadlock.

Proof. Let us assume a task τ self-suspending at time t_s and that t_s is the first instance when self-suspension happens in the given system. Let us assume task τ was holding resources at time t_s, and the system ceiling was $\overline{\pi}$. Without loss of generality, we will only consider the case of holding one resource r, therefore $\overline{\pi} = \lceil r \rceil$. At t_s the system ceiling remains unchanged. Therefore, any task τ' executing after t_s and trying to lock a resource will have to pass the BPreCP_PLT, such that $\pi(\tau') > \overline{\pi} = \lceil r \rceil$. If $\pi(\tau') \leq \overline{\pi}$ then the task will block, preventing a deadlock. If it does pass the test then it follows that it will not need r, since if it did it would be $\lceil r \rceil \geq \pi(\tau')$. Also, if τ wakes up, it will not be able to lock anything since the system ceiling will be $\overline{\pi'} = \lceil r' \rceil \geq \pi(\tau') > \overline{\pi} \geq \pi(\tau)$. Hence, there can again be no deadlock.

A second question is what happens if some resources were already locked by other tasks at the time τ suspended. If this is the case then it follows that the preemption level of τ must be greater than the system ceiling $\overline{\text{`}\pi}$ at the time $\tau$ locked $r$ ($\pi(\tau) > \overline{\text{`}\pi}$) and, therefore, τ could not be using any of the already locked resources. Conversely, if a task `$\tau$ holding such a resource is selected to run, it will not be able to lock another resource because of the system ceiling ($\pi(\text{`}\tau) \leq \overline{\text{`}\pi} < \pi(\tau) \leq \overline{\pi}$). If `τ does not try to lock anything, it will proceed with the execution of its critical section. If τ wakes up before `$\tau$ finishes then the system will be in a state equivalent to just prior to $t_s$. If `τ executes its critical section uninterrupted, it will proceed to unlock the resource. At that point the new system ceiling $\overline{\pi''}$, as stated in Definition 3.3, should be equal to the highest ceiling amongst the locked resources, which means $\overline{\pi''} = \overline{\pi} = \lceil r \rceil$. This leads the system to a state equivalent to the state it had at t_s, but with one additional free resource. Hence, there can again be no deadlock. ■

Corollary 3.1: The BPreCP is deadlock-free under any circumstances.

Proof. Suspension while locking is a stronger condition than non-suspension while locking. Since the protocol is deadlock-free even when suspension is allowed while locking, it will be deadlock free in general. ■

From the above proof of Proposition 3.3 we can see that a key issue in guaranteeing the correctness of BPreCP with self-suspending tasks is to provide an appropriate representation of the system ceiling. For example, modelling the system ceiling as a stack would produce erroneous results, if a

task tried to unlock a resource while the task holding the highest locked resource was suspended. On the other hand, keeping the system ceiling as a list, where we can delete any arbitrary link, would be appropriate. Also, we note that there is no need for locks on resources used under BPreCP.

In general, self-suspension while holding no locks is not a problematic task behaviour. The self-suspending task should know the consequences of suspension, for example, possible additional blocking, and use self-suspension at its own risk. Self-suspension outside of critical sections can only affect the suspending task. As far as the system is concerned, it can consider suspension as the end of the task's execution. When the task wakes up the system can consider that to be the release of a new task with shorter relative deadline, shorter period, and shorter execution cost than the initial task. Therefore, the protocol will handle it as any other task release.

However, the case of suspending while locking a resource can present serious problems to certain resource sharing protocols, as with the case of SRP. The approach different real-time languages take towards this problem varies. For example, Ada does not allow suspension while locking a resource [Ada-Europe 2007], while the RTSJ allows it [Bollella et al. 2000]. As it has already been stated, in the interest of applicability our scheduling framework takes the more general approach of allowing task self-suspension within a critical section.

3.2.4 The role of the PLT and EI

The preemption level test and eligibility inheritance rule each fulfill a particular function, which is the same in both BPreCP and SRP. When combined, they achieve deadlock-free, maximally bounded execution eligibility inversion control.

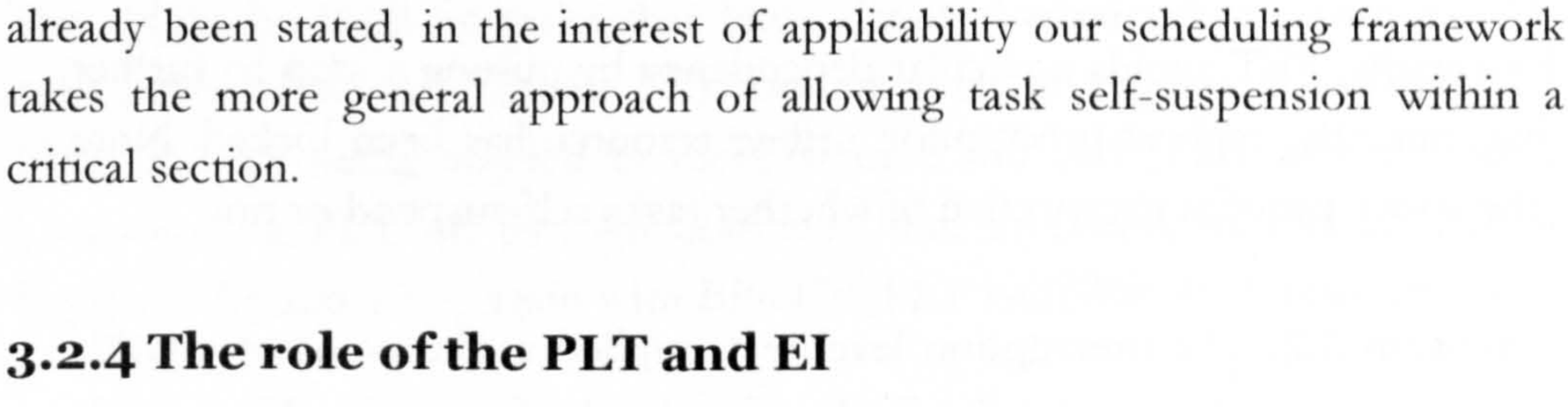


Figure 3.2: A circular dependency of *n* tasks

Theorem 3.1: The preemption level test, applied when requesting a resource, is sufficient to prevent deadlocks.

Proof: To describe a deadlock situation in a general way, let us suppose we have $n \in \mathrm{N}$ tasks $\{\tau_1,\ldots,\tau_n\}$, with $\pi(\tau_1) < \ldots < \pi(\tau_{n-1}) < \pi(\tau_n)$, that use n single-unit resources $\{r_1,\ldots,r_n\}$ in a circular fashion. This is shown in Figure 3.2, where $g,i,j,k \in [1,n]$. The use of single-unit resources is preferred for simplicity and does not imply loss of generality. Each task τ_i uses exactly two resources, r_i and r_j. Each resource r_i can be used by exactly two tasks, τ_i and τ_g. Therefore, there will be two resources that both have the highest preemption level as ceiling, the first being r_n and the second being the second resource that task τ_n uses, which we name r_k. In order to have a circular wait condition, every resource must be locked. However, once either r_n or r_k is locked by the running task τ_r, any other task trying to lock a resource will fail the preemption level test, since it cannot have a higher preemption level than the system ceiling. Therefore, the only task that will be able to lock a resource is τ_r. Since τ_r was able to lock either r_n or r_k, it follows that the second of the two resources is also unlocked, so the task can proceed with locking that as well. Therefore, the running task can finish its execution, thereby eliminating the possibility of a circular wait. Hence there can be no deadlock. ■

Essentially, PLT avoids a circular dependency by putting a stop to further locking, once the highest preemption ceiling resource has been locked. Note that the above proof is irrespective of whether tasks self-suspend or not.

Theorem 3.2: The preemption level test, applied at the point of a task's release, is sufficient to prevent deadlocks, if and only if no task self-suspends while holding a resource.

Proof: The "*if*" part of the theorem follows from Proposition 3.2. To prove the "*only if*" part let us suppose that there is a deadlock while PLT is applied at the release of a task and there is no self-suspension while holding resources. To have a deadlock a circular wait must exist, as in Figure 3.2, where all resources have been locked and a task τ_i, having locked r_i, is requesting r_j.

Since self-suspension is not allowed, task τ_i could only have started its execution by preempting at the point of its release the previously running task. Also, since all resources must be locked, τ_i must be the one locking the resource with the greatest preemption level r_n, i.e. $r_i \equiv r_n$ and $r_j \equiv r_k$. If it was not, then the task locking it would have run before τ_i and would have set

the system ceiling to $\pi(\tau_i) \leq \overline{\pi} = \lceil r_n \rceil$, thus blocking τ_i at the point of its release. This means that the system ceiling when τ_i was released was $\overline{\pi} < \pi(\tau_i)$ and that the current ceiling is $\overline{\pi} = \lceil r_n \rceil$. Also because τ_i will not self-suspend, only tasks with greater preemption level and execution eligibility will be able to preempt it. However, these tasks do not affect the circular dependency. So, for task τ_i to cause a deadlock, it must block on the second resource r_k whose ceiling is equal to r_n. However, if r_k is already locked, τ_i would have blocked when it was released – a contradiction. ■

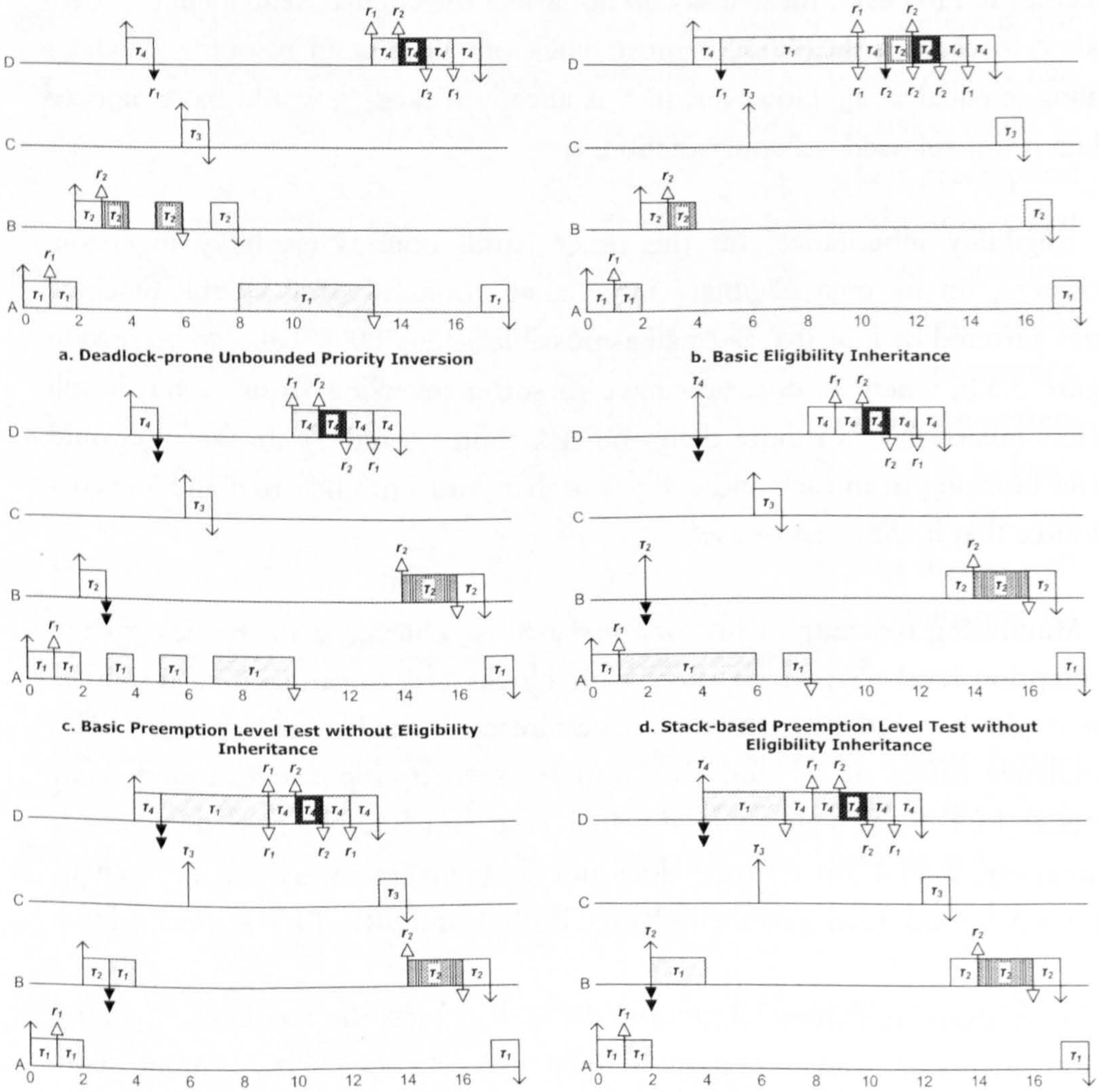
Eligibility inheritance, on the other hand, bounds eligibility inversion. However, on its own eligibility inheritance "does not reduce the blocking times suffered by jobs (to) as small as possible" [Liu 2000]. This can be seen in Figure 3.3.b, where τ_4 does not have to suffer inversion from τ_3, but it still suffers inversion from more than one task (both τ_1 and τ_2). In fact, τ_4 would suffer blocking from each and every task that got to run before it and locked a resource that it will need to use.

Minimising blocking to just one instance is achieved with the help of the preemption level test, which specifically blocks tasks from locking resources that might be used by tasks that have already locked other resources. The cumulative effect of EI and PLT can be seen in Figures 3.3.e and 3.3.f, BPreCP_PLT being applied in the first case and SRP_PLT in the second. Conversely, a PLT on its own does not limit inversion, as can be seen in Figures 3.3.c and 3.3.d, again with BPreCP_PLT and SRP_PLT respectively.

Let us examine Figure 3.3 in more detail. It depicts the classic paradigm of priority inversion with 4 non-suspending tasks τ_1, τ_2, τ_3, τ_4, running with eligibilities A, B, C and D, respectively. The eligibility e, release time t_r and execution cost c of each task are given in Table 3.1 below.

Table 3.1: Task release times and execution costs

	τ_1	τ_2	τ_3	τ_4
e	*A*	*B*	*C*	*D*
t_r	*0*	*2*	*6*	*4*
c	*8*	*4*	*1*	*5*



a. Deadlock-prone Unbounded Priority Inversion

b. Basic Eligibility Inheritance

c. Basic Preemption Level Test without Eligibility Inheritance

d. Stack-based Preemption Level Test without Eligibility Inheritance

e. Basic Preemption Level Test with Eligibility Inheritance

f. Stack-based Preemption Level Test with Eligibility Inheritance

Figure 3.3: The role of eligibility inheritance and the preemption level test in resource sharing protocols

Execution eligibility A is the lowest and D the highest of the four. We think of these as the eligibilities for the current job of each task, which, in the general case, can be task-level dynamic eligibilities, i.e. they could change between two different jobs of a task (Section 5.3.1 will examine the case of job-level dynamic eligibility algorithms). Irrespective of their current eligibilities, the task preemption levels are $\pi(\tau_1) < \pi(\tau_2) < \pi(\tau_4) < \pi(\tau_3)$. Tasks τ_1 and τ_4 use resource r_1, therefore $\lceil r_1 \rceil = \pi(\tau_4)$. Tasks τ_2 and τ_4 use resource r_2, therefore $\lceil r_2 \rceil = \pi(\tau_4)$. Both resources are single-unit.

First, Figure 3.3.a shows what happens when there is *no resource sharing protocol applied.* Task τ_1 is released and locks r_1. Then it is preempted by τ_2, which locks r_2. At time 4 task τ_4 is released, preempts τ_2 and tries to lock r_1 but blocks on its lock. Task τ_2 continues the execution of its critical section and at time 6 it unlocks r_2. At the same time it is preempted by task τ_3. Both the execution of τ_2 and of τ_3 constitute unbounded priority inversion, that delays the time τ_1 releases r_1 and, consequently, the time at which τ_4 will finish its execution. Furthermore, deadlocks are possible. For example, if at time 5 τ_2 had requested r_1, it would have blocked on its lock. Then, if at a later time τ_1 had requested r_2, deadlock would have occurred.

Figure 3.3.b illustrates the case where we enforce the eligibility inheritance rule. So, again, task τ_1 is released first and locks r_1. τ_2 preempts it at time $t=2$ and proceeds to lock r_2. At $t=4$ task τ_4 is released, but when it tries to lock r_1 it blocks on its lock and the eligibility of τ_1 is raised to D. τ_1 now executes at D and τ_3, released at $t=6$, cannot preempt it. At $t=10$ τ_1 releases r_1 and τ_4 promptly locks it. Then at $t=11$ it tries to lock r_2 but blocks on its lock. This immediately raises the eligibility of τ_2 to D. τ_2 executes at D and then unlocks r_2, which is immediately locked by τ_4 in a nested way. τ_4 proceeds to unlock both resources and finishes its execution. Only then is τ_3 able to run. Finally, all tasks end one by one. Here we can see that eligibility inheritance was able to prevent one case of inversion, at time $t=6$ when τ_3 was unable to preempt τ_1. However, τ_4 still suffers multiple eligibility inversion, since it is blocked by both τ_1, at $t=5$, and τ_2, at $t=11$. Also, the possibility of deadlock is not removed. We can understand this if we consider the case where τ_2 requests r_1 at, say, $t=3.5$ and τ_1 requests r_2 at $t=3.7$. The two tasks would then be caught up in a deadlock.

Figure 3.3.c shows the application of the preemption level test at the point of requesting a resource, as in the BPreCP. However, the eligibility inheritance rule is not applied here, in order to observe the role of just preemption levels. Again task τ_1 is released and locks r_1, this time setting the system ceiling to $\bar{\pi} = \lceil r_1 \rceil = \pi(\tau_4)$. τ_2 is then released but when it tries to lock r_2 it triggers the preemption level test, which it fails since $\pi(\tau_2) < \bar{\pi}$. Notice that τ_2 fails the test even though it tries to lock a free resource. Therefore τ_2 is blocked on the system ceiling and τ_1 continues to run. At $t=4$ it is preempted by τ_4, which tries to lock r_1 and also fails ($\pi(\tau_4) \leq \bar{\pi}$). τ_1 again continues to execute. However, at $t=6$ it is preempted by τ_3, since $\pi(\tau_3) > \bar{\pi}$. The fact that τ_3 has a greater preemption level but lower eligibility than τ_4 should not be viewed as a contradiction. For example, if the scheduling policy was EDF, then it might be the case that τ_3 has a shorter relative deadline than τ_4, but in this particular instance has a greater absolute deadline than τ_4. As a result, we see that the preemption level test on its own, when applied at the point of accessing a resource, cannot prevent unbounded eligibility inversion. However, it is sufficient to prevent deadlock, as shown by Theorem 3.1.

Figure 3.3.d illustrates the case where we enforce the preemption level test at the point of a task's release, as in the SRP, again without EI. Task τ_1 is again released and locks the resource, setting the system ceiling to $\bar{\pi} = \lceil r_1 \rceil$. This time, when τ_2 and τ_4 are released, the preemption level test is enforced and the tasks are blocked, since $\pi(\tau_2) < \pi(\tau_4) \leq \bar{\pi}$. However, τ_3 is again able to preempt τ_1, causing unbounded eligibility inversion as in the previous case. Again, as per Theorem 3.1, no deadlock is possible.

In Figure 3.3.e we produce the BPreCP, by combining EI and PLT at the point of locking. We again have the situation of τ_1 locking r_1, while τ_2 and τ_4 block due to the preemption level test. However, because of eligibility inheritance, task τ_1 inherits the eligibility first of τ_2 and then of τ_4, when they block on the system ceiling. This way task τ_3 cannot preempt τ_1 when it is released. Hence, priority inversion is limited to only one block from a lower eligibility task.

Finally, in Figure 3.3.f we enforce the SRP, by combining EI and PLT the instance a task is released. τ_1 is released and locks r_1, τ_2 and τ_4 block due to the preemption level test, and τ_3 cannot preempt τ_1 when it is released. So, again, priority inversion is limited to only one block from a lower eligibility task. The

difference here is that τ_2 and τ_4 do not get to execute at all before τ_1 releases r_1. Therefore, contexts can be loaded on a stack, with a context being unloaded off the stack only at the end of a task's job. This also has the effect of a better response time for the highest eligibility task, compared to BPreCP.

To give a more complete picture of the effects of each protocol, the following Table 3.2 summarises the response time for each task under each protocol.

Table 3.2: Response times for tasks of Table 3.1

	τ_1	τ_2	τ_3	τ_4
No protocol	18	6	1	13
EI	18	15	10	11
BPreCP_PLT without EI	18	15	1	10
SRP_PLT without EI	18	15	1	9
BPreCP_PLT with EI	18	15	8	9
SRP_PLT with EI	18	15	7	8

We will now give a more formal definition of the roles of EI and PLT. Given a task τ locking a resource r we will define three sets of tasks:

- The set $\tilde{T}$ of tasks with eligibility higher than that of the locking task τ; e.g. $\{\tau_2, \tau_3, \tau_4\}$ in Figure 3.3.
- The subset $\hat{T}$ of $\tilde{T}$ containing those tasks whose preemption level is lower or equal to the system ceiling, not including the locking task; e.g. $\{\tau_2, \tau_4\}$ in Figure 3.3.
- The subset $\breve{T}$ of $\tilde{T}$ containing the tasks with preemption level higher than the system ceiling, but lower eligibility than the highest eligibility task that uses the latest locked resource; e.g. $\{\tau_3\}$ in Figure 3.3.

When enforced, the preemption level test blocks set $\hat{T}$, while the eligibility inheritance rule blocks $\breve{T}$. With the above example we can see the dual role of preemption levels: on the one hand they enforce mutual exclusion, and on the other they help limit eligibility inversion down to one instance.

3.3 The relation between preemption-ceiling and priority-ceiling protocols

In Section 3.2 we explained how preemption levels are assigned and how they are independent of any particular policy. Rule 3.1 is a guide for optimum preemption level assignment and we showed how it translates in the case of EDF. Since application of preemption levels is not restricted by the policy, we could also apply the rule to fixed-priority scheduling. In this special case preemption levels equal task priorities. This way mutual exclusion and bounded priority inversion is achieved using only one metric, priority. Yet, if we do this, we see that the schedule produced by BPreCP is exactly the same with that produced by PCP. This means that PCP is an instance of BPreCP. Or, in other words, PCP is the incarnation of BPreCP for the special case of fixed-priority scheduling. So, in the example of Figure 3.3, if we take A,B,C,D to be priority levels, we can see that enforcing PCP produces the same schedule as Figure 3.3.e.

We have mentioned that PCP is to SPCP what BPreCP is to SRP. Therefore, since PCP produces the same schedule as BPreCP when preemption levels are based on priorities, the same is true for SPCP with regard to SRP, but only in the sense that both protocols produce schedules where the same tasks run at the same moments in time. The SPCP protocol is an incarnation of SRP for fixed-priority scheduling, but, according to the protocol's definition in [Liu 2000], it differs in one aspect: it does not apply an eligibility (priority) inheritance (EI) rule. The reason for this is that ceilings are based on eligibility, i.e. priority. The SPCP takes advantage of that, realising that there cannot be such discrepancy as in the case of the SRP, where a task can have a higher preemption level than the system ceiling and yet lower eligibility than the highest locking task. So, by setting the system ceiling to the priority of the highest locking task the protocol is certain that any released task failing the SRP_PLT test will also fail the EI test. Therefore, there is no need for eligibility inheritance (EI). In the case of the SPCP, the SRP_PLT incorporates the EI test.

Another way to look at this is by considering the task sets defined in the previous section. In the case of SPCP it is $\breve{T} = \varnothing$. Therefore, in this case only tasks that belong to $\hat{T}$ must be blocked, and SRP_PLT is sufficient for that. However, in the case of PCP, where we do not apply the PLT as soon as a task is released, any task belonging to $\hat{T}$ will have the chance to preempt the

locking task. To bound this preemption we have to apply the priority inheritance rule, which wouldn't have been needed if PLT was applied at the point of a task's release.

The schedule produced by the SPCP can be seen in Figure 3.4.

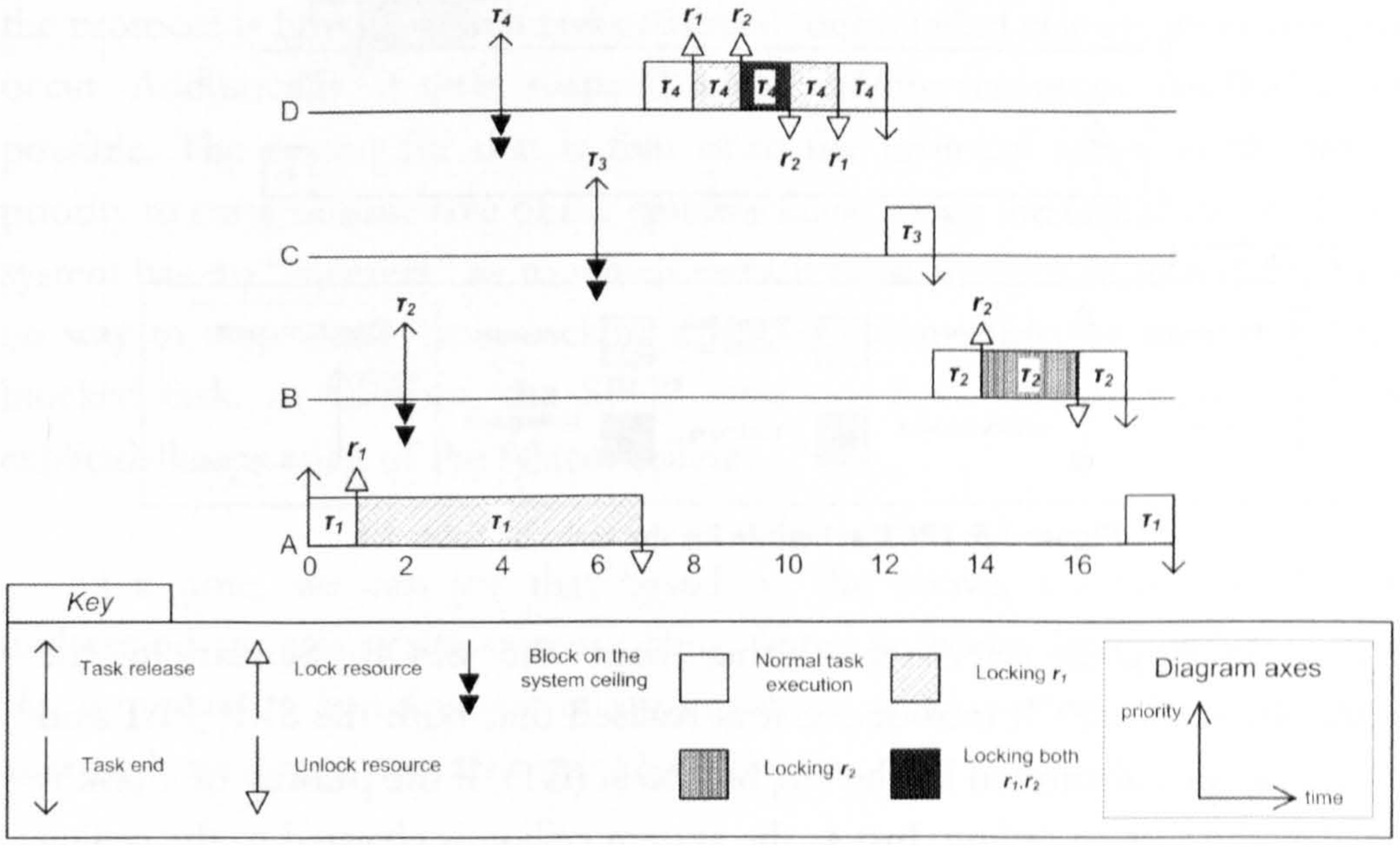


Figure 3.4: SPCP schedule for the tasks of Table 3.1

The drawback of not explicitly applying the EI rule, however, is that released tasks, that are going to be blocked by the PLT test, still have to perform the test. This, presumably, introduces some overhead. Applying EI would mean that, when a task blocks, no other task with equal or lower priority than the blocked task will need to perform the PLT test. Therefore, the issue is one of optimisation. By applying the EI test, the SPCP becomes the exact translation of SRP for fixed-priority scheduling, and would look like Figure 3.3.f.

The last priority-ceiling protocol is IPCP. Taking the same example, the IPCP produces the same schedule as the SPCP, but in its own unique way. This can be seen in Figure 3.5. The ceilings of resources are equal to the maximum priority amongst the tasks that use them, so $\lceil r_1 \rceil = \lceil r_2 \rceil = p(\tau_4)$. τ_1 is released and as soon as it locks r_1 its priority is elevated to the resource ceiling. Hence, all tasks τ_i with priorities $p(\tau_i) \leq p(\tau_4)$ are prevented from running upon their release, until $t=7$ when τ_1 releases the resource.

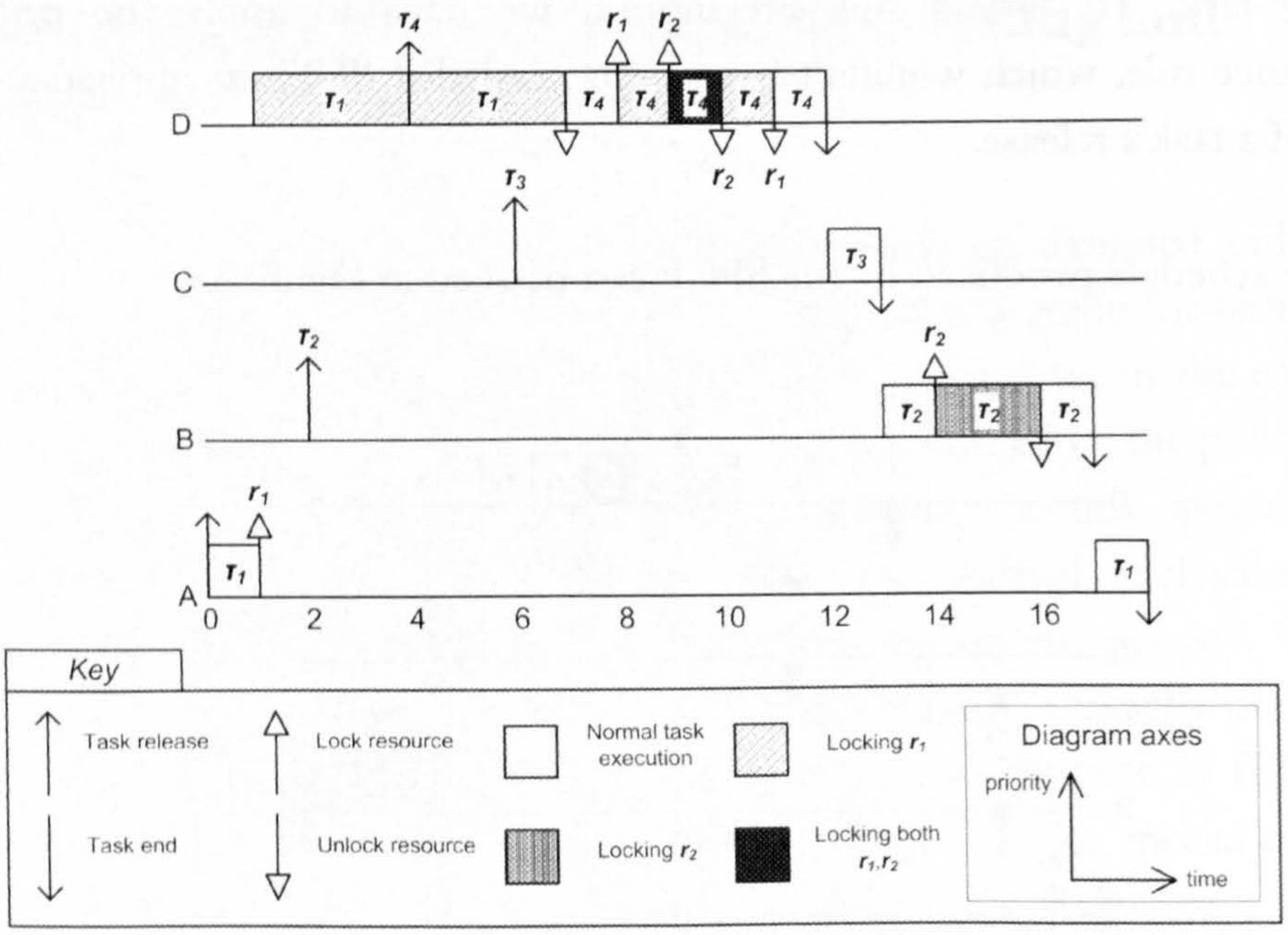


Figure 3.5: IPCP schedule for the tasks of Table 3.1

The IPCP goes one step further than the SPCP. Similar to the simplification the SPCP introduces, it is realised that both the SRP_PLT and EI tests can be substituted by the eligibility test (ET), if the priority of a task is treated as the system ceiling. Just as the system ceiling is elevated to the ceiling of a resource as soon as that resource is locked, so is, in the IPCP, the priority of a task elevated to the ceiling of a resource as soon as it is locked. This, again, can be done because ceilings are based on priorities. This is a sort of priority inheritance, where the inherited priority is not the priority of another task but the ceiling of a resource. IPCP also has no need for the equivalent of a preemption level test (PLT), since it does not use a system ceiling. Therefore, the only mechanism the IPCP uses is the execution eligibility test (ET). In other words, it simply relies on fixed-priority dispatching to enforce the necessary conditions for deadlock prevention and bounded priority inversion.[8]

The gain by using IPCP is that it is easier to implement. It just uses preemptive fixed-priority dispatching together with the standard definition of resource ceilings. Priority inheritance only happens once at the point of locking and not every time a higher priority task is blocked. Also, there is no need for special checks at the point of a task's release. If the released task is not eligible to run, the dispatcher will simply not select it. This can be seen in Figure 3.5, when at time $t=2$ τ_2 is released. Remarkably, in spite of all these differences, the schedule it produces is the same as the SPCP (Figure 3.4), in

[8] This is why IPCP is also known as the Priority Ceiling *Emulation* Protocol.

the sense that in both diagrams the same tasks run at the same moments in time. On this merit it can be considered to support sharing of the run-time stack.

However, there is a drawback to IPCP. The above observations are true only in the case where tasks do not suspend themselves. Once that happens, the protocol is broken. When tasks suspend, unbounded priority inversion can occur. Additionally, if tasks suspend while holding resources, deadlocks are possible. The reason for that is that since the protocol relies on the task's priority to carry out the role of the system ceiling, once the task is blocked, the system has no "memory" as to which tasks it must prevent from running and no way to stop a task from locking a resource that might be needed by the blocked task. In contrast, the SPCP does not have this problem, since it explicitly keeps track of the system ceiling.

As a note, we can say that based on the above, we can now better understand the role of the system ceiling. *Just as preemption levels allow the resource sharing protocol to be independent of any particular scheduling policy, the system ceiling disentangles the protocol from the dispatching mechanism.*

3.4 The Preemption Level Protocol (PLP)

In this section we examine the preemption level protocol (PLP), which, as already explained, is of particular relevance to our work. The principles on which PLP achieves resource sharing are not new. Rather, it is a novel way of implementing a preemption-ceiling protocol to handle resource sharing of EDF tasks scheduled on priority queues. The protocol is specified in the context of the Ada programming language, which does not allow suspension while locking. Furthermore, it has been accepted as part of the new proposed Ada standard [Ada-Europe 2007], *where the assumption is that it enforces the SRP.* We will first describe the protocol and then explain its mechanism.

Under PLP, an EDF task's preemption level is assigned as in the SRP, monotonically according to its relative deadline. The EDF task's priority parameter holds its preemption level and not its eligibility (the task's eligibility is its absolute deadline). This does not present a problem, since preemption levels are static as are priorities in fixed-priority scheduling. However, task dispatching is still based on the priority parameter. Resource ceilings are also

assigned as in the SRP. Additionally, the protocol assumes that in Ada we can specify, within the range of priority queues, a number of consecutive priority queues dedicated to EDF scheduling. PLP is described in [Burns et al. 2004] with the following three rules:

> **Rule 1**: *All EDF priority queues are ordered by absolute deadline (shorter absolute deadline implies closer to the head of the ready queue, i.e. EDF dispatching).*
>
> **Rule 2**: *Execution while holding a resource occurs at the priority ceiling of that resource (i.e. the existing rule in IPCP, which Ada uses).*
>
> **Rule 3**: *Whenever a task τ becomes runnable, it is placed on the highest non-empty EDF priority queue R, such that the base priority of τ is greater than R and the absolute deadline of τ is less than the deadline of the task at the tail of queue R. If no such R exists, τ is added to the lowest EDF priority queue.*

Let us suppose we have a priority range [*min*,*max*] and we apply EDF scheduling to priority queues [*low*,*low+3*]. Three tasks τ_1, τ_2, τ_3 with relative deadlines $D_1 > D_2 > D_3 \Leftrightarrow \pi(\tau_1) < \pi(\tau_2) < \pi(\tau_3)$. Let us suppose that $\pi(\tau_1) = 1, \pi(\tau_2) = 2, \pi(\tau_3) = 3$. Furthermore, τ_2 uses r_1 and τ_3 uses r_2, hence $\lceil r_1 \rceil = 2, \lceil r_2 \rceil = 3$. If the tasks execute as in the Gantt chart of Figure 3.6.a, the EDF priority queues under PLP would look like Figure 3.6.b. We assume that during this particular release of each task their absolute deadlines are $d_1 > d_2 > d_3$.

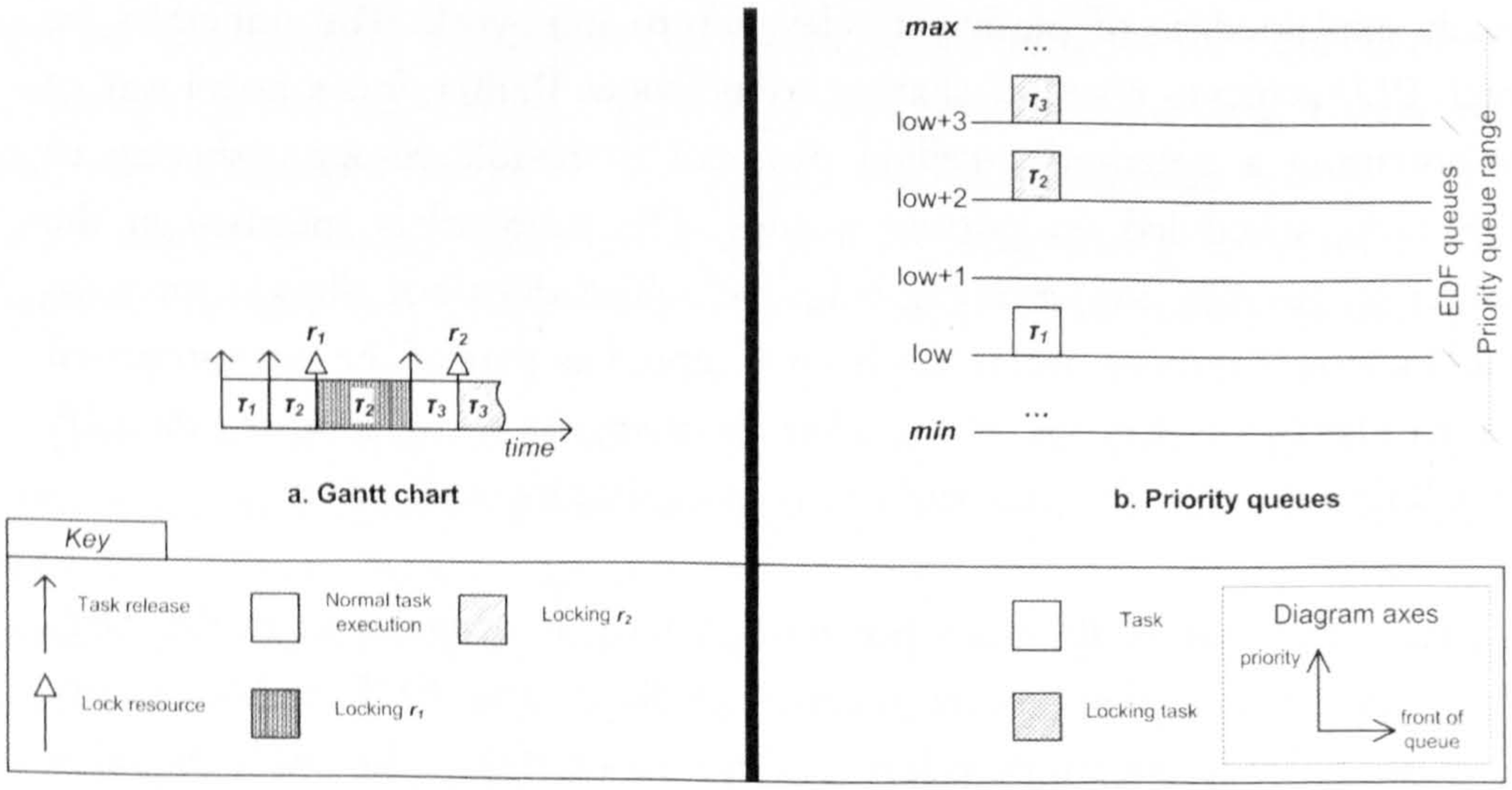


Figure 3.6: Priority queues with PLP

Task τ_1 executes first at priority *low*. After a while it is preempted by τ_2, also at *low*, which has a shorter absolute deadline, and τ_1 is placed on the *low*

queue. τ_2 proceeds to lock r_1 and, therefore, its active priority parameter, i.e. its preemption level, is raised to the ceiling of r_1. Then τ_3 is released and in turn preempts τ_2, since its absolute deadline is shorter than τ_2 *and* its preemption level is greater than the ceiling of r_1, which happens to be the highest locked resource, i.e. the system ceiling. If a fourth task τ_4 comes in with $\pi(\tau_4) = 4$ and $d_2 > d_4 \geq d_3$, it will not be able to preempt τ_3 and will be placed on queue *low+2* in front of τ_2.

Most tasks are on the *low* queue, not locking any resource. Each locking task is on a queue above *low* and, conversely, each occupied queue above *low* signifies a resource being locked. For queue *low+i* the locked resource has a ceiling of *i*. The locking task is always at the back of queue *low+i*. For a task to release a resource no other task must exist on the queue in front of it (or it would have been preempted). Therefore, when the task releases the resource the queue will be empty. The task's priority parameter (i.e. its preemption level) is then set to the value it had before locking, and it is placed on a lower queue according to Rule 3. In front of a locking task on the same queue are tasks that can preempt it. Since PLP tries to apply SRP, preempting a locking task would mean satisfying the SRP rule. So, let us examine what happens when a task is released.

To implement the SRP rule we must enforce the eligibility test (ET), the preemption level test at the point of a task's release (SRP_PLT) and eligibility inheritance (EI). As we can see, the rules force a locking task to occupy a higher queue than *low*. This is because PLP uses the same mechanism as IPCP and raises the priority parameter of a task to the ceiling of the locked resource. However, there is a qualitative difference. In IPCP the priority parameter is both the eligibility of the task and its preemption level. In PLP it is just its preemption level, because PLP deals with tasks whose preemption level is different from their eligibility. Therefore, this mechanism cannot emulate both the SRP_PLT and EI, as in the IPCP. Instead, ceiling emulation in PLP enforces only the SRP_PLT test. Raising the preemption level of the task has the effect, as in IPCP, of emulating the system ceiling and emulating early blocking of tasks[9]. The priority of the highest occupied queue is, in essence, the system ceiling. In fact, every occupied queue above *low* becomes the

[9] Tasks with lower priority than the ceiling of the highest locked resource do not really block; they just have lower priority than the locking task and wait on the priority queues to be dispatched.

system ceiling, when all locking tasks on queues higher than it have released their resources.

Proposition 3.4: The preemption level protocol enforces the preemption level test at the point of a task's release.

Proof: In preemption-ceiling protocols, when no resources are locked, the system ceiling must be *zero* (lower than any legal preemption level value) and therefore all tasks will have greater preemption level than it. This is emulated in the PLP by all tasks being on the *low* queue and all higher EDF queues being empty. In this case a released task will be placed, by default, on the low queue and the only criterion for its position will be its eligibility (i.e. its absolute deadline). This amounts, by default, to having greater preemption level than the system ceiling.

In order to preempt the running task when resources are locked, a released task, apart from greater execution eligibility, also needs a greater preemption level than the priority of the highest occupied queue. This priority is the ceiling of the highest locked resource, which, by definition, is the system ceiling. Therefore, the mechanism is equivalent to the preemption level test. ■

Proposition 3.5: In the context of the Ada language, the preemption level protocol is deadlock-free.

Proof: Since Ada does not allow a task to self-suspend while locking, proof of this proposition follows from Proposition 3.4 and Theorem 3.2. ■

By extension, the PLP cannot be implemented in systems where task suspension while locking is possible. Additionally, it does not enforce the EI and therefore allows potentially unbounded eligibility inversion [Zerzelidis et al. 2007]. This fact is demonstrated with examples. First, we show in Figure 3.7 the schedule produced by PLP for the tasks of Table 3.1. Figure 3.7 presents two ways of viewing the execution of tasks under PLP. Figure 3.7.a shows each executing task according to its eligibility, that is, its absolute deadline. Figure 3.7.b shows task execution according to which priority queue each task is executing on. Task preemption levels were specified as $\pi(\tau_1) < \pi(\tau_2) < \pi(\tau_4) < \pi(\tau_3)$. Let us assume that they are $\pi(\tau_1) = low + 1$, $\pi(\tau_2) = low + 2$, $\pi(\tau_4) = low + 3$, $\pi(\tau_3) = low + 4$. At time 0 task τ_1 is released, executes at priority *low* and proceeds to lock r_1. At that point Figure 3.7.b shows that its priority is raised to the ceiling of the resource, which is *low+3*. At times 2 and 4 tasks τ_2 and τ_4 are respectively released, but cannot preempt

as if they were blocked by the system ceiling (Figure 3.7.a). In fact, they cannot preempt because they are released at priority *low* according to the PLP rules (Figure 3.7.b). However, task τ_3, released at time 6, can preempt τ_1, because its preemption level is higher than priority of the queue where τ_1 executes *and* its eligibility is higher than that of τ_1 (Figure 3.7.a). This constitutes eligibility inversion for τ_4 which wouldn't occur under SRP. This is possible since τ_1 does not inherit the eligibility of τ_4.

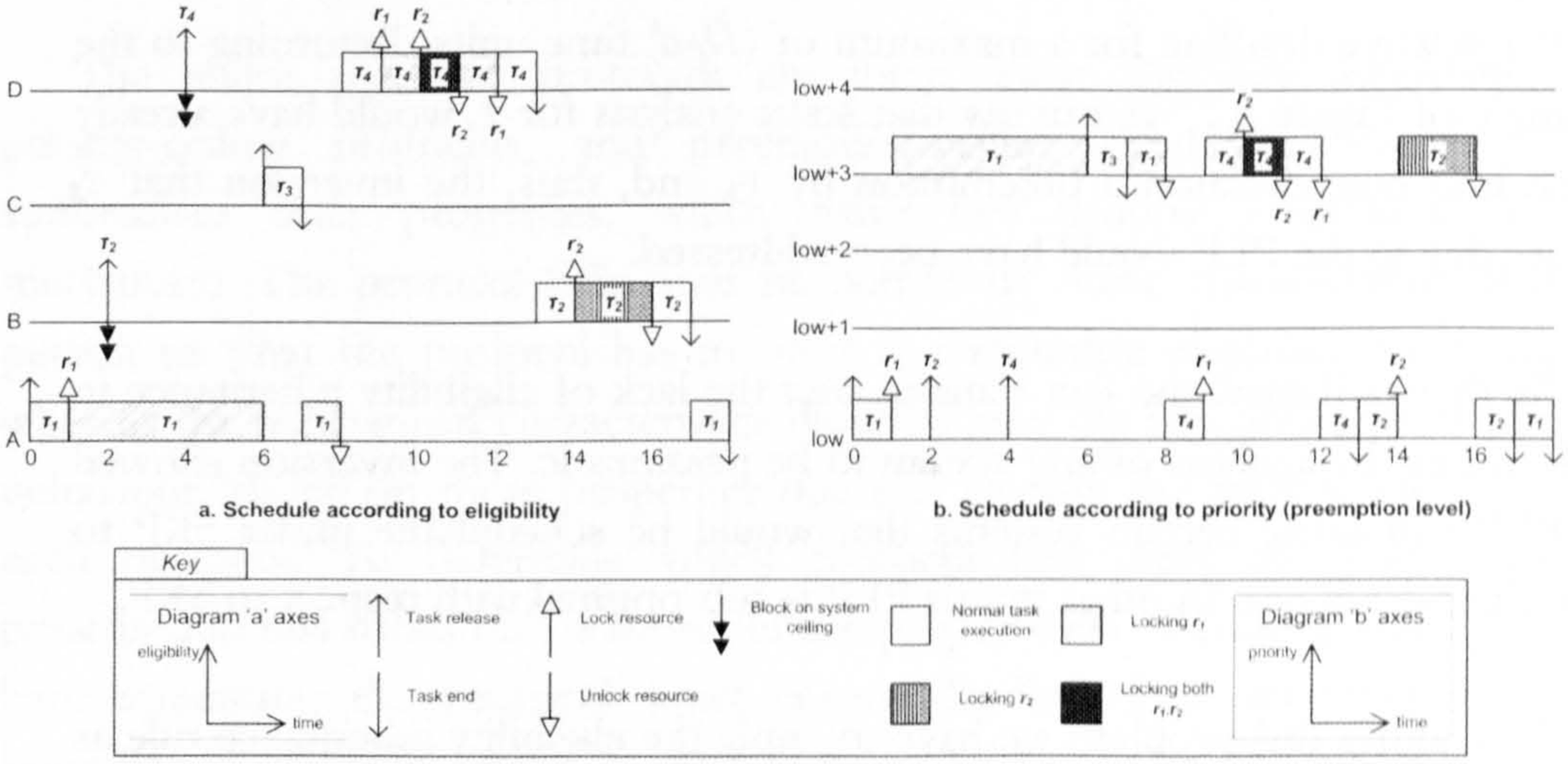


Figure 3.7: PLP schedule for the tasks of Table 3.1

Therefore the PLP does not implement the SRP correctly. This is of significant importance since the protocol has already been adopted in the new Ada reference specification as part of the Real-Time Annex [Ada-Europe 2007]. This could mislead the programmer into believing that eligibility inversion is bounded as in the SRP, while in fact it is not.

We have to point out, however, that the notion of unbounded eligibility inversion here is not the same as the familiar one of unbounded priority inversion. In a static eligibility system there is no possibility a high eligibility task can ever be preempted by a low eligibility task under normal conditions. Therefore, when a medium priority task preempts a low task, which in turn is locking a resource on which a high task has blocked, the middle task is doing something that would normally be impossible. Namely, it is executing in preference to the high task. This effect is the classic case of unbounded priority inversion. On the other hand, when task-level dynamic eligibility algorithms, like EDF, are used, any task has the potential to preempt any other task under specific circumstances. In the case of EDF, for example, we can have two tasks τ_i and τ_j that have relative deadlines D_i and D_j respectively, with

$(D_i - D_j)=d>0$. Usually, τ_j will execute in preference to τ_i since it has a shorter relative deadline. But in some cases it could be that τ_i is released d to D_i time units before the release of τ_j. In that case τ_i will execute in preference to τ_j. So, in task-level dynamic algorithms we cannot be certain of the precedence order between any two tasks for their every possible release. Therefore, in such systems a case can be made that the static analysis will always take into account the possibility of any task preempting any other task. Thus, in the above example for EDF, task τ_j could potentially be preempted by any task τ_i with greater relative deadline for a maximum of (D_i-d) time units. Returning to the example of Figure 3.7, we can say that static analysis for τ_4 would have already taken into consideration a preemption by τ_3, and, thus, the inversion that τ_3 causes due to the PLP would have been addressed.

Be that as it may, the fact remains that the lack of eligibility inheritance in PLP causes the analysis of any system to be pessimistic. The inversion allowed by PLP will cause certain systems that would be schedulable under SRP to miss their deadlines. In other words PLP is sub-optimal with respect to SRP.

To counter this problem we have to apply the eligibility inheritance rule as part of the third rule of PLP. A key feature of the protocol that must be recognised in order to counter the problem is that, due to the way the PLP emulates the system ceiling, a locking task must always be at the back of the queue. This way, once the locking task releases the resource, the queue will become empty, thus emulating a drop of the system ceiling. This rule can be amended as follows:

> **New Rule 3**: *Whenever a task τ becomes runnable, it is placed on the highest non-empty EDF priority queue R, such that the base priority of τ is greater than R and the absolute deadline of τ is less than the deadline of the task at the tail of queue R and also less than the deadlines of all other tasks on ready queues with priorities strictly less than R. If no such R exists, τ is added to the lowest EDF priority queue.*

The new rule makes, of course, the implementation of PLP more difficult, but it is necessary for a correct translation of the SRP. This amendment is equivalent to the solution given in [Zerzelidis et al. 2007], which specifically corrects the wording of the protocol as described in [Ada-Europe 2007]. [Zerzelidis et al. 2007] also provides the proof for this correction, which is not included here, since it is outside the scope of this thesis.

3.5 Summary

In this chapter we examined the most widely known and used resource sharing protocols, in order to form an opinion on which would be the best choice in a scheduling environment of arbitrary schedulers, both in number and in policies.

The major types of protocols are three: basic eligibility inheritance, priority-ceiling protocols, and preemption-ceiling protocols. Table 3.3 summarises their properties, which form two groups: behaviour and mechanism. The protocol behaviour properties are those characteristics that pertain to what the protocol has to offer in countering eligibility inversion, whereas the mechanism characteristics describe how the protocol achieves its behaviour. Based on these properties one can identify the relative merits of each protocol. To determine which protocol best suits the framework presented in this thesis only a subset of the properties is of consequence. At a bare minimum, the protocol must ensure deadlock-free operation while bounding eligibility inversion as much as possible. As can be seen, most of the protocols achieve this. However, the main concern is for the protocol to possess a policy-independent mechanism. The only protocols able to operate under different scheduling policies are the SRP, BPreCP and MSRP. Finally, in order for the framework to address the greatest possible number of systems, it must cater for situations where tasks self-suspend while holding resources. BPreCP is the only protocol satisfying all three criteria. It is resilient to task self-suspension, as proven with Proposition 3.3, since the BPreCP_PLT takes place at the point of locking a resource and the system ceiling guarantees that no other task will be able to lock potentially needed resources.

Table 3.3: Comparison of resource sharing protocols

	Basic Eligibility Inheri-tance	Priority-ceiling protocols			Preemption-ceiling protocols			
		PCP	SPCP	IPCP	BPreCP	SRP	MSRP	PLP
Behaviour								
Deadlock-free	✗	✓	✓	✓	✓	✓	✓	✓
Bounded Eligibility Inversion	Not smallest possible	One instance	One instance	One instance	One instance	One instance	One instance	Not smallest possible
Resilient to task suspension while locking	✗	✓	✓	✗	✓	✗	✗	N/A [10]
Allows sharing of run-time stack	✗	✗	✓	✓	✗	✓	✓	✓
Minimum blocking with stack sharing	✗	✗	✗	✗	✗	✗	✓	✗
Mechanism								
Policy independent mechanism	✗	✗	✗	✗	✓	✓	✓	✗
System ceiling	✗	✓	✓	✗	✓	✓	✓	✗
System ceiling emulation	✗	✗	✗	✓	✗	✗	✗	✓
Eligibility inheritance	✓	✓	✗	✗	✓	✓	✓	✗
Early blocking	✗	✗	✓	✗	✗	✓	✓	✗
Early blocking emulation	✗	✗	✗	✓	✗	✗	✗	✓

Taking all factors into consideration, the BPreCP is the best choice. Therefore, it constitutes the basis for the framework's resource sharing protocol, which will be discussed in Section 4.5.

[10] The PLP specifically addresses the Ada programming language, which does not permit task suspension while locking. An implementation of PLP without the no-suspension rule would allow deadlocks.

Chapter 4

A Flexible Middleware Scheduling Framework

This chapter presents a generic framework for task scheduling at the middleware level that allows multiple, diverse, user-defined scheduling policies to co-exist in the system, each one dictating the execution of a particular subset of the task set. The only OS scheduler required by the middleware is the standard fixed-priority preemptive scheduler found in most real-time operating systems, e.g. POSIX, all real-time Linux, Integrity OS, VxWorks, QNX, Windows CE. The user-defined policies are enforced through the manipulation of task priorities. This use of dynamic priority changes to support alternative scheduling policies is well established. The approach adopted here is based on [Burns and Wellings 1997].

The chapter describes the framework, starting from the assumptions made, the choice of protocols, and concluding with the description of the actual mechanism. More specifically, Section 4.1 presents the basic assumptions on which the framework is built. Section 4.2 shows how different scheduling

policies can be enforced through the use of a fixed-priority scheduler. Section 4.3 introduces the notion of a scheduling band operation. Section 4.4 explains how multiple schedulers could be introduced under the framework and gives a detailed definition of the notion of a scheduling band. Section 4.5 talks about sharing resources between different bands. Section 4.6 presents the framework protocol and gives some basic properties. Finally, Section 4.7 provides some concluding remarks.

4.1 Basic assumptions

In constructing the Flexible Middleware Scheduling Framework (FMSF) we assume the following characteristics for the environment it is going to be applied in:

- The middleware runs within the same user-level process as the applications, on top of a real-time operating system. Thus, any objects shared between middleware tasks are defined within the same address space and no issue arises in sharing them.
- Middleware scheduling decisions are taken as part of a task's execution; more specifically, we view the middleware as a collection of libraries to which the task makes calls and part of this library is the middleware scheduler, or *base scheduler (BS)*.
- The *only* scheduling policy enforced by the base scheduler is fixed-priority preemptive dispatching. This means that each middleware application task has, at the least, a fixed priority as a scheduling parameter and changes in task priorities are assumed to have immediate effect.
- Because of the base fixed-priority preemptive dispatching, the scheduling model supported is event-triggered.
- The particular implementation of middleware tasks is irrelevant, as long as the needed scheduling behaviour is supported.
- Every middleware base scheduler operation immediately translates to an equivalent operating system scheduling operation – in particular, changes in task priorities are assumed to have immediate effect on the corresponding operating system task.
- The default model of interprocess communication and synchronisation is shared memory; however, message passing can also be accommodated.
- Tasks are permitted to suspend while holding a resource.

In such a system the middleware application is allowed to introduce its own scheduling policies. These policies are part of application-defined libraries and constitute a second level of scheduling. It is the function of the framework to translate the decisions of all second-level schedulers to base scheduler decisions. However, an important observation needs to be made. The framework does not make provisions for each and every scheduling policy to be supported. Only those policies that can be implemented simply based on the above assumptions are directly supported. Policies that need special provisions are not directly supported; for example, a policy that assigns eligibility based on the exact time tasks have executed on the CPU. In this case the middleware would have to be able to provide a way for the application scheduler to measure or acquire the exact time each one of its tasks has had control of the CPU for. Needless to say, the more capabilities the middleware provides (e.g. CPU time clocks) the better. There is no point, however, for the framework to require them, because its purpose is to be a general scheduling framework and not to show preference for particular scheduling policies by indicating which capabilities should be present. Section 5.3 of the next chapter will examine support for scheduling policies that fall outside the model given by our basic assumptions.

The addition of second-level schedulers means that some middleware tasks might have additional scheduling parameters depending on the type of policies that will schedule them during their lifetime. When a scheduling decision needs to be made, the base scheduler queries the relevant second-level scheduler via an application programming interface (API). This means that second-level schedulers are "passive" and, when queried, provide information to the base scheduler, who carries out the low level scheduling, i.e. the dispatching. They designate which task they want to run, according to their policy, and the base scheduler translates this to a fixed-priority scheduling decision (e.g. changes the fixed priority of a task). We will refer to second-level schedulers as "application schedulers" or "application-defined schedulers" or "application-level schedulers", and we will use these terms interchangeably.

4.2 Enforcing diverse scheduling policies using priority levels

This section presents the rationale for the mechanism of supporting any arbitrary scheduling policy using priority levels, followed by the definition of

when and *where* application-defined policy decisions take place.

4.2.1 Basic scheduling of *n* tasks

Theoretically, leaving aside for the moment the problem of how scheduling decisions are triggered, the decisions of any scheduling policy can be supported on a preemptive fixed-priority scheduler simply by manipulating priorities. This easily follows if we consider that the role of any scheduling policy is to set $n \in \mathrm{N}, n > 1$ number of tasks in order of precedence according to a particular metric. This metric defines the measure of a task's *eligibility* according to the particular policy, so that the ordering of tasks runs from the task with the lowest eligibility to the task with the highest eligibility. Once we have that ordering, we can always find n nonnegative integers $p_1, p_2, \ldots, p_n \in \mathrm{N}$, such that $p_1 < p_2 < \ldots < p_n$, and assign them to each task according to that task's position in the precedence list. These n numbers are then appropriate priority assignments for scheduling the task set under priority scheduling, irrespective of the higher-level scheduling policy that decided on the particular ordering. Moreover, in a single processor system, there can be only one task running at any time, thus splitting the n tasks into two classes, the running task and the $n-1$ remaining tasks. The inequality then becomes $p_1 \leq p_2 \leq \ldots \leq p_{n-1} < p_n$. This means that in the most basic case we only need two priority levels, *medium* and *low* with *medium>low*, in order to express a valid task schedule, regardless of the number of tasks. These two priorities form the basis of what we call a *scheduling band.*

Definition 4.1: *A* scheduling band *is a range of consecutive priorities, being a subset of the base scheduler priority range, which facilitates the dispatching of tasks scheduled according to a particular application-defined policy.*

Definition 4.2: *A task scheduled according to an application-defined policy is called a* band task.

We will symbolise the eligibility of a task τ according to some arbitrary policy as $e(\tau)$. Sometimes, given a set of n band tasks, the task with the highest eligibility according to the band's policy cannot run due to resource sharing. In that case the task that is actually assigned the *medium* priority is the highest eligibility *runnable* task. The concept of a scheduling band is principal to our framework and will be refined and fully defined in subsequent sections.

4.2.2 Scheduling points

The need to always have the correct ordering of tasks according to the policy's eligibility metric gives rise to the need for determining the instances in each task's execution where the application scheduler will need to make a new scheduling decision. We name such an instance a *scheduling point.* As has been already pointed in Section 4.1, the framework is based on event-triggered scheduling due to the nature of the standard fixed-priority scheduler. Therefore, a scheduling point is linked with the occurrence of certain events. Its definition follows:

Definition 4.3: *A* scheduling point *is a call to the operating system or to the middleware, contained within a task's code that can do one or more of the following: i) change the task set, ii) potentially cause the preemption of the task, iii) potentially suspend the task.*

It is useful, at this point, to repeat the definition of suspension given in Chapter 3.

Definition 4.4: *A suspended task is one that temporarily cannot be considered for being assigned the CPU, but rather waits for an event upon which it will become schedulable again.*

A task might become suspended due to an external factor, or it might suspend itself. Alternatively, we will use the term *blocked* for a task that has been suspended purely due to an external factor.

Examples of scheduling points are the following:

- the release of a task
- the termination of a task
- the equivalent of a POSIX `pthread_join()`, `sched_yield()` or `sleep()` call
- a change in the scheduling parameters of a task

Thus, at a scheduling point the precedence ordering of tasks can possibly change, i.e. we might have a context switch. When at a scheduling point scheduling decisions are dictated by the application-defined scheduler. The exact mechanism is presented in Section 4.3.

As a result of the definition of a scheduling point, the scheduling policies supported by the framework are event-triggered and not time-triggered. This means that scheduling decisions are taken in reaction to some event and not at predefined time instances. This does not mean that timers cannot be used at the application level. The expiration of a timer is an event that will trigger a scheduling decision. Rather, it means that scheduling policies that rely on the existence of timers *within the scheduler* are not directly supported. Chapter 5 provides a more complete evaluation of the different scheduling policies supported by the framework.

4.2.3 Decoupling scheduling from normal task execution

In order for scheduling decisions to be made without interference from any of the other band tasks, it is necessary to introduce a third higher priority level *high* in a scheduling band, such that *high>medium>low*. A task's priority is elevated to *high* in order to execute a scheduling point and the accompanying application scheduler code. This allows us to decouple the functions of a task executing application code and executing scheduling code, which is necessary for the correctness of our system. The need for the third priority level is explained with the following property:

Property 4.1: Using the same priority level for application code execution and scheduling code execution leads to incorrect system behaviour.

Proof: This will be demonstrated with a simple example. Suppose task τ_1 is running at *medium* when another task τ_2 with greater application scheduler eligibility is released. τ_2 cannot preempt τ_1, since they have the same fixed priority *medium*, and, therefore, is placed on the queue for *medium*. As a result τ_2 does not have the chance to run the scheduling code associated with its release (which would preempt τ_1) and eligibility inversion takes place. ■

So, to sum up, in the uniprocessor case we need three priority levels from the base scheduler priority range to enforce an application scheduler decision using base scheduler dispatching: the first, *high*, is where execution of scheduling code takes place; the second, *medium*, is where the most eligible task runs; and the third, *low*, is where, in the general case, all other tasks lie. The *high* priority level has expanded the concept of a scheduling band, given in Section 4.2, which will be fully expanded in Section 4.4. Using this scheme we are relying on the operating system priority-based dispatching to carry out application-defined scheduling policy decisions.

4.3 Scheduling band operations

As already mentioned, a scheduling point, by definition, can potentially cause a context switch. Therefore, when a task is at a scheduling point, it is necessary to inform the application scheduler, so that a scheduling decision can be made. This is achieved by *surrounding every scheduling point with calls to the base scheduler*. The base scheduler library, in turn, makes the necessary calls to the application scheduler, which returns its scheduling decision. Hence, in the general case, there are two distinct calls, one *immediately prior* to the scheduling point and one *right after* it. According to our assumption, these calls to the base scheduler library code execute as part of the running task.

In order to canonise the appearance of scheduling points in a task's execution, we identify eight *scheduling band operations* in our framework, each one corresponding to a particular type of scheduling point. All of them are *potentially suspending* operations.

1) A *release* operation, $\overrightarrow{rel(\tau)}$, corresponds to a task τ being released for the first time.

2) A *lock* operation, $\overrightarrow{loc(\tau,r)}$, corresponds to a task τ trying to lock a synchronization construct r, e.g. semaphore, monitor etc. We will refer to these constructs as *resources*, because their standard use is to regulate access to shared resources, e.g. a data bus.

3) An *unlock* operation, $\overrightarrow{unL(\tau,r)}$, corresponds to a task τ unlocking a resource r.

4) A *wait* operation, $\overrightarrow{wt(\tau,r,cv)}$, corresponds to a task τ performing the equivalent of the POSIX `pthread_cond_wait()` call [IEEE 2004] on the condition variable *cv* while locking mutex r. The semantics of *wait* is that the task must be locking r before the operation; executing the scheduling point unlocks the resource and blocks the task, which becomes unblocked only after re-locking r becomes possible.

5) A *suspension* operation, $\overrightarrow{sus(\tau)}$, corresponds to a scheduling point, other than locking or unlocking, that could either potentially or definitely suspend task τ. In general, this encompasses every potentially suspending operation, other than *lock* and *wait*, since they are all treated the same way. The only requisite is that such an operation does not introduce race conditions

between tasks. In that case it would be considered a *lock* operation since the race condition would have to be avoided by enforcing some kind of mutual exclusion, e.g. use of a mutex.

6) A *yield* operation, $\overrightarrow{yld(\tau)}$, corresponds to a task τ executing the equivalent of a `sched_yield()` POSIX call [IEEE 2004]. Therefore, the *yield* operation will always have an effect at the base scheduler level, i.e. yielding the CPU to the next task on the queue for the yielding task's priority. However, as explained in Section 4.2, an executing band task is the only task with priority *medium*. Therefore, a base scheduler *yield* will not cause the CPU to be handed over to another band task. Consequently, within the framework a *yield* operation must have the added semantics of asking the yielding task's application scheduler for a scheduling decision, in light of the desire to yield. The application scheduler will then specify the next task to execute.

7) A *change* operation, $\overrightarrow{chg(\tau)}$, corresponds to a change in the scheduling parameters of task τ. This operation is special in that it is not necessary for τ to execute it; other tasks can change τ's parameters as well. Additionally, because τ is scheduled by an application scheduler, its scheduling parameters will, in the general case, be relevant only to that scheduler. Therefore, this operation's scheduling point will not be an operating system call, but rather a call to the application-defined scheduling library. The only operating system scheduling parameter used by τ is its priority and only the base scheduler should change it. A *change* operation in a task's scheduling parameters might also include a change in its preemption level, when that is needed.

8) An *end* operation, $\overrightarrow{end(\tau)}$, corresponds to a task τ terminating its execution. This, essentially, is a *suspend* operation consisting of a *definitely suspending* scheduling point. However, its semantics are unique and this is why it is presented as a separate case.

Based on the above definitions we can see that certain scheduling operations are made up from the combination of others. A *wait* operation is, in essence, the consecutive execution of *unlock*, *suspend* and *lock*. A *change* operation, which might cause preemption of the running task (e.g. by changing an application policy related scheduling parameter or associating the task with a new band), can be seen as a suspension operation. In this case, suspension would mean *preemption* rather than *blocking*, these two concepts having the same end result from a middleware perspective, although being different in nature. Namely, in both situations the previously running task hands the CPU over to

a new task that is selected by the application scheduler. Finally, a *yield* can also be seen as suspension, in this case self-suspension. Thus, of the eight operations we identify five basic operations, which are sufficient to test against for the framework's correctness, as will be shown in Chapter 5. These are *release*, *end*, *lock*, *unlock* and *suspend*.

Based on the above, and with the definition of a scheduling point in mind, we now give a more complete definition of a scheduling band operation:

Definition 4.5: *A* scheduling band operation *is the combination of a scheduling point and its surrounding base scheduler calls.*

For brevity, we write the preceding base scheduler call of a scheduling operation as *pbsc* and the succeeding ball as *sbsc*. For a particular operation we specify its preceding and succeeding base scheduler calls using the dot notation, e.g. $\overrightarrow{sus(\tau)}.pbsc$.

To ensure the correctness of a scheduling operation, all base scheduler calls must be implemented in a *thread-safe* way. This means that a base scheduler call could be preempted by higher tasks but not for the purpose of executing another base scheduler call. This rule guarantees the consistency of both the base and application scheduler's internal data. Moreover, the base scheduler should be implemented in a way that could recover, should an API call to an application scheduler block or fail to return.

Conceptually a scheduling band operation is depicted in the following diagram:

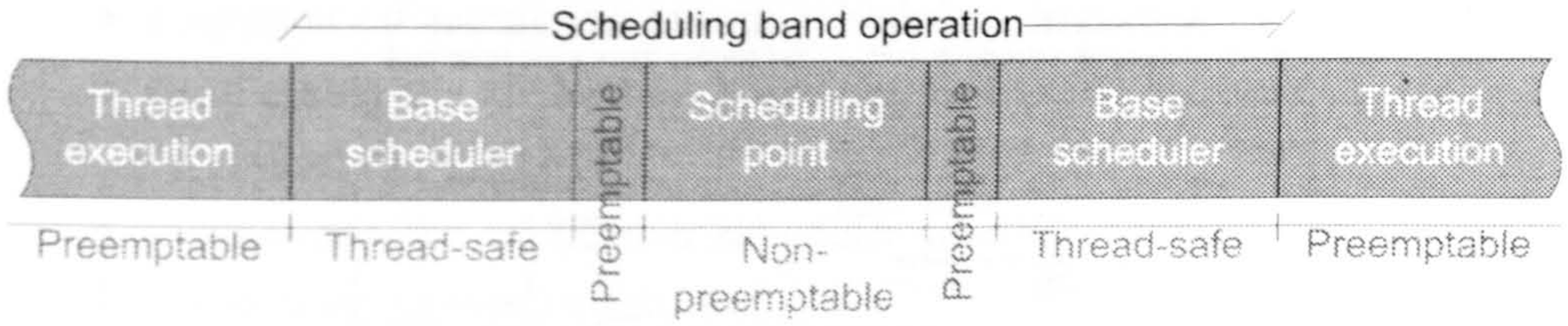


Figure 4.1: Scheduling band operation

There are two obvious cases where there is need for only one call to the base scheduler, as part of the scheduling band operation. These are:

i) When a task is not executing before a scheduling point, and hence we only need a base scheduler call after the scheduling point (e.g. when a task is released). We note that the priority of a band task at the time of a new release will always be the *high* priority of its band.

ii) When a task terminates after a scheduling point, and hence we only need a base scheduler call before the scheduling point (e.g. at the end of a task's execution).

4.3.1 Keeping an operation atomic

There are cases where there is no possibility of a task being blocked at a scheduling point; for example, during an *unlock* operation. One could think that there is no need for a base scheduler call preceding such a scheduling point. Indeed, as we will see, the protocol itself would not be compromised by the lack of a preceding call. However, the call is necessary for the optimal performance of the protocol.

As we can see in Figure 4.1, the task can be preempted in the time between the scheduling point and a base scheduler call. If there is no call prior to the scheduling point, then the task will execute the scheduling point at *medium* (or *medium-lock*; see Section 4.4) priority. Because of that, the possibility exists that another band task will be released and preempt the running task, *right after* the latter has executed the scheduling point and before the succeeding base scheduler call. This can be seen in Figure 4.2 below.

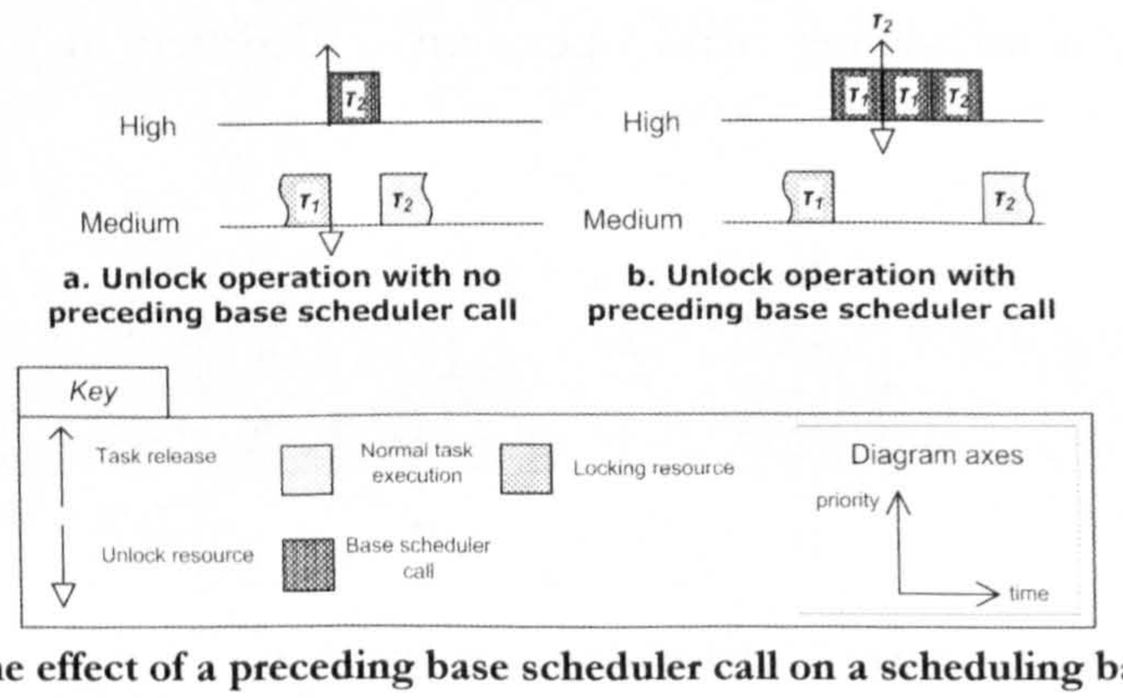


Figure 4.2: The effect of a preceding base scheduler call on a scheduling band operation

In Figure 4.2.a task τ_1 proceeds to unlock a resource without first making a preceding base scheduler call. As soon as it releases the resource it is preempted by τ_2, which executes a release operation. It is decided that τ_2 can preempt τ_1, therefore it is given *medium* priority and τ_1 is put on *low*. Such

preemption of the unlocking task "breaks up" the *unlock* operation by postponing the execution of its succeeding base scheduler call.

In Figure 4.2.b τ_1 first calls the base scheduler, which takes it to *high*, and then releases the resource. This means that when τ_2 is released, it will not be able to preempt τ_1, due to the FIFO ordering of priority queues. τ_1 will execute the second base scheduler call, which will put it on *medium*. Then, τ_2 will run, causing the preemption of τ_1 and its subsequent relegation to priority *low*, while τ_2 continues to run at *medium*.

Although τ_2 gets to run sooner without a preceding call, this is not considered a significant benefit, since it would be very hard to include such a gain in a worst-case execution time analysis. It is far more desirable to keep a scheduling band operation as an undivided whole, thus restricting the possibility of inducing the system with errant behaviour. It is true that having the preceding call cannot prevent a higher priority task outside the band from preempting τ_1, but this is not considered a problem, in the sense that this higher task can not affect scheduling within the band, apart from being able to preempt any task in the band, nor can the band affect it.

4.3.2 The application scheduler API

As mentioned in Sections 4.1 and 4.3, during the course of any scheduling operation the base scheduler informs the task's application scheduler of the operation, using a standard API that every application scheduler library is assumed to have implemented. We consider this API to consist of the following:

- a separate call pertaining to each scheduling operation
- a call asking the application scheduler to provide its most eligible task, e.g. `getMostEligible()`
- a call asking the application scheduler whether a certain task is eligible for execution, e.g. `isEligible(τ)`

As we will see in subsequent sections, resource sharing could cause a band task to execute outside its own band. Under these conditions, it could be the case that certain operations, executed while the task is outside its own band, would not affect the scheduling decisions taken under certain scheduling policies. However, this cannot be generalised and, therefore, a task's own

application scheduler must always be informed of any scheduling operation, regardless of the priority of the task.

4.3.3 Mode changes

Given the previously defined scheduling operations we observe that they are versatile enough so as to allow an application to effect mode changes. In particular, we can identify three such operations: *release*, *end*, *change*. Using the terminology found in [Real and Crespo 2004] we can describe how these operations can support a mode change. Wholly new tasks can be introduced to the system through the *release* operation. The *end* operation will take care of the termination of old-mode aborted tasks and old-mode completed tasks. Finally, the *change* operation can be used to create new-mode, changed tasks.

The actual mode change protocol used by the application is of no interest to the framework. Its operation will be completely unknown to the framework, which simply provides the scheduling operations that can enforce a mode change. For example, the framework does not need to know about the offsets in the release time of new-mode tasks, as described in [Real and Crespo 2004]. Similarly, the framework will not know of any changes in the way tasks use resources. A change in a task's preemption level could affect its resource using potential. Additionally, a change in the set of tasks using a resource would probably render the resource ceiling obsolete. That is, a *lock* operation by a new-mode task with a higher preemption level than the resource's ceiling would cause an error. The framework assumes that such issues will have been considered and resolved at the application level.

4.4 Multiple Schedulers

The mechanism described up to now put forth what is needed for application-defined scheduling within the context of one application scheduler. This concept can be repeatedly applied to non-overlapping ranges of base scheduler priorities to allow us to have multiple application-defined schedulers co-existing under the base scheduler. This way a hierarchy is created amongst application schedulers, with tasks under a particular scheduler executing *if and only if* there are no runnable tasks at higher priorities. However, an important consideration, and one that must be accounted for, is the *sharing of resources* between tasks running under different schedulers. This will be

described at length in Section 4.5. For now, we will point out that this introduces the need for a fourth priority level under application scheduler control, which we name *medium-lock*. Thus, we associate *four* priority levels from the base scheduler range with an application-defined scheduler. We can now give a more detailed definition of a scheduling band.

Definition 4.6: *A* scheduling band *is a range of four consecutive priorities named high, medium, medium-lock and low, being a subset of the base scheduler priority range that facilitates the dispatching of tasks scheduled according to a particular associated application-defined policy.*

We will be referring to a particular scheduling band by using the value of its *low* priority level, e.g. band 3 occupies priorities 3-6, and will symbolise it with B_{low}, e.g. band 3 will be written B_3. Also, we will call the band that a task was initially released in the task's *own band* and symbolise it with B_τ. So, for example, if τ was released in band 3, it will be $B_\tau = B_3$. Furthermore, we will symbolise a band's priority levels with $p_H(B_n)=n+3$, $p_M(B_n)=n+2$, $p_{ML}(B_n)=n+1$, $p_L(B_n)=n$, and the band's application scheduler as $S(B_n)$.

These priorities are to be used in the following general manner – the full protocol governing the use of priority levels in a band will be given in Section 4.6:

- When tasks are released or become unblocked, they execute at priority level *high*. This is the priority of a task executing a scheduling band operation when running within a band.
- A band's highest execution eligibility runnable task normally runs at the *medium* priority level.
- A band's *medium-lock* priority level is occupied solely by tasks coming from lower bands, when locking a resource that is also used by tasks in this band. This level is above *low*, so that the band's non-eligible tasks will not interfere with the locking task's execution. It is, however, below *medium*, since eligible tasks in the band should, by default, be able to preempt any lower-band tasks.
- Finally, priority *low* is the priority usually assigned to runnable application scheduler tasks waiting to run.

Figure 4.3 gives an abstract view of the framework, where the band hierarchy can be seen. The scheduler nearest to the minimum system priority

is the lowest while the one nearest to the maximum system priority is the highest.

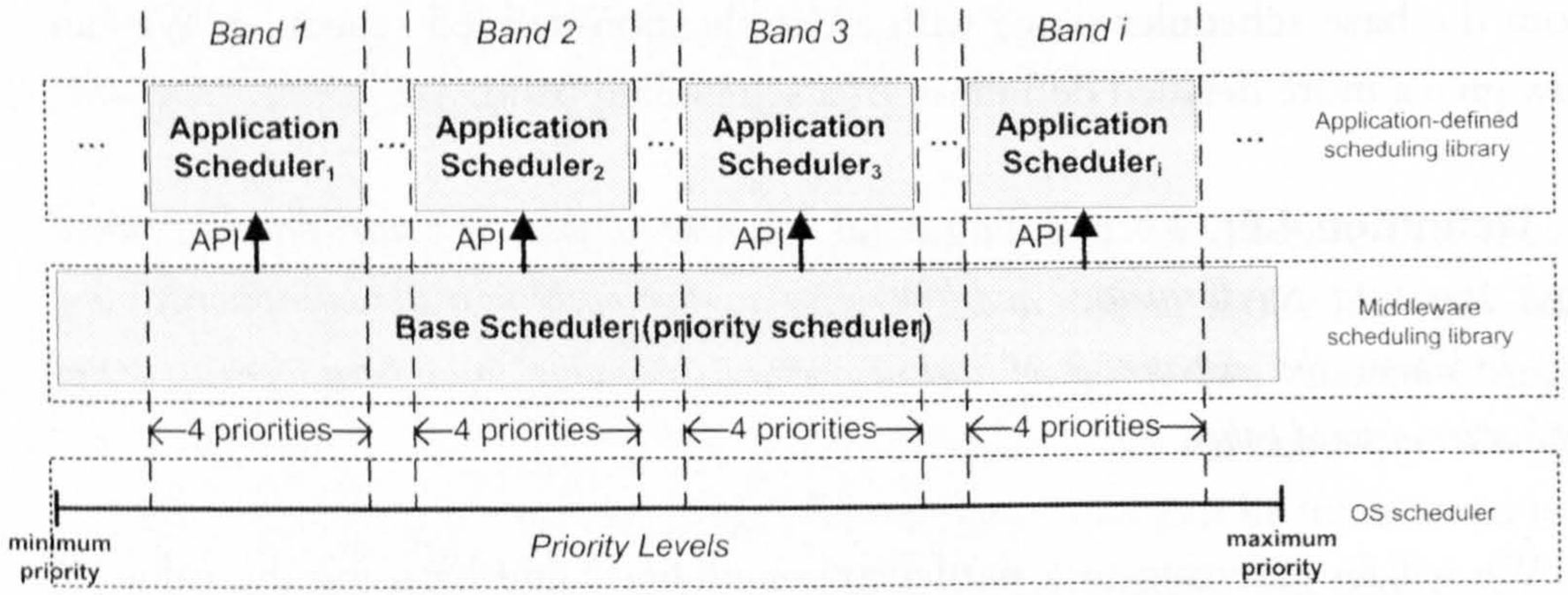


Figure 4.3: Abstract view of flexible middleware scheduling

In Figure 4.3 we can see, at the top, the application-defined schedulers being part of the application-defined scheduling library and each one being in charge of a particular scheduling band. Below them, the priority scheduler is the only one defined by the middleware and is the main part of the middleware scheduling library. Between the two types of schedulers we see one-way arrows signifying the one-way API between them, initiated by the base scheduler. At the lowest level is the operating system scheduler. We can see that the middleware scheduler's range of priorities *conceptually* has a one-to-one correspondence with the operating system priority range. On this range, each band occupies four priorities.

Based on this hierarchical arrangement of application schedulers we can give a system-wide definition of eligibility:

- If the priority of the oldest highest priority runnable task τ is a non-band priority level, then that task is the most eligible in the system.
- The eligibility relation between tasks running in fixed-priority non-band levels is the same as that specified by their priorities.
- Tasks running at a fixed-priority non-band level k are, by default, more eligible than tasks executing in bands below k and less eligible than tasks executing in bands above k.
- Tasks belonging to bands with greater *low* priority are, by default, more eligible than tasks belonging to bands with lesser *low* priority. That is to say: $\forall i, j \in N^* \ni i > j \Rightarrow e(\tau_k) > e(\tau_m), \forall \tau_k \in B_i, \tau_m \in B_j$

- If the priority of the oldest highest priority runnable task τ is a band priority level *other than low*, then the most eligible task in the system is the oldest highest eligibility task whose priority belongs in the corresponding band. That is to say, if $\exists B_i \ni i < p(\tau) \leq i+3$ then the most eligible task in the system is oldest highest eligibility task τ' such that $i \leq p(\tau') \leq i+3$.
- It is an *error* for the priority of the oldest highest priority runnable task to be the *low* priority of a band.

To better understand this, let us first consider the case of the oldest highest priority task in the system running at the *medium_lock* priority of a band. For this to be so, the task must belong to a lower band, *and* additionally, no task in this band can be at *high* or *medium*. Therefore, by default the task at *medium_lock* is the most eligible in this band. Secondly, consider the oldest highest priority task in the system running at a band's *high* priority. It can either belong to this band or to a lower band. In either case, it might not be selected to run, once its scheduling operation finishes. Therefore, we say that, in this case, the highest eligibility task in the system is, in essence, the oldest highest eligibility task in this band. Consequently, a task running at *high* is a source of potential bounded eligibility inversion. Hence, scheduling operations, as far as the base scheduler is concerned, are kept as small as possible. However, their execution time depends on the application schedulers as well. Therefore, implementers of application schedulers must take great care to make the scheduler API as efficient as possible.

4.5 Sharing resources in the framework

As we have already mentioned, the main goal of our framework is to facilitate the use of diverse application-defined scheduling policies in the middleware. After having discussed the conditions and mechanisms under which a variety of scheduling policies can be enforced by using the framework, an equally important property of the framework will now be discussed, which is to allow resource sharing between tasks in the framework. We can distinguish four cases of resource sharing:

- sharing between tasks of the same band
- sharing between tasks of different bands
- sharing between band tasks and non-band tasks

- sharing strictly between non-band tasks

Of these, only the first three are under framework control. The fourth one does not directly involve the framework and will be discussed later in this section. Moreover, the first case, which will also be discussed separately, need not necessarily be arbitrated by the framework's protocol; nonetheless the framework must be able to handle it, in order to guarantee the maximum level of flexibility.

In the three cases where the framework applies, the framework's resource sharing protocol must be able to perform the basic functions of deadlock avoidance and bounded eligibility inversion. Moreover, because of the multi-scheduler environment that the framework harbours, its protocol must also have the characteristic of being independent of any particular scheduling policy. Furthermore, as we have seen in Section 4.1, one of the main assumptions in our framework is that *tasks can self-suspend while holding resources.* The protocol must, therefore, be able to maintain system integrity should such a situation arise.

Chapter 3 provided an analysis of the best known protocols and presented their relative merits. From this analysis it is clear that in systems where task self-suspension is *not allowed*, the best choice would be the Stack Resource Policy (SRP), since, apart from satisfying our criteria, it also allows task sharing of a run-time stack. On the other hand, in systems where such behaviour *is allowed*, only the Basic Preemption-Ceiling Protocol (BPreCP) can retain its correctness and support all the needed requirements, as shown in Section 3.2.3. Therefore, the framework's resource sharing protocol is based on the BPreCP[11]. To enforce the BPreCP protocol means to enforce the BPreCP eligibility test (BPreCP_ET), the BPreCP preemption level test (BPreCP_PLT), and eligibility inheritance (EI) when attempting to lock a resource.

4.5.1 The eligibility test (FMSF_ET)

Within the framework the eligibility test is carried out by a band's application scheduler and its purpose is to decide which task is the most eligible to execute within the band. Task execution in this case means

---

[11] If task self-suspension while locking is not an issue, then the framework could be altered to implement the SRP instead of the BPreCP.

execution of non-scheduling code. Hence, the eligibility test decides which task is most eligible to run at the *medium* or *medium_lock* priorities. As we will see, execution at *medium* priority is directly enforced, by changing the priority of the most eligible task. Execution at *medium_lock* is indirectly enforced, by not allowing any lower eligibility task to preempt the task already present at *medium_lock*. A necessary condition for the correctness of the eligibility test is the application of eligibility inheritance as part of the *lock* operation. The test can be expressed as follows:

> **FMSF Eligibility Test (FMSF_ET):** *A task can execute in a band B only if it is the oldest, highest effective eligibility runnable task in the band, and provided that execution eligibility inheritance is being applied by the band's application scheduler S(B), when needed.*

The term "effective eligibility" denotes the task's base eligibility or an eligibility acquired through the application of eligibility inheritance, whichever is higher. The term will be better defined in Section 4.5.3.

4.5.2 The preemption level test (FMSF_PLT)

The preemption level test used in the framework is the same as that of the Basic Preemption-Ceiling Protocol and is given below.

> **FMSF Preemption Level Test (FMSF_PLT):** *A task is granted a request for a resource only if its preemption level is higher than the system ceiling, or if it is the task holding the resource whose preemption ceiling is equal to the system ceiling.*

In applying the FMSF_PLT two other issues need to be addressed, namely, by whom and how the test is carried out. One way would be to have each application scheduler carry out the test. It is immediately apparent, though, that this would unnecessarily complicate the design of an application scheduler, since the scheduler would need to know what the system ceiling is and how to handle it. Furthermore, all application schedulers would need access to the system ceiling, which would, therefore, have to be treated as a shared resource. Finally, any changes to the way the system ceiling is represented would translate to changes to the schedulers. For these reasons the framework assigns responsibility of the FMSF_PLT to the base scheduler. However, when a task blocks due to the system ceiling, the base scheduler

needs to inform the task's application scheduler in order for eligibility inheritance to be applied as per the BPreCP, if necessary.

To examine how the test can be carried out, we note that the comparison it makes between preemption level and system ceiling must be meaningful, irrespective of the band the task belongs to. In other words, the system ceiling needs to be calculated in such a way that the FMSF_PLT can compare it to preemption levels of tasks from all bands. Since the system ceiling is the highest ceiling amongst the locked resources, and a resource ceiling is the highest preemption level among the tasks that use it, any way of assigning unique preemption levels to tasks in the system would seem adequate. However, if the programmer uses one continuous range of values for assigning preemption levels to all tasks in the system, then to assign levels to tasks within a particular band, one should know which preemption levels have been already used for tasks in lower bands. This is a complication of the usage of preemption levels that needs to be avoided, in order to keep the framework as flexible as possible. For this reason we use two notions of a task's preemption level, the *relative preemption level* (π) and the *absolute preemption level* ($|\pi|$). Each band is assigned the same range of relative preemption levels $[1,l]$, where $l \geq 4$ is a natural number chosen at system start-up[12]. From this range, each task in the band is assigned its relative preemption level, according to Rule 3.1 seen in the previous chapter. A task's absolute preemption level is a function of its relative preemption level and the band i it belongs to. The value of $|\pi|$ for a task with a given π, belonging to band i, is given by the following equation:

$$|\pi|(i,l,\pi) = \begin{cases} (\lceil \frac{i}{4} \rceil - 1) \times l + i \bmod 4 + \pi - 1, & when \quad i \bmod 4 \neq 0 \\ (\lceil \frac{i}{4} \rceil - 1) \times l + 4 + \pi - 1, & when \quad i \bmod 4 = 0 \end{cases} \quad , l \geq 4, \pi \in [1,l]$$

Equation 4.1: Absolute preemption level $|\pi|$

[12] It must be $l \geq 4$ in order to guarantee an efficient assignment of preemption levels. Let us assume we have created band 5 and allow only 3 relative preemption levels per band, i.e. *l=3*. According to Equation 4.1, the $|\pi|$ for a task in the band with π=*1* would be *4*. This would mean that for priority levels *1-4* we would only have 3 preemption levels available to assign. This cannot be allowed.

where $i, l, \pi \in \mathrm{N}^*$. Equation 4.1 is a monotonically increasing function for i and π.

$$|\pi|(i+1, l, \pi) > |\pi|(i, l, \pi) \text{ with } l, \pi \text{ constants}$$
$$|\pi|(i, l, \pi+1) > |\pi|(i, l, \pi) \text{ with } i, l \text{ constants}$$

For l it is a non-decreasing function.

$$\begin{cases} |\pi|(i, l+1, \pi) = |\pi|(i, l, \pi), & i \leq 4 \\ |\pi|(i, l+1, \pi) > |\pi|(i, l, \pi), & i > 4 \end{cases} \text{ with } i, \pi \text{ constants}$$

Regarding l as a constant in any given system, we can think of an $|\pi|$ value as a pair (i,π). Since this pair is unique for each task τ, we can write the absolute preemption level function as $|\pi|(\tau)$.

Based on the above definition of absolute preemption levels, we give the following definition of resource ceilings and of the system ceiling.

Definition 4.7: *Each resource is assigned a* preemption ceiling *value that is equal to the highest absolute preemption level among the tasks that use it.*

Definition 4.8: *The* system ceiling *is equal to the highest ceiling amongst the locked resources and, thus, has an absolute preemption level value.*

The reason we assign each band the same number of preemption levels is that it allows for a static, a priori allocation of absolute preemption levels to priority levels. Once the number of relative preemption levels needed per band has been specified at system start-up, we can construct a table which, for every priority level i, will contain the minimum absolute preemption level value that would be used in a possible band i. We will call this table *apl_table*. Its use is made clearer in the following example.

Let us assume that the needed number of relative preemption levels per band is 100. Then for the first 9 priority levels the table would look like Table 4.1.

Table 4.1: Absolute preemption level distribution (*apl_table*)

Priority	1	2	3	4	5	6	7	8	9
$\lvert\pi\rvert$	1	2	3	4	101	102	103	104	201

We can easily see that there is a pattern that changes every four priority levels, first at 5 and then at 9, 13, 17, 21 etc. For example, we can calculate that $\lvert\pi\rvert$=1001 will be at priority level 41. If we assume a priority range of 256 priorities for the base scheduler, then the maximum possible absolute preemption level will be 6400 and will be assigned to tasks in band 253, which is the highest possible band. The way to interpret this table is that if we create a band at priority *i*, occupying range [*i*,*i*+3], then absolute preemption levels used in this band will be in the range [*apl_ table*[*i*],*apl_ table*[*i*+4]-1]. So, for example, if we construct band 3, then the absolute preemption levels reserved for this band will be in the range [*apl_ table*[3],*apl_ table*[7])=[3,103)≡[3,102]. For a band *B*, we symbolise this range with $\Pi(B)$=[*apl_ table*[$p_L(B)$],*apl_ table*[$p_L(B)$+4]).

In addition, the *apl_ table*[*i*] value for priority level *i* is the $\lvert\pi\rvert$ of every fixed-priority non-band task whose priority is *i*. For example, fixed-priority tasks running at priority 8 will have an absolute preemption level of 104, which will be used to calculate the ceiling of any resources it uses jointly with band tasks. We note that no band can make use of $\lvert\pi\rvert$=104. Since priority level 8 is assigned to and used by a fixed-priority task, it follows that it cannot be allocated to a band. At the most, a lower band will start at priority 4, which means that its range of absolute preemption levels will be [4,103]. On the other hand, the lowest priority a higher band can start at is 9, which translates to the range [201,300].

Based on the *apl_ table* and on Definition 4.7 above, we say that *a resource r belongs to band i*, if $\lceil r \rceil \in$ [*apl_ table*[*i*],*apl_ table*[*i*+4]-1]. Similarly, we say that *resource r belongs to the non-band priority k*, if $\lceil r \rceil$=*apl_ table*[*k*]. Furthermore, for the purposes of resource sharing the following definitions apply:

Definition 4.9: *We will call the band to which the ceiling of a resource r belongs to a* locking band *and symbolise it with* B_r^{loc}.

It follows that there can be more than one locking band. In this case:

Definition 4.10: *The* highest locking band *is the band with the highest low priority such that there exists at least one resource that belongs to it.*

The flexibility provided by the use of the *apl_ table* can be demonstrated if we consider how easily an application can add a new scheduler in the system. The range of absolute preemption levels needed by the scheduler will be immediately available without the need for any reshuffling.

As we can see, some preemption levels can be "lost" according to this scheme. In the above example, values 105-200 are not used. This, however, does not present a problem, since, on the one hand, there is no reason for values of $|\pi|$ to be consecutive, and secondly, the maximum value is still containable in a 16-bit integer variable. Even if we consider that the optimum number of relative preemption levels per band is 256 – the same way 256 priority levels are considered optimum for scheduling under a priority scheduler – the maximum value is $|\pi|$=16385, which still fits in a 16-bit variable. In fact, the maximum number of relative preemption levels per band we could accommodate with 16 bits is 1023.

In systems where the lowest priority in the priority range is 0, the framework will treat it as being 1. Similarly, in systems where priority 0 is the highest, the framework will treat it as if it had the value of the lowest priority in the system. So, for example, in a system where the range of priorities is from 255 (lowest) to 0 (highest), the framework will consider priority 255 to be 1 and priority 0 to be 256.

We cannot assume preemption levels are equal to priorities, as in the case of priority-ceiling protocols, seen in Section 3.3. This is easy to understand if we consider, for example, priority 5. If a fixed-priority task with priority 5 had a preemption level of 5, then the number of preemption levels available to the band occupying priorities 1-4 would be only 4. This is too restricting. However, as with priority-ceiling protocols, we assign the same preemption level to fixed-priority tasks with the same priority.

It is important to note that absolute preemption levels are not an issue for the programmer. The only thing the programmer needs to do is to define the bands he needs and set the relative preemption levels of tasks, without needing to know the corresponding $|\pi|$ value (although it can be easily calculated). In

other words, absolute preemption levels are for internal use by the framework and the above discussion aimed at making their implementation clear.

4.5.3 Eligibility inheritance (FMSF_PI, FMSF_EI)

The framework applies eligibility inheritance on two levels. The first is priority inheritance applied by the base scheduler, similar to that of the Immediate Priority Ceiling Protocol (IPCP). The other is eligibility inheritance applied by application schedulers similar to BPreCP eligibility inheritance, but only when a task blocks within its own band. These two rules are given below.

> **FMSF Priority Inheritance (FMSF_PI):** *When a task τ_l locks a resource that belongs to a band higher than its own, it executes the critical section associated with this resource at the locking band's* medium_lock *priority. If the resource belongs to a higher non-band priority level, τ_l executes its critical section at that non-band priority.*

> **FMSF Eligibility Inheritance (FMSF_EI):** *When a band task τ_b blocks within its own band due to the system ceiling, the blocking task τ_l acquires an effective eligibility and executes as if it had inherited the execution eligibility of τ_b, if this is greater than its own current effective eligibility. Upon unlocking, the effective eligibility of the blocking task returns to the value it had at the time of locking the resource.*

The term *effective eligibility* is used to cater for situations where there can be no actual inheritance between the blocking and blocked tasks, i.e. the two tasks do not belong to the same band. The only situation where this can happen is when the blocking task is from a lower band or non-band priority than the blocked task. We will write the effective eligibility of task τ as $e_f(\tau)$. If the two tasks do belong to the same band then actual inheritance can take place and the blocking task is said to acquire an *active eligibility*. We will write the active eligibility of task τ as $e_a(\tau)$. To explain the relation between active and effective eligibilities we can say the following: *a blocking task τ_1 acquiring an effective eligibility from some blocked task τ_2 of a higher band, is scheduled as if it was a task of the same higher band and had acquired an active eligibility from τ_2.* A band task's initial effective eligibility is its active eligibility, and its initial active eligibility is its base eligibility. A task can acquire an active eligibility different than its base eligibility only when locking within its own band. When this happens, it must be that $e_a(\tau) > e(\tau)$. A task can acquire an effective eligibility different than its

active eligibility only when locking within a higher band. When this happens, it must be that $e_f(\tau) > e_a(\tau)$. It is possible that a task will have $e_f(\tau) > e_a(\tau) > e(\tau)$.

This section provides justification and explains the use of these rules for each of the four resource sharing situations described at the start of this section.

4.5.3.1 Sharing resources within one band

This is the situation where a resource is shared only between tasks of the same band. Using the framework's protocol for resource sharing within one band is not the only option. The band's scheduler can introduce its own protocol to address eligibility inversion for resources used solely within its band, and this case will be discussed in Section 4.5.4. However, if an application scheduler opts to take advantage of the FMSF, it must perform eligibility inheritance on its tasks, based on the type of eligibility metric it is using. This is because the only meaning eligibility inheritance can have in this case is in relation to the scheduling policy that schedules these tasks. The following rule applies:

Rule 4.1: *The application scheduler needs to make sure that, when one of its tasks is locking a resource, its eligibility will either be the highest eligibility amongst all those tasks that belong to its own band and have blocked on that resource, or the eligibility it had before locking the resource, whichever is higher. We will call this eligibility the* active eligibility *of the task.*

Eligibility inheritance is applied when the FMSF_PLT test fails, and since the aim is to apply the equivalent of the BPreCP rule, it is during the $\overrightarrow{loc(\tau, r)}$ operation that the EI must take place, if required. During locking, the base scheduler informs the application scheduler of the operation and the latter must carry out eligibility inheritance for tasks in its band. This way the next time the application scheduler is asked about its most eligible task, it will return a task whose active eligibility will be the highest eligibility among all pending tasks in the scheduler. This task will be placed on the *medium* priority and will be selected by the dispatcher, if there are no tasks on higher bands and if there are no tasks of the same band about to execute a scheduling operation at priority *high*. During the $\overrightarrow{unL(\tau, r)}$ operation the application scheduler is again informed and must now undo any eligibility inheritance

caused by the locking of the task. Locking within a band is demonstrated in Figure 4.4, where the application scheduler is depicted as a "black box" with respect to the base scheduler.

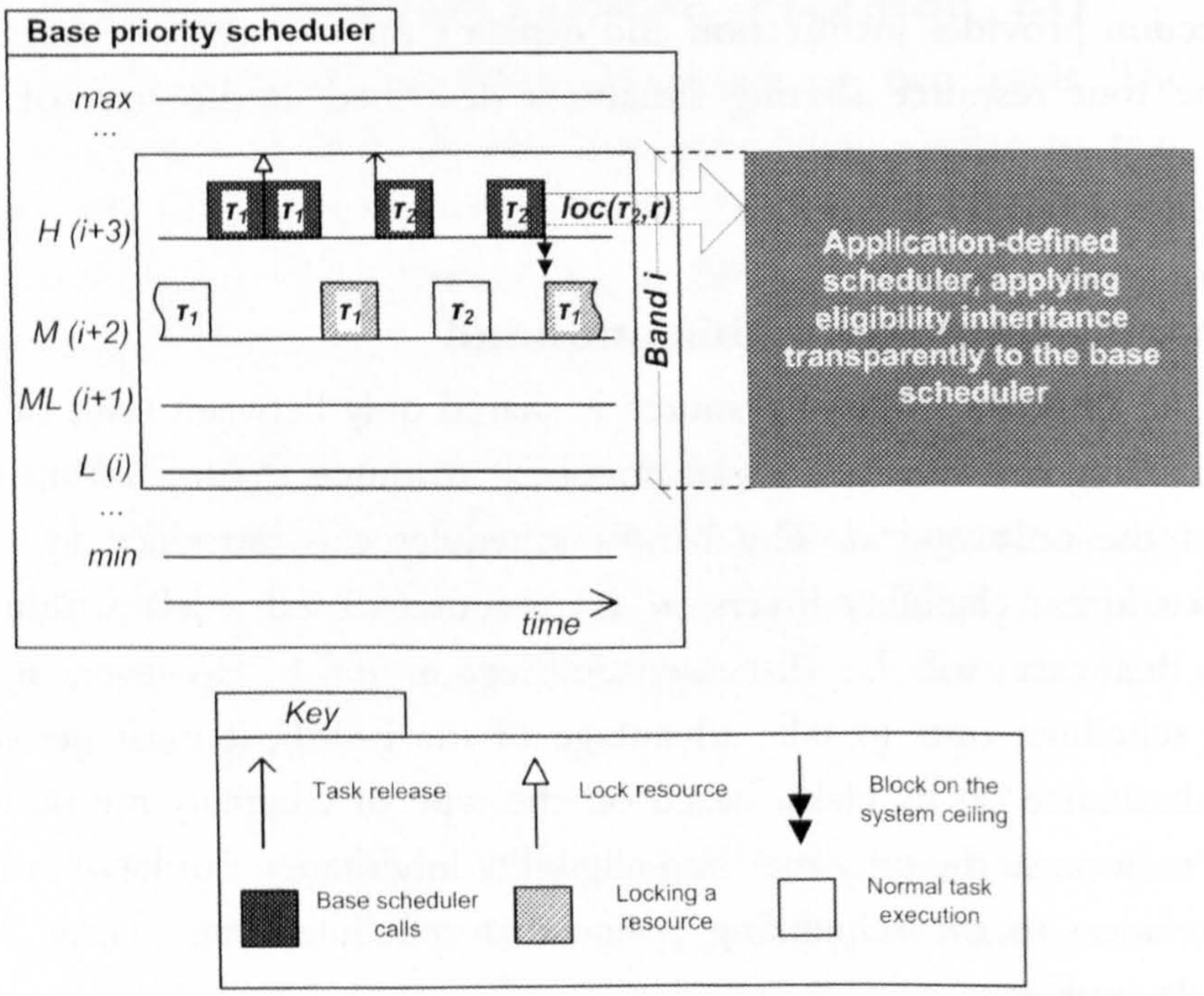


Figure 4.4: Eligibility inheritance as a "black box" operation when locking within a band

In this figure we can see task τ_1 in band *i* locking resource *r*. The task is first executing at *medium* priority and then is moved to *high* to execute the $\overrightarrow{loc(\tau, r)}$ scheduling operation. The scheduling point of locking the resource is surrounded by calls to the base scheduler, which inform the application scheduler of the locking. After the scheduling operation the task continues to run at *medium*. Then task τ_2 is released preempting τ_1. It executes for a while at medium and then tries to lock *r*. However, it blocks on the FMSF_PLT and the application scheduler must apply eligibility inheritance internally, as appropriate.

4.5.3.2 Sharing resources between band and non-band tasks

This section refers to the specific case that a resource is used by *both* band and non-band tasks. In such a situation, we can distinguish two cases: one where the resource ceiling belongs to a non-band priority level and the other where it belongs to a band.

Resource ceiling belongs to a non-band level

This case can be split into two sub-cases. The first is when a band task locks the resource, while the second is when a non-band task locks at the non-band level.

In the first case, eligibility inheritance is enforced in the form of priority inheritance, exactly as in the IPCP. Any band or non-band task that locks the resource has its base scheduler priority raised to the priority of the *apl_table* corresponding to the ceiling of the resource. This has the effect of not allowing tasks with priorities higher than the *high* priority of the locking task's band and lower than the priority of the highest-locking non-band task to preempt. Additionally, any task released at the same priority will not be able to preempt due to the FIFO ordering of the queues. This priority inheritance, in contrast to the eligibility inheritance that takes place within an application scheduler, is happening at the base scheduler level. As explained in Section 3.3, this has the effect of enforcing both the PLT and EI in a way analogous to the SRP, as long as the locking task does not suspend. If it does suspend, the dispatcher will select an equal or lower priority task, which can possibly lead to multiple inversions. However, the presence of the FMSF_PLT in the $\overrightarrow{loc(\tau, r)}$ operation will prevent deadlocks. During the $\overrightarrow{unL(\tau, r)}$ operation the task will return to its band or to its original non-band priority, depending on whether it is a band or non-band task. If it is a band task, the application scheduler has to decide if the unlocking task is still the most eligible task in the band. Locking at a non-band level is shown in Figure 4.5 below.

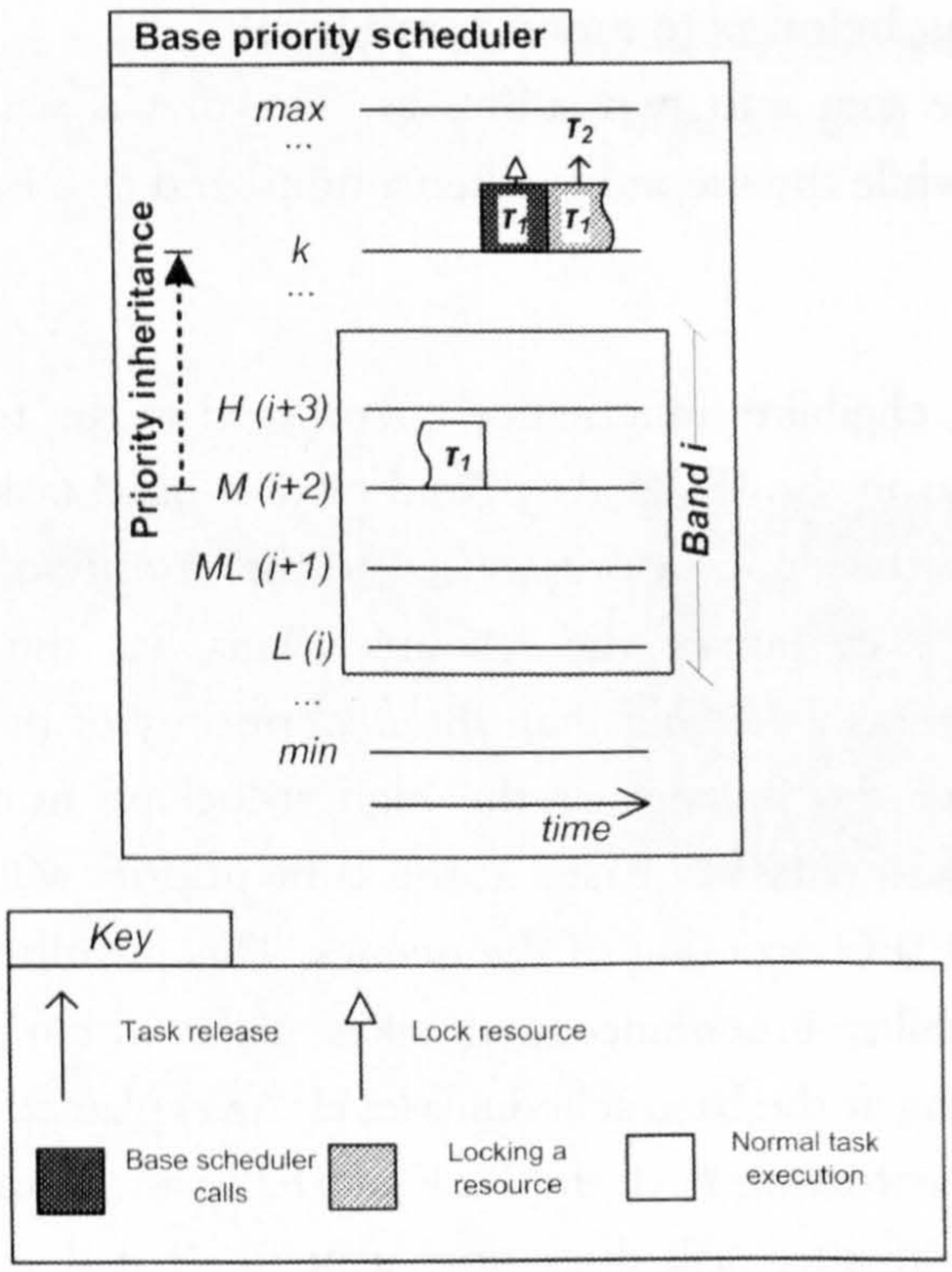


Figure 4.5: Locking at a non-band priority with priority inheritance

Here we can see task τ_1, belonging to band *i*, locking resource *r* whose ceiling corresponds to a non-band priority *k*. Initially, the task again executes at *medium* priority. When the locking scheduling operation begins, the base scheduler determines that the ceiling of *r* lies outside the task's own band and moves the task to priority *k*. This priority inheritance, shown with a dashed arrow (⋯▸), ensures that the scheduling operation will be executed as an atomic operation, as explained in Section 4.3.1. Thus, when task τ_2 is released at priority *k* it cannot preempt τ_1. This way, only tasks with higher priorities than *k* will be able to preempt the locking task. These tasks have, by default, higher preemption levels than the system ceiling and, therefore, will never use *r*.

In the second case, where this time a non-band task locks the same resource *r* of the previous paragraph, the protocol behaves exactly as the IPCP, boosting the task's priority to *k* for the duration of the lock. As a note it has to be pointed out that the implementation will need to check whether *r* is also used by band tasks, otherwise this would be a case of normal non-framework resource sharing between non-band tasks.

Resource ceiling belongs to a band

This is the situation where a non-band task locks a resource whose highest locker is a band task. Because the resource's ceiling belongs to a band, the eligibility inheritance applied is a combination of eligibility inheritance in the locking band and priority inheritance. This is because in this case there are two sets of tasks that must be prevented from causing unbounded eligibility inversion. One is the set of tasks with priorities higher than the locking task's initial priority and lower than the *low* priority of the locking band. The other is the set of tasks in the locking band that have a higher absolute preemption level than the system ceiling but lower eligibility than tasks in the band that have blocked on the system ceiling.

We note that the first group of tasks is identical to the tasks whose preemption is avoided with the use of priority inheritance, in the case of a non-band highest-locker task. Therefore, priority inheritance is again applied. However, since the highest locker is now a band task, there is no specific base priority associated with it. Hence, the inherited priority cannot be that of a task. Instead, it is the "priority" of the locking band. It is exactly this notion of a band's "priority" that the *medium_lock* priority level is. This means that every locking task coming from outside the band, will have its priority promoted to the *medium_lock* priority of the locking band, effectively not allowing any task situated between the locking task's initial priority and the locking band to preempt. Again, the only way for the dispatcher to select one of these intermediate tasks, while the resource is locked, is for the locking task to suspend. Deadlocks will again be avoided due to the FMSF_PLT. The positioning of the *middle_lock* priority level within the band has to allow more eligible tasks to execute in preference to the locking task, while not allowing lower eligibility tasks to preempt. This is why the *medium_lock* priority is placed below the *medium* priority and above *low*.

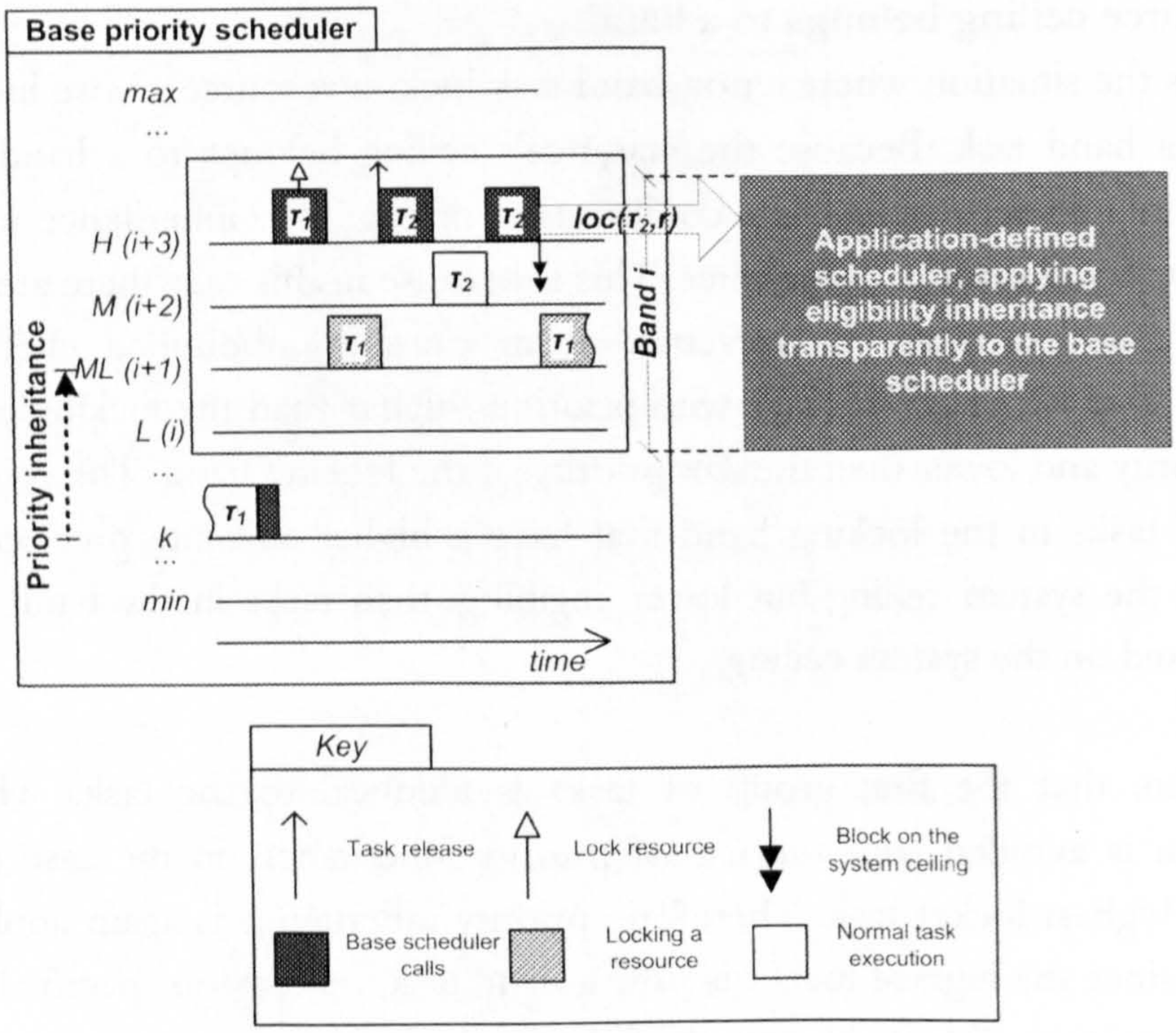


Figure 4.6: Non-band task locking at a band

The second set of tasks, namely the set of tasks in the locking band that could cause eligibility inversion, cannot be stopped by means of priority inheritance. Instead, eligibility inversion is stopped by means of eligibility inheritance in the locking band, as was the case with locking within a band. Application of this inheritance happens as a result of the FMSF, which is based on the BPreCP rule. Section 3.2.1 explained how the PLT test can trigger the application of eligibility inheritance, when the executing task (i.e. the locking task, in the case of the FMSF) has lower or equal preemption level than the system ceiling but higher eligibility than the oldest highest execution eligibility pending task. It is important to note that this inheritance cannot be performed on the locking task, since the notion of eligibility used is understood only within the locking band. Therefore, an application scheduler must always be able to keep track of the eligibility of its oldest highest pending task, even though the locking task might not belong to its band.

Figure 4.6 above demonstrates a non-band task locking at a band. The non-band task τ_1 executes at its priority and calls for locking resource r whose ceiling belongs to band i. The base scheduler initially executes at the base priority of the task and once it determines which band the resource ceiling belongs to, it moves the task to the *high* priority of band i. Locking takes place and then the succeeding base scheduler call moves the task to the

medium_lock priority of band *i*. While τ_1 is executing, τ_2 is released in band *i* at priority *high*. It preempts τ_1 and is moved to *medium* in order to execute. When it tries to lock a resource, it is blocked by the FMSF_PLT while running at priority *high*. Thus, τ_1 is selected by the dispatcher and continues the execution of its critical section.

4.5.3.3 Sharing resources between bands

Resource sharing between tasks belonging to different bands is handled the same way as when a non-band task locks a resource within a band, i.e. by enforcing both priority inheritance and eligibility inheritance within the locking band. The locking task again inherits the "priority" of the locking band, i.e. its *medium_lock* priority[13]. The locking band again keeps track of its oldest highest pending task and performs eligibility inheritance on its tasks, when needed.

Figure 4.7 demonstrates what happens in such an event. In this figure task τ_1 belongs to band *i* and locks a resource whose ceiling belongs to band *j*. At the beginning, τ_1 executes at the *medium* priority of band *i*. As soon as it calls the locking operation, the base scheduler moves it to the *high* priority of band *i*, which is standard for every scheduling band operation. Then, the base scheduler determines that the resource ceiling belongs to a higher locking band, therefore it elevates the priority of τ_1 to the *high* priority of band *j* in order for the locking to take place. After the scheduling point, the task is moved to the locking band's *medium_lock* priority, which constitutes the priority inheritance part of the protocol. While τ_1 executes at this priority, task τ_2 is released in band *j*. The release takes place at priority *high*, thus preempting τ_1. The application scheduler of band *j* is informed of the release and, since there is no other task in the band, allows τ_2 to execute.

---

[13] This use of priority inheritance means that at any point in time an arbitrary band task can be executing in either its own band or in a higher locking band. Thus we say that either of these can be the task's *current band*. We symbolise this with B_τ^{cur}.

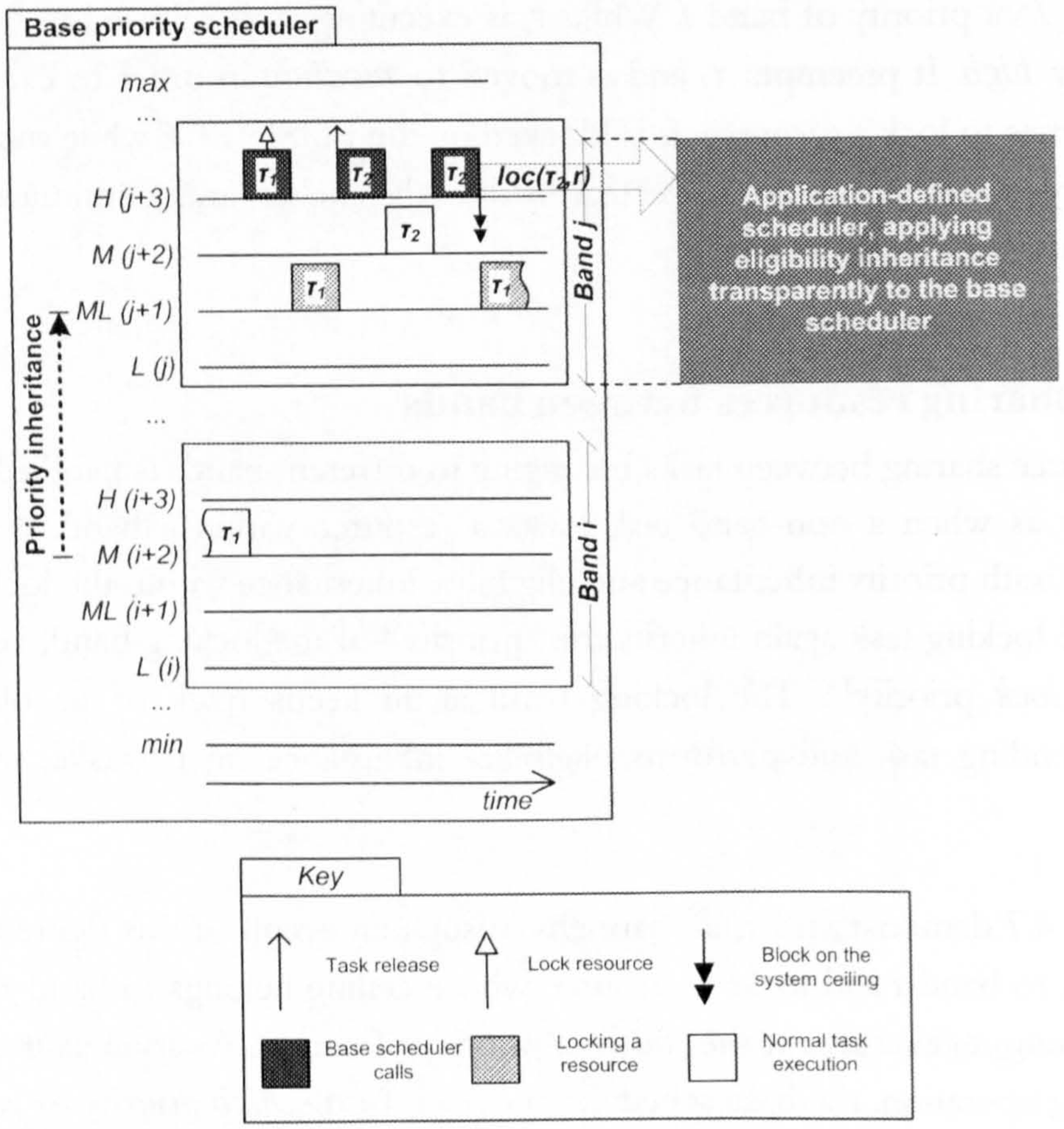


Figure 4.7: Priority and eligibility inheritance when locking between two bands

Accordingly, the base scheduler moves τ_2 to the *medium* priority of band *j*. Then τ_2 executes a locking operation and, hence, is moved to *high*. However, since its preemption level is not higher than the system ceiling, τ_2 blocks and its scheduler is informed of the blocking. As a result, eligibility inheritance is applied by the application scheduler of the locking band. The base scheduler now selects τ_1 to run, since it is the next highest runnable task in the system. It is important to note that the application scheduler of the locking band has no knowledge of τ_1 or of the system ceiling. It is passively informed of the blocking by the base scheduler.

4.5.3.4 Sharing resources strictly between non-band tasks

Although it should be obvious that this case falls outside the framework's jurisdiction, it is important to point out that the framework's resource sharing protocol does not, in any way, inadvertently affect resource sharing strictly between non-band tasks. Essentially, as far as non-band tasks are concerned, bands are "invisible" entities. Non-band tasks, which do not share resources with band tasks, will never execute within a band. If they lock a resource

shared with other non-band tasks, the priority-based resource sharing protocol that governs that resource will be applied by the base scheduler, as normal.

4.5.4 Using an application-defined resource sharing protocol

An important aspect of the "scheduling encapsulation" a band provides to its tasks, is the fact that it can also make use of its own "custom" resource sharing protocol. It is important to keep in mind, though, that this is possible only for those resources that are solely used by tasks of one band. The base scheduler could be instructed not to perform the preemption level test when locking resources which it identifies to be governed by a protocol other than the framework's protocol. Instead, for these resources the band's application scheduler can enforce its own protocol every time it is informed of a *lock* or *unlock* operation, or it can even choose not to enforce a protocol.

If the application scheduler does use a protocol, it can be to bound eligibility inversion, avoid deadlocks, or both. It is worth pointing out that enforcement of the application-defined protocol does not have to happen only during a *lock* or *unlock* operation. For example, an application scheduler might want to enforce the SRP, if it is certain that its tasks do not self-suspend. In this case, it would have to carry out its own PLT test during each band task's *release* operation. It is important to note here that even if the application scheduler does use its own resource sharing protocol, it can still choose to make use of the FMSF_PLT for deadlock avoidance, as will be discussed in Section 5.3.2.1.

Another point worth making is that use of an application-defined protocol within a band does not exclude a task of that band from using resources outside the band. The only requirement is that the task has been assigned a relative preemption level, because such resources could potentially be used by tasks in other bands that do use the FMSF. Therefore, the framework must be able to perform its PLT when locking takes place, so as to guarantee correct behaviour. Since absolute preemption levels are assigned to each band statically, regardless of whether the band's scheduler wants to use the FMSF or not, the base scheduler is always able to assign an absolute preemption level to a task from any band and thus carry out the preemption level test for that task.

4.5.5 Properties of the resource sharing protocol

This section contains some important observations concerning the framework's resource sharing protocol.

Observation on the preemption level test

A consequence of the preemption level test is that an application scheduler needs to provide scheduling decisions only for those of its tasks that execute within its band. Tasks locking at higher bands or non-band priority levels are scheduled directly by the base scheduler for the duration of their execution outside their band.

To understand this we note that a band task τ of band B_i (B_τ=B_i), locking a resource r at a higher band k or a higher non-band priority level k (k>i+3), sets a system ceiling $\bar{\pi} = \lceil r \rceil \geq apl\_table[k] > apl\_table[i+4]-1$, which is higher than the absolute preemption level of any task below the locking band or locking non-band priority level. Now, if τ does not suspend, it can obviously not be preempted by any task below the locking band or below the non-band priority that it is locking at. But even if τ does suspend, no lower task will be able to lock a resource, since it will fail the FMSF_PLT.

The above observation has two consequences. The first is that τ is the only task outside band k or priority level k that can be locking within this band or priority level. Secondly, no task from B_τ will be able to lock outside the band. Instead, they will block at the appropriate priority as specified by priority inheritance. Therefore, there can never be the case that two or more tasks from the same band can be executing outside their band. This, in turn, means that there can never be execution eligibility contention between tasks of the same band while executing outside the band. Therefore, the application scheduler will never be asked to provide an execution eligibility decision on a task executing outside a band. Instead, the base scheduler will schedule the task when it becomes the oldest highest priority runnable task in the system.

More formally we can say that:

- At any one time, there can only be one task not belonging to a particular band or non-band priority level that is locking resources at that band or non-priority level.
- An application scheduler does not need to keep track of *where* a task goes once it locks a resource outside the band.

That is not to say, however, that a task's own scheduler should never be informed of the scheduling operations the task makes outside its band, as the scheduler's policy might be taking into account some of these operations. For example, a policy might be interested in the amount of blocking time a task has had, which means that it would have to know when the task blocks, even if this happens outside its band. Therefore, we choose to always notify a task's application scheduler of scheduling operations taken by the task outside its band, even though such notification might be disregarded by the scheduler.

Observations on eligibility inheritance

Now that the resource sharing protocol of the framework is defined, some important observations can be made on its eligibility inheritance. The first is the following proposition concerning the application of eligibility inheritance.

Proposition 4.1: Eligibility inheritance can be applied *only* within the band the system ceiling belongs to, and *only* when the task blocking on the system ceiling belongs to this same band.

Proof: If the system ceiling does not belong to a band (i.e. locking takes place at a non-band priority level) there can be, by default, no notion of eligibility inheritance.

If the system ceiling does belong to a band, this will be the highest locking band. Since the system ceiling is only lowered upon unlocking a resource, blocking on the system ceiling will always be due to the highest locked resource. Therefore, eligibility inheritance needs to make sure that the task locking this resource, which will be executing in the highest locking band, will finish without unnecessary preemptions. The only tasks that could cause unwanted preemption are tasks of the highest locking band, since tasks in higher bands are by default more eligible to run and tasks in lower bands cannot preempt. Consequently, the only band that can perform this eligibility inheritance is the highest locking band. QED ■

Furthermore, we can deduce a simple way for application schedulers to keep track of locking within their band, based on the following simple observation. Because of the FMSF_PLT, *the task that set the ceiling will be the last task to have locked a resource.* This is easy to understand, if we consider that no task τ can lock a resource, unless $|\pi|(\tau) > \overline{\pi}$, or τ is the task that set the system ceiling. If τ is the latest task in the system to lock a resource, say r, it will be $\lceil r \rceil \geq |\pi|(\tau)$ and, therefore, also $\lceil r \rceil > \overline{\pi}$. From this it follows that, upon locking r, the system ceiling must be updated. In the case where τ was already locking

a resource before locking *r*, it is evident that it will continue to be the task whose locking operation set the system ceiling.

By applying this observation to tasks locking within their band we conclude that the last task to do so is also that task which is currently holding the highest preemption ceiling locked resource. Therefore, a conceptual list where the last band task to lock within the band is placed at the head, is enough for an application scheduler to easily apply eligibility inheritance. Every time the application scheduler is informed that one of its tasks has blocked while trying to lock a resource within the band, it will know which locking task to check for eligibility inheritance, namely the first task on the list. Due to the FMSF_PLT, multiple entries of the same task, locking different resources in a nested way, will be grouped together, as shown in Figure 4.8.

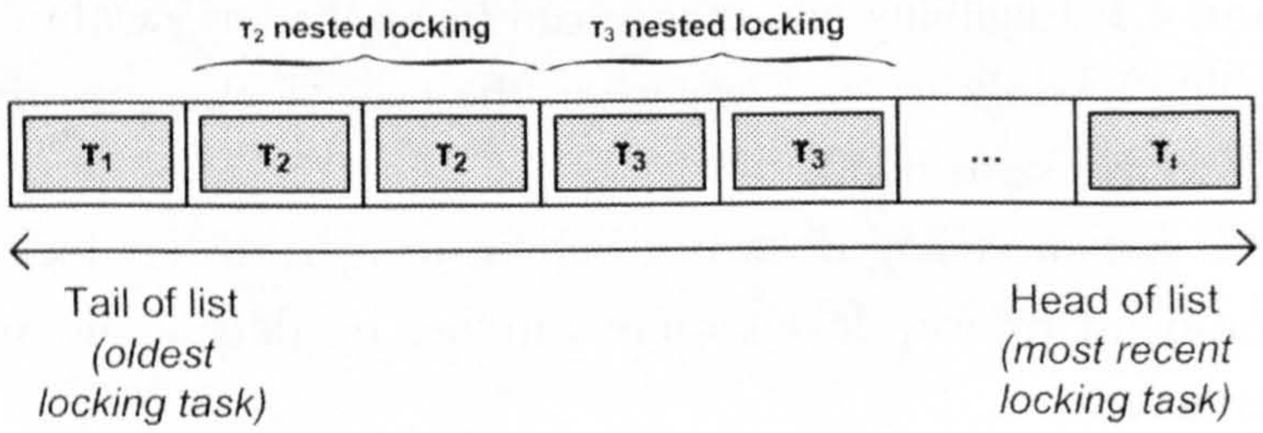


Figure 4.8: An application scheduler's conceptual locking list

Use of such a list implies that the application scheduler does not need to know anything about the ceilings of the locked resources. In other words, locking is transparent to application schedulers, which only need to know that a task is locking within the scheduler's band.

Implementation of priority inheritance

Ideally, the implementation of the framework should guarantee that, when a task is to be placed at a non-band priority level upon unlocking a resource, it will be placed at the front of the conceptual queue for that priority level. This is the usual behaviour for priority-based resource sharing protocols and its purpose is to preserve the FIFO ordering of the queue by not allowing other tasks in the queue to preempt the unlocking task. The FIFO ordering can be seen as a policy for deciding the eligibility of tasks of the same priority. So, in this sense, placing a task at the front of the queue when unlocking has the effect of not allowing lower eligibility tasks to preempt.

This can be seen in Figure 4.9 below. In this figure we can see three tasks, τ_L, τ_M, τ_H. Tasks τ_M and τ_H belong to the same non-band priority level k, while

τ_L can belong to either a lower priority level or a lower band. Furthermore, τ_L and τ_H share a resource r_1, while τ_L also uses r_2, which has a higher preemption ceiling than r_1. τ_L initially holds both resources, while τ_M and τ_H are on queue k, with τ_H in front of τ_M.

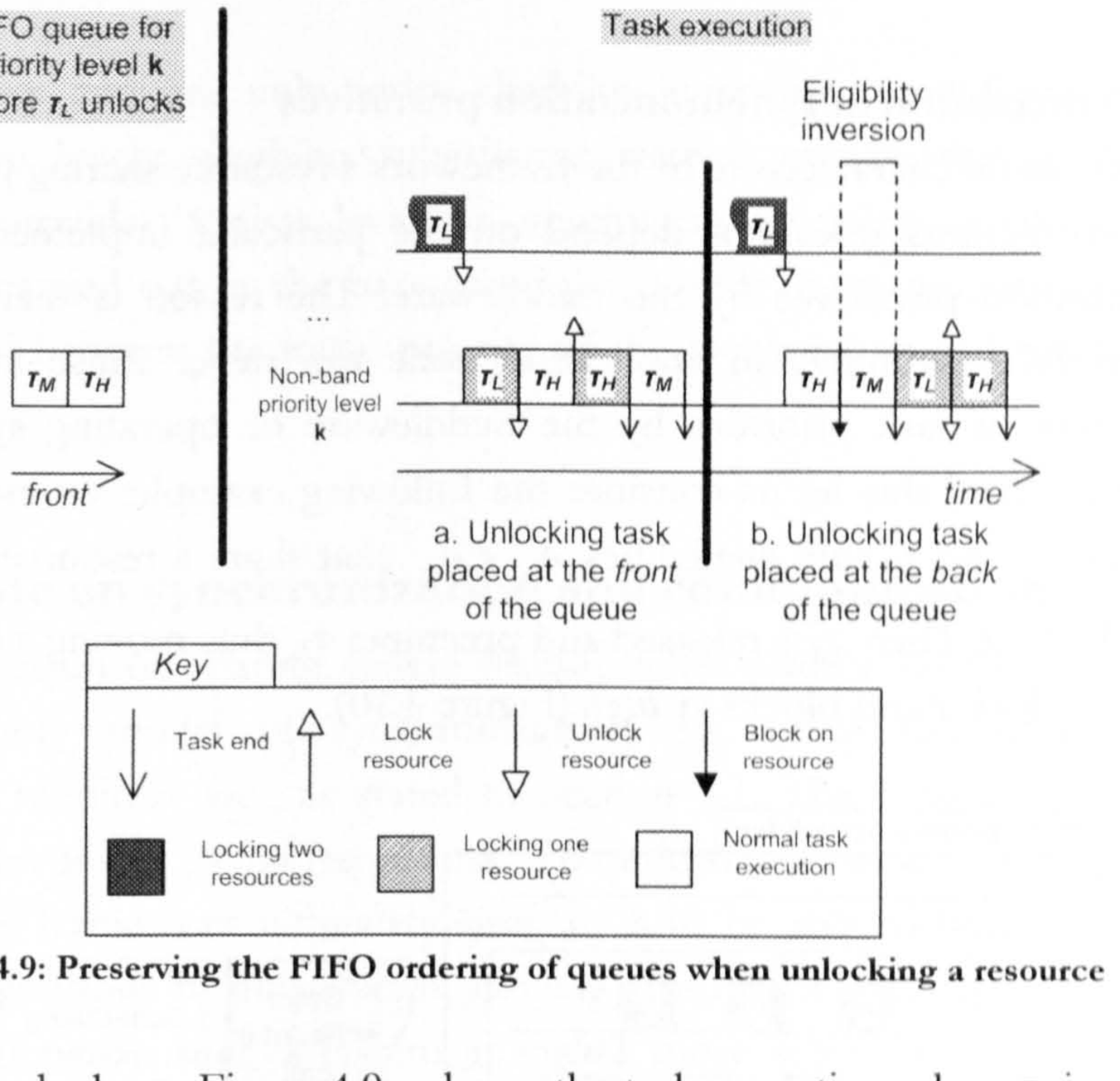


Figure 4.9: Preserving the FIFO ordering of queues when unlocking a resource

Then τ_L unlocks r_2. Figure 4.9.a shows the task execution when τ_L is placed at the front of queue k. τ_L proceeds to unlock r_1 and is immediately preempted by τ_H, which locks r_1. Then τ_H finishes and τ_M executes last. Figure 4.9.b shows the eligibility inversion that occurs, if τ_L goes to the back of the queue. τ_H runs first and blocks on r_1. τ_M is next on the queue and runs, introducing eligibility inversion. After τ_M finishes, τ_L runs and unlocks the resource, and finally τ_H gets to lock r_1.

Despite this situation, eligibility inversion of this sort is not as severe as is inversion by lower priority tasks. The FIFO ordering of tasks at the same level is not supposed to be an exact method for determining eligibility. For example, if it was the case that τ_M was released a fraction earlier than τ_H, there would not be such a problem. Therefore, we do not consider adding τ_L at the back of the queue a breach of the protocol.

Nested locking of resources

We observe that the framework allows nested locking of resources thanks to the FMSF_PLT. The task does not have to be executing in its own band in

order to lock a resource. The base scheduler, knowing the ceiling of the resource to be locked, can decide how to perform priority inheritance on the task irrespective of its current priority. The only restriction is that critical sections must be properly nested, i.e. unlocking of resources should happen in the reverse order of locking them.

Implementation of synchronization primitives

A very useful characteristic of the framework's resource sharing protocol is that its correctness does not depend on the particular implementation of synchronization primitives by the middleware. The reason is that use of a synchronization primitive in application code can never cause inadvertent manipulation of task priorities by the middleware or operating system. To better understand this let us consider the following example: we assume two band tasks, τ_1 and τ_2 with eligibilities $e_{\tau_1} < e_{\tau_2}$ that share a resource *r*. τ_1 runs first and locks *r*. Then τ_2 is released and preempts τ_1, thus moving τ_1 to *low*. τ_2 then tries to lock *r* and blocks at *high* (Figure 4.10).

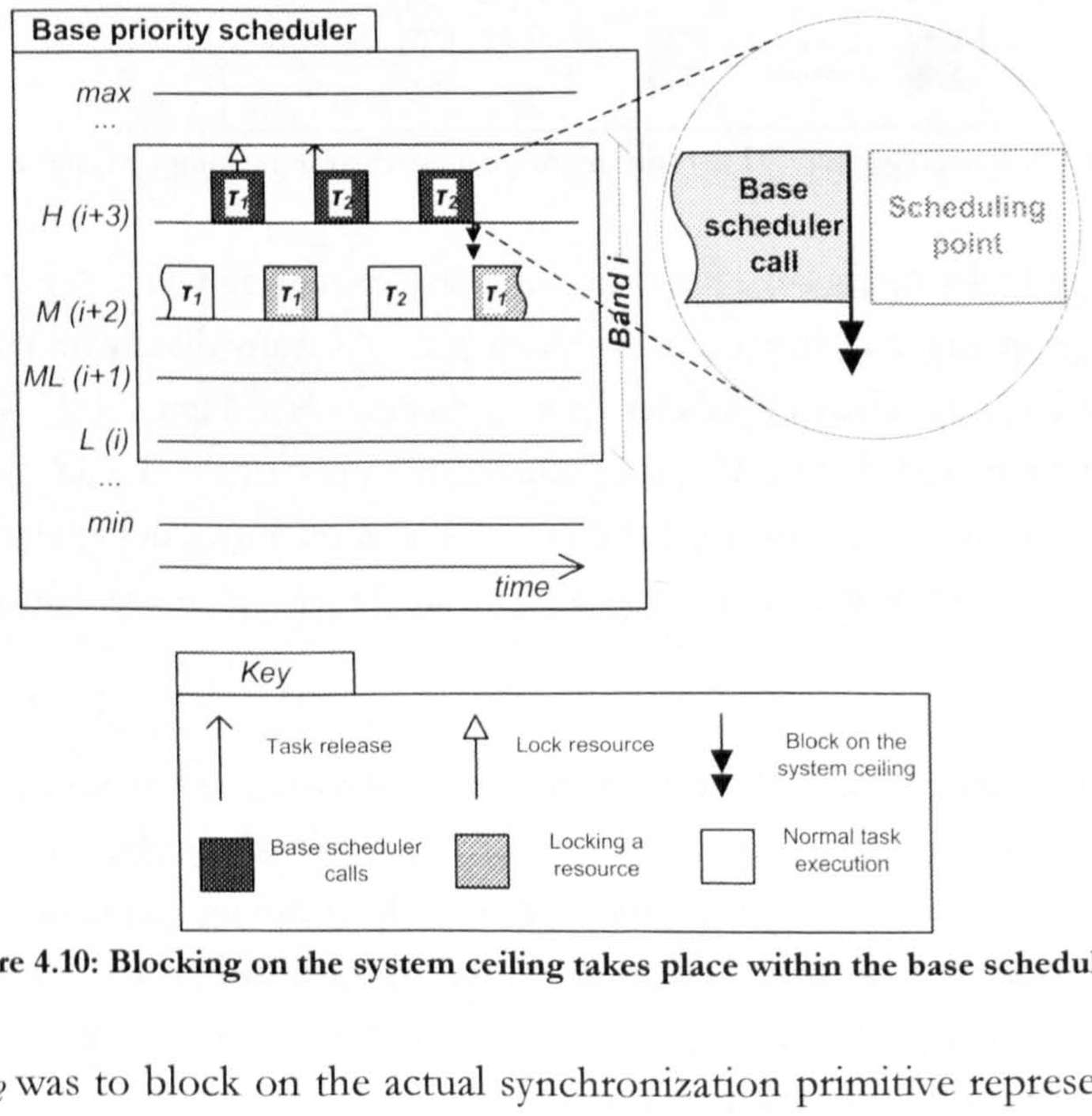


Figure 4.10: Blocking on the system ceiling takes place within the base scheduler call

If τ_2 was to block on the actual synchronization primitive representing the shared resource, the middleware (or operating system, depending on the implementation), assuming that the default resource sharing protocol is priority inheritance, would have to elevate the priority of τ_1 to *high*. This would constitute a breach of the protocol. However, the use of the system

ceiling in the protocol prevents such a case. τ_2 blocks within the preceding base scheduler call and not at the scheduling point, as can be seen in Figure 4.10. The scheduling point, which is the call to a locking primitive in the middleware library, will be executed *only* when τ_2's preemption level is greater than the system ceiling, i.e. when the resource has been unlocked.

To sum up, avoiding unbounded eligibility inversion in our framework is done on two levels: eligibility inheritance, carried out by the application schedulers, provides a first level of minimising inversion, while priority inheritance, carried out by the base scheduler, avoids the inversion caused by tasks situated between the initial priority of the locking task and the locking band.

4.5.6 A note on synchronization and communication

The discussion on sharing system resources has centred on the use of the shared memory model of synchronization, e.g. using shared variables, semaphores, monitors etc., as stated in Section 4.1. The reason is that this model is better suited to the framework. To understand this we must consider that what the framework ultimately aims at, is to be able to house different types of applications in one system, ranging from hard to soft, to non-real-time applications. In such a system, it makes more sense to assume a large number of tasks, with rapid task creation and deletion, rather than a static number of tasks with predefined communication paths, and according to [Lauer and Needham 1979], more dynamic systems tend to be better implemented on the shared memory model.

However, message passing can still be supported by our framework. To demonstrate this, let us assume two band tasks, τ_1 and τ_2, communicating using the asynchronous type of message passing, which is the most basic variant for message passing [Burns and Wellings 2001; p.284]. More specifically, they use two message transmission operations: `send()` and `wait()`. Calling `wait()` must be modelled as a suspension operation, but `send()` need not be, since it is asynchronous. At some point τ_2 executes `wait(message)`, which means executing the preceding base scheduler call and then blocking at the actual scheduling point, at priority *high*. Later, τ_1 passes a message to τ_2 by executing `send(message)`. τ_1 does not execute a scheduling band operation, does not block, but continues with its execution at *medium*. However, τ_2 is now unblocked and since its priority is higher than τ_1,

it preempts it and executes the succeeding base scheduler call for `wait()`, which decides if τ_2 can preempt τ_1.

Using the asynchronous variation of message passing we can construct both the synchronous and remote invocation variations [Burns and Wellings 2001; p.284]. Therefore, message passing can be fully supported in our framework, if needed. Additionally, it has been shown in [Lauer and Needham 1979] that the two communication techniques are "duals of each other" and that a system which is constructed according to one model has a direct counterpart in the other. This means that we do not lose any expressivity by supporting only the shared memory model, and that any system which is based on message passing can be faithfully constructed using shared memory techniques.

4.6 The Flexible Middleware Scheduling Protocol (FMSF)

Having described its constituent parts, this section presents the protocol in its entirety. The protocol is, in essence, a two-level scheduling mechanism. There are four basic rules on which the definition of the protocol relies, and these are given below:

A. As part of any scheduling operation, the task's application scheduler is always informed of the operation. Rule A will always take place in an operation before Rules C and D.
B. The first action of a *pbsc* of any scheduling operation, apart from a *lock*, is to elevate the calling task's priority to the *high* priority of the band in which the task is executing, if it is executing in one. If the task is executing at a non-band priority level, the task priority remains unchanged.
   The *lock* operation moves the executing task to the *high* priority of the band to which the resource belongs to, if that is the case. If the resource belongs to a non-band priority level, $\overrightarrow{loc(\tau, r)}.pbsc$ sets the task's priority to that level.
C. During the *pbsc* part of any *suspending* scheduling operation, a task shall always inquire the next most eligible task of its own and current band's application scheduler, if the two bands are different.

D. When executing the *sbsc* of a scheduling operation at its own band's high priority, a task τ shall always ask its application scheduler to perform an eligibility test. If τ is no longer the highest eligibility task in the band, it is put to $p_L(B_\tau)$, otherwise it is put to $p_M(B_\tau)$. In the latter case, if there was another task previously running at $p_M(B_\tau)$, that task is put to $p_L(B_\tau)$.

Based on these four rules, the framework's protocol is given below, describing the steps taken during each of the scheduling operations.

1. Immediately after the release of task τ the $\overrightarrow{rel(\tau)}$ operation is performed.
   1.1. Task τ is released.
      1.1.1. If τ belongs to band B_τ, it is released at the *high* priority level of its band ($p(\tau)=p_H(B_\tau)$).
      1.1.2. If τ is not a band task, it is released at its assigned (non-band) base priority.
   1.2. $\overrightarrow{rel(\tau)}.sbsc$ does the following:
      1.2.1. If τ is a band task, it informs $S(B_\tau)$ of the release and asks it to perform an eligibility test, as in Rule D.
2. A task τ can perform $loc(\tau,r)$ at any point during its execution. The protocol does the following:
   2.1. The following steps are followed in $\overrightarrow{loc(\tau,r)}.pbsc$:
      2.1.1. The task is moved to the appropriate priority.
         2.1.1.1. If the resource ceiling belongs to a band, the task is moved to the *high* priority of the locking band ($p_H(B_r^{loc})$). If $B_r^{loc} \neq B_\tau^{cur}$, there is no need to first move the task to $p_H(B_\tau^{cur})$.
         2.1.1.2. If the resource ceiling belongs to a non-band level, the task is moved to that priority level.
      2.1.2. If the resource is governed by the FMSF protocol, the base scheduler continues with 2.1.3 to apply the FMSF_PLT. If the resource is governed by some other resource sharing protocol, the base scheduler, depending on the programmer's choice, can continue with 2.1.3 or go to 2.1.5.
      2.1.3. If $|\pi|(\tau) \leq \overline{\pi}$, then the task blocks against the current system ceiling until $|\pi|(\tau) > \overline{\pi}$, at which point the protocol continues with 2.1.4.

2.1.3.1. The base scheduler informs τ's application scheduler of the blocking.

2.1.3.1.1. If $B_\tau = B_r^{loc}$, then if the blocked task is more eligible than the oldest highest execution eligibility task in B_r^{loc}, the band's scheduler must perform eligibility inheritance based on the blocked task.

2.1.3.1.2. If τ locks at a higher band $B_r^{loc} > B_\tau$, or if it locks at a non-band level, no eligibility inheritance need take place.

2.1.3.2. The base scheduler asks τ's scheduler for its most eligible task, which it places at $p_M(B_\tau)$.

2.1.4. When $|\pi|(\tau) > \overline{\pi}$, the base scheduler updates the system ceiling ($\overline{\pi} = \lceil r \rceil$).

2.1.5. The base scheduler informs $S(B_\tau)$ of the locking.

2.1.6. If τ is locking within a higher band $B_r^{loc} > B_\tau$, the base scheduler also informs $S(B_r^{loc})$.

2.2. The actual resource locking takes place.

2.3. The $\overrightarrow{loc(\tau, r)}.sbsc$ modifies the task's priority appropriately:

2.3.1. If $B_\tau = B_r^{loc}$, the task's application scheduler is asked to perform an eligibility check, as in Rule D.

2.3.2. If $B_r^{loc} > B_\tau$, the base scheduler puts τ at $p_{ML}(B_r^{loc})$.

2.3.3. If τ is locking at a non-band level, its priority remains unchanged.

3. A task τ can perform $\overrightarrow{unL(\tau, r)}$ only after having performed the corresponding $\overrightarrow{loc(\tau, r)}$ operation. The protocol does the following:

3.1. $\overrightarrow{unL(\tau, r)}.pbsc$ takes the following steps:

3.1.1. If the task is executing within a band, it is moved to $p_H(B_r^{loc})$. Otherwise, the task's priority remains unaltered.

3.2. The task unlocks the resource.

3.3. The $\overrightarrow{unL(\tau, r)}.sbsc$ does the following:

3.3.1. The base scheduler informs $S(B_\tau)$ of the unlocking.

3.3.2. If $B_r^{loc} > B_\tau$, the base scheduler also informs $S(B_r^{loc})$ of the unlocking operation.

3.3.3. If the resource is governed by the FMSF protocol, the system ceiling is modified accordingly by removing the resource ceiling

from the system ceiling list, even if r was not the highest locked resource, and all tasks waiting on the unlocked resource are notified.

3.3.4. The base scheduler moves the task to the appropriate priority.

3.3.4.1. If the task does not hold any other resources, or if the next highest resource it holds belongs to its own band, the base scheduler moves the task to $p_{II}(B_\tau)$ and asks the task's application scheduler to perform an eligibility test, as described in Rule D.

3.3.4.2. If the next highest resource r' the task holds belongs to a higher band than its own band, the task's priority is set to $p_{ML}(B_{r'}^{loc})$.

3.3.4.3. If the next highest resource the task holds belongs to a non-band level, the task's priority is set to that priority level.

3.3.4.4. If the task is a non-band task and holds no other resources, it is moved to its own priority.

4. A *wait* operation $\overrightarrow{wt(\tau,r,cv)}$ can *only* be performed between a $\overrightarrow{loc(\tau,r)}$ operation and its corresponding $\overrightarrow{unL(\tau,r)}$ operation. The protocol does the following:

4.1. The $wt(\tau,r,cv).pbsc$ does the following:

4.1.1. If the task is executing within a band, it is moved to $p_{II}(B_\tau^{cur})$. Otherwise, the task's priority remains unaltered.

4.1.2. The system ceiling is modified accordingly.

4.1.2.1. If r was the highest locked resource, then the head of the conceptual system ceiling queue is removed.

4.1.2.2. If r was not the highest locked resource, the list link corresponding to the resource ceiling is removed.

4.1.3. The base scheduler informs τ's application scheduler of the wait.

4.1.4. The base scheduler asks the application scheduler of τ for its most eligible task τ', if any, and places τ' at $p_M(B_\tau)$.

4.1.5. If $B_\tau^{cur} \neq B_\tau$, it asks the application scheduler of B_τ^{cur} for its most eligible task τ'' and moves τ'' to $p_M(B_\tau^{cur})$.

4.2. The task performs the wait and blocks.

4.3. The $\overrightarrow{wt(\tau,r,cv)}.sbsc$ does the following:

4.3.1. The base scheduler updates the system ceiling ($\overline{\pi} = \lceil r \rceil$).

4.3.2. If the locking task is in its own band, the base scheduler informs the task's application scheduler of the locking.

4.3.3. The base scheduler modifies the task's priority appropriately.

4.3.3.1. If $B_\tau^{cur} = B_\tau$, the base scheduler asks the task's application scheduler to perform an eligibility test, as in Rule D.

4.3.3.2. If $B_\tau^{cur} > B_\tau$, it is placed at $p_{ML}(B_\tau^{cur})$.

4.3.3.3. If the task is at a non-band priority, it continues to run at the same priority.

5. A task τ can perform $\overrightarrow{sus(\tau)}$ at any point during its execution.

5.1. $\overrightarrow{sus(\tau)}.pbsc$ does the following:

5.1.1. The task is moved to $p_H(B_\tau^{cur})$, if executing within a band; otherwise its priority remains unchanged.

5.1.2. If the base scheduler can determine that the particular scheduling point, at this particular execution, will not cause the task to suspend, it simply lets the task proceed with the scheduling point.

5.1.3. If there is a possibility that the task might suspend at the scheduling point, the base scheduler does the following:

5.1.3.1. It informs the task's application scheduler of the suspension.

5.1.3.2. It asks the application scheduler of B_τ for its most eligible task τ', if any, and places τ' at $p_M(B_\tau)$.

5.1.3.3. If $B_\tau^{cur} \neq B_\tau$, it asks the application scheduler of B_τ^{cur} for its most eligible task τ'', if any, and moves τ'' to $p_M(B_\tau^{cur})$.

5.2. The task executes the scheduling point and it either suspends or not.

5.3. $\overrightarrow{sus(\tau)}.sbsc$ does the following:

5.3.1. The base scheduler informs $S(B_\tau)$ of the resumption.

5.3.2. If $B_\tau^{cur} = B_\tau$, the base scheduler asks the task's application scheduler to perform an eligibility test, as in Rule D.

5.3.3. If $B_\tau^{cur} > B_\tau$, $S(B_\tau^{cur})$ is notified of the resumption and the task is placed at $p_{ML}(B_\tau^{cur})$.

5.3.4. If the task is at a non-band priority, it continues to run at the same priority.

6. A task τ can perform $\overrightarrow{yld(\tau)}$ at any point during its execution. The protocol does the following:

6.1. $\overrightarrow{yld(\tau)}.pbsc$ performs the following:

6.1.1. If and only if $B_\tau^{cur} = B_\tau$, the task is moved to $p_H(B_\tau)$.

6.2. The task executes the yield primitive.

6.3. $\overrightarrow{yld(\tau)}.sbsc$ performs the following:

6.3.1. If and only if $B_\tau^{cur} = B_\tau$, the following is done:

6.3.1.1. The task's application scheduler is informed of the yield.

6.3.1.2. The task's application scheduler is asked for its most eligible task τ', which is placed at $p_M(B_\tau)$. If $\tau' \neq \tau$, τ is placed at $p_L(B_\tau)$.

7. The $\overrightarrow{chg(\tau)}$ operation on τ can be performed at any time by any task τ_c of the *same* band, i.e. $B_{\tau_c} = B_\tau$. Additionally, only a task itself can change its own band, and this can occur only if the task is not holding any resources.

7.1. If $B_{\tau_c}^{cur} = B_{\tau_c}$, $\overrightarrow{chg(\tau)}.pbsc$ moves τ_c to $p_H(B_{\tau_c})$, otherwise its priority remains unchanged.

7.2. The change in τ's scheduling parameters takes place.

7.3. $\overrightarrow{chg(\tau)}.sbsc$ does the following:

7.3.1. The application scheduler $S(B_\tau)$ is informed of the change in τ's scheduling parameters.

7.3.1.1. If $\tau_c = \tau$ and the operation changed τ_c's own band to another band $B_{\tau_c}^{new}$, then:

7.3.1.1.1. $S(B_{\tau_c}^{new})$ is informed of the operation as well.

7.3.1.1.2. $S(B_{\tau_c}^{old})$ is asked for its next most eligible task, which is moved to $p_M(B_{\tau_c}^{old})$.

7.3.1.1.3. The priority of τ_c is changed to $p_H(B_{\tau_c}^{new})$.

7.3.2. If $B_{\tau_c}^{cur} = B_{\tau_c}$ then $S(\tau_c)$ is asked to perform an eligibility test, as per Rule D.

8. A task τ can successfully perform $\overrightarrow{end(\tau)}$ *only* when it has executed a corresponding $\overrightarrow{unL(\tau, r)}$ operation *for every* $\overrightarrow{loc(\tau, r)}$ operation it has performed. The base scheduler does the following:

8.1. $\overrightarrow{end(\tau)}.pbsc$ moves τ to $p_H(B_\tau)$.

8.2. The base scheduler queries the task's own band for its most eligible task, and places the returned task, if any, at $p_M(B_\tau)$.

8.3. The task terminates.

As a comment we can say that the base scheduler needs to keep the system ceiling, $\bar{\pi}$, conceptually as an ordered linked list, so that it can accommodate nested locking. Now that the framework's protocol has been described, an example is provided to help in better illustrating the way the framework behaves. Let there be four tasks, such that τ_1 belongs to band 1 and is released at $t_1=0$, task τ_2 is a non-band task with priority 5, released at $t_2=2$, task τ_3 belongs to band 6 and is released at $t_3=6$, and task τ_4 belongs to band 10 and is released at $t_4=4$. Furthermore, let us assume they share two resources: tasks τ_1 and τ_4 share r_1, and tasks τ_2 and τ_4 share r_2. Figure 4.11 shows the schedule produced by the framework protocol.

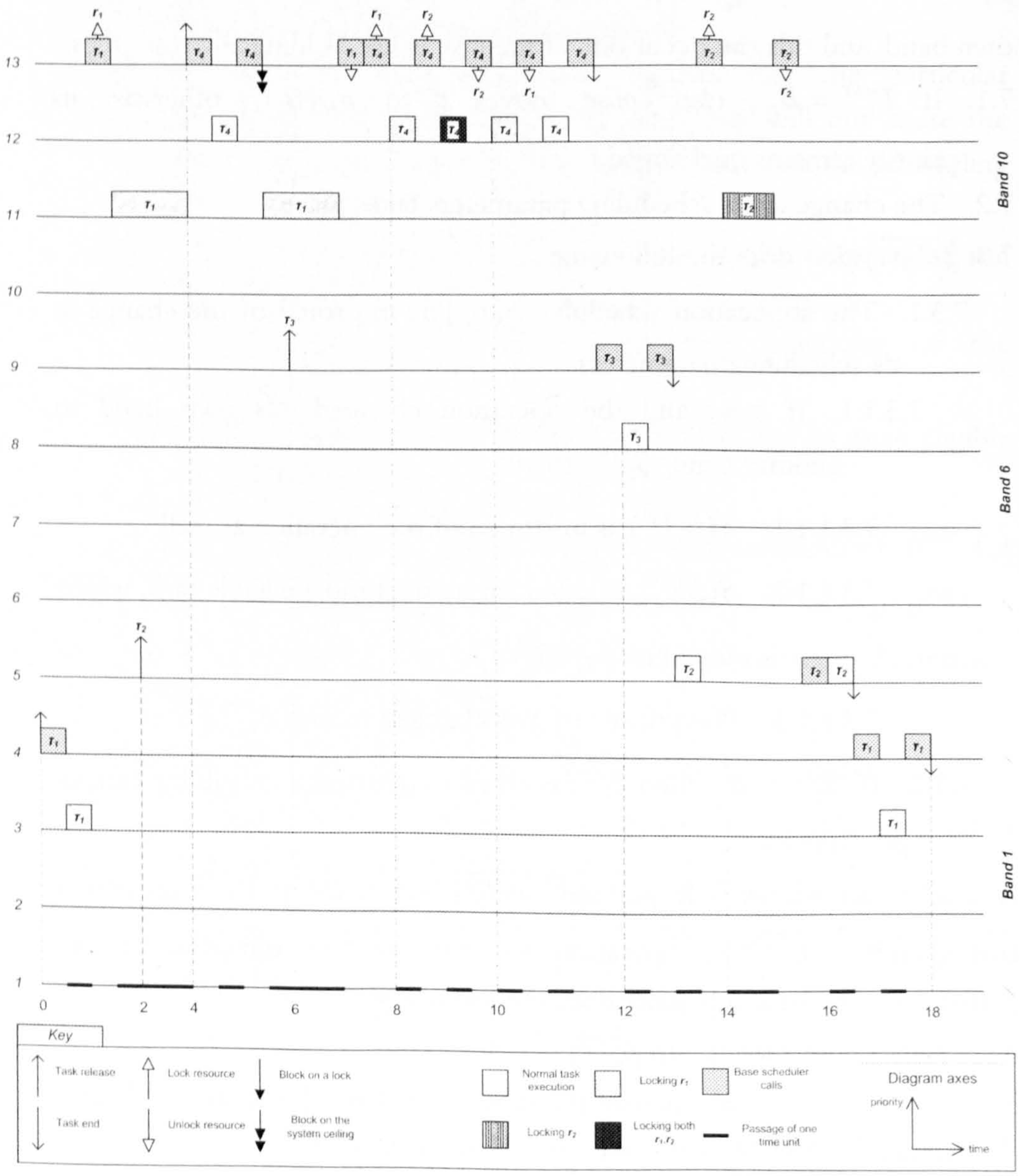


Figure 4.11: An example of a schedule of four tasks running within the framework

The figure shows the execution of the four tasks on the base scheduler priority queues. Although scheduling operations are a distinct part of a task's execution, their execution cost is negligible compared to the task's total execution cost. Therefore, the passage of time, shown on the X axis as a thick black line, does not take scheduling operations into account. Initially, task τ_1 is released at its band's *high* priority, as per paragraph 1.1.1 of the protocol. After an eligibility test, as per paragraph 1.2.1 and Rule D, it is moved to *medium* where it executes, until it decides to lock resource r_1. It is then moved to the *high* priority of band 10, as per paragraph 2.1.1.1, where it locks the resource and is then promptly moved to *medium_lock* priority (priority 11) as per paragraph 2.3.2. While executing at priority 11, task τ_2 is released at priority 5, but cannot preempt. Then task τ_4 is released at priority $p_H(B_{10})=13$, preempting τ_1. After an eligibility test, as per paragraph 1.2.1 and Rule D, it is moved to *medium* (priority 12) where it executes for a while before deciding to lock r_2. However, it blocks due to the FMSF_PLT, as per paragraph 2.1.3, and τ_1 continues to run. Meanwhile, task τ_3 is released at $p_H(B_6)=9$, as per 1.1.1, but cannot preempt. Then, task τ_1 is again moved to $p_H(B_{10})$ to unlock τ_1, as per 3.1.1. It is then moved to its own band's *high* priority, as per 3.3.4.1, and is immediately preempted by τ_4, which promptly locks r_1 and is moved to *medium*, as per 2.3.1. Then, τ_4 successively locks r_2, unlocks r_2 and r_1, and proceeds to end its release at priority *high*, as per 8.1. The base scheduler then selects the next most eligible task, which is τ_3 at priority 9. τ_3 executes its release operation, is moved to its *medium* priority, as per 1.2.1, and then finishes its run again at $p_H(B_6)=9$. The next most eligible task is τ_2, which executes at non-band priority level 5. After executing for a while it decides to lock r_2 and is moved to $p_H(B_{10})=13$, according to paragraph 2.1.1.1. After locking, the task executes at the locking band's *medium_lock* level, as per 2.3.2, and then unlocks the resource. This takes it back to its own priority level, as per 3.3.4.4, where it finishes its execution. Finally, task τ_1 is selected and gets to finish its *unlock* operation. It is moved to *medium*, executes for a while and then executes an *end* operation.

4.6.1 Protocol properties

Three are the main protocol properties that guarantee its correctness and are given below:

- Resource usage under the protocol is deadlock-free.

- Blocking of tasks is reduced to the duration of one critical section from a lower eligibility task, *if and only if* tasks do not self-suspend.
- The framework always selects the oldest, highest effective eligibility, runnable task in the system to execute.

Following, we will prove each one of these properties.

Deadlock avoidance: *Under the FMSF protocol there can be no deadlock.*

Proof: Liu [2000: p.313] mentions that the BPreCP is deadlock-free and that it bounds eligibility inversion to, at most, one critical section of a lower eligibility task. Additionally, Proposition 3.3 and Corollary 3.1 have also proved that the BPreCP is deadlock-free even when self-suspension while locking is allowed and even when tasks unlock resources with lower ceiling than the system ceiling. The BPreCP_PLT part of the protocol was adequate to prove these properties, and the same is indicated in [Liu 2000: p.313]. Since the framework's FMSF_PLT is exactly the same as the BPreCP_PLT, it follows that the framework will also be deadlock-free. ■

Bounded eligibility inversion: *Under the FMSF protocol no task can be blocked for longer than the duration of one outermost critical section of a lower eligibility task.*

Proof: We base our proof on the equivalent proof for the PCP protocol, provided in [Sha et al. 1990]. Suppose a task τ_H is blocked by a lower eligibility task τ_L. This means that when τ_H was released, τ_L was already locking a shared resource *r*. We can distinguish four situations: i) τ_H and τ_L are both non-band tasks, ii) τ_H is a non-band task, but τ_L is a band task, iii) τ_H is a band task, but τ_L is a non-band task, and iv) τ_H and τ_L are both band tasks.

In i), it is assumed that *r* is also used by at least one band task. According to the framework, τ_L will be running at the priority of τ_H at the time when τ_H is released. Therefore, τ_H will not be able to preempt it and the protocol will work like the IPCP. As soon as τ_L unlocks the resource, it loses the inherited priority and τ_H preempts it. τ_L will not be able to lock another resource and block τ_H.

Situation ii) is effectively the same as i), because they both take place at $p(\tau_H)$ in exactly the same manner, irrespective of where task τ_L comes from.

In iii), τ_L will be at priority $p_{ML}(B_{\tau_H})$ at the time when τ_H is blocked (at priority $p_H(B_{\tau_H})$). In order for τ_L to have locked within B_{τ_H}, no band task could have been locking a resource prior to τ_L locking *r*. Furthermore, as

shown in Section 4.5.5, τ_L will be the only task not belonging to B_{τ_H} that is locking within the band. Also, no band task could have locked another resource prior to τ_H being released, otherwise τ_H would not have blocked on r. Before blocking, τ_H will have already executed for some time, preempting τ_L, since, by default, its eligibility is higher. τ_H will block due to a $\overrightarrow{loc(\tau_H, r)}$ operation. $S(B_{\tau_H})$ will be informed of the blocking and will keep the eligibility of τ_H as the highest eligibility of a pending task in the band. Any band task τ_M being released after τ_H blocks will trigger, by default, an eligibility test by $S(B_{\tau_H})$ as part of its $\overrightarrow{rel(\tau_M)}$ operation. If its eligibility is not higher than the highest pending eligibility, τ_M will be moved to $p_L(B_{\tau_H})$ and, therefore, eligibility inversion will not occur. When τ_L performs the $\overrightarrow{unL(\tau_L, r)}$, it will wake τ_H up. τ_H will run at its priority ($p_H(B_{\tau_H})$), preempting τ_L. τ_L will not be able to block τ_H again. When all tasks in B_{τ_H} finish execution, τ_L will be selected to execute, it will finish its *unlock* operation and move back to its initial priority level.

Situation iv) is practically the same as iii), because it again leads to eligibility inheritance within the band, thus preventing any medium eligibility task τ_M from delaying τ_H.

Therefore, we have proved that τ_H can be blocked by a particular lower eligibility task for the duration of its longest critical section, at the most.

Now let us suppose that τ_H can be blocked by more than one lower eligibility task, each time for the duration of that task's longest critical section. Suppose that the first two lower eligibility tasks to block τ_H are τ_{L1} and τ_{L2}. If τ_{L1} and τ_{L2} belong to bands or non-band priority levels below B_{τ_H}, then, by default, they will have absolute preemption levels that are lower than that of τ_H. If either or both τ_{L1} and τ_{L2} belong to the same band as τ_H then, because of their lower eligibility, they will have been assigned lower relative preemption levels that again translate to lower absolute preemption levels than τ_H.

In order for both tasks to block τ_H they must both be locking a shared resource when τ_H is released, say r_1 and r_2 respectively. Let us assume that τ_{L2} has the lowest eligibility and was the one to lock a shared resource first. It would have set the system ceiling to $\bar{\pi} = \lceil r_2 \rceil \geq |\pi|_{\tau_{L2}}$. Under the FMSF_PLT, in order for τ_{L1} to lock r_1, it must be that $|\pi|_{\tau_{L1}} > \lceil r_2 \rceil \geq |\pi|_{\tau_{L2}}$. Since we assume that τ_H can be blocked by τ_{L2} it must be $|\pi|_{\tau_H} \leq \lceil r_2 \rceil < |\pi|_{\tau_{L1}}$. A contradiction,

since by assumption it is $|\pi|_{\tau_H} > |\pi|_{\tau_{L1}}$. Thus, it is impossible for task τ_H to have eligibility higher than both tasks τ_{L1} and τ_{L2} and to be blocked by both of them under the framework protocol. QED■

Highest effective eligibility dispatching: *The framework behaves as if the system's dispatcher is always selecting the runnable task with the oldest highest effective eligibility, provided that all possible types of scheduling points have been identified.*

Proof: First, we are going to prove the correct execution of tasks within a band, according to that band's policy. This, in effect, means guaranteeing the correct order of execution for the non-scheduling code of band tasks according to their eligibilities. The correctness of task execution is proved with the following six observations on the rules of the protocol.

Observation I: The application scheduler needs to provide a new scheduling decision, i.e. specify the next task to run at *medium* priority, every time the set $E=\{e(\tau_1),e(\tau_2),\ldots,e(\tau_i)\}$ of the eligibilities of *runnable* tasks changes. A scheduling point represents a potential change to this set. Given our assumptions that all scheduling points have been identified and that the framework addresses only event-triggered policies (as explained in Sections 4.1 and 4.2.2), the fact that it provides a scheduling operation for each scheduling point guarantees that a scheduling decision will be made by the application scheduler when needed.

Observation II: Rule D guarantees that the application scheduler will always make a scheduling decision after every scheduling point.

Observation III: Rule B guarantees that scheduling operations will be executed without interference from normal task execution. In addition, the rule guarantees the serialisation and, thus, integrity of the scheduling operations. Each operation will finish before another one is executed.

Observation IV: Rule A guarantees that a band's application scheduler will always be able to assess the effects of a scheduling point and keep the correct information on its tasks. Since changes in E can only happen at scheduling points and since Rule A always precedes Rules C and D, it follows that an application scheduler will always have an updated E set when taking a scheduling decision, thus guaranteeing the correctness of the decision.

Observation V: Rules B and D guarantee the preemptive execution of tasks, since any executing task running at *medium* can be preempted by a scheduling operation of another task running at *high*. Thus, the application scheduler can make a new scheduling decision whenever needed, possibly indicating a new task to execute.

Observation VI: Rules C and D guarantee that a task will always be running in the band, if one is available.

With the above observations we can show that the execution of scheduling operations leads to the desired behaviour within a band. Based on the definition of scheduling points we can infer that the execution of a task is nothing more than the execution of code segments, each segment contained within two scheduling points, with each task having at least one such code segment contained within its release and its termination. Section 4.3 has given the five basic scheduling operations corresponding to five basic scheduling points: *release*, *end*, *lock*, *unlock*, *suspend*. We will examine each one in turn.

- A band task is always released at its band's *high* priority. This, as in (V), guarantees that it will be able to preempt the running task, if needed.
- The *end* operation, due to observation (VI) above, always allows the application scheduler to select the next most eligible runnable task to execute in the band, if such a task exists.
- Similarly, the *pbsc* of a *suspend* operation, due to (VI), will allow the application scheduler to select the next most eligible runnable task to execute in the band, if such a task exists. The *sbsc* of the suspend operation, because of (V), will act exactly like the *sbsc* of the release operation, again guaranteeing the ability to preempt, if necessary.
- A *lock* operation is possibly suspending. Thus, in this respect it has the same characteristics as *suspend* with respect to (V) and (VI). Additionally, a *lock* operation, in conjunction with its corresponding *unlock* operation, enforces the FMSF_PLT and triggers the EI rule.
- The *unlock* operation, as already stated, enforces the FMSF_PLT, EI and FMSF_ET rules, and like any other operation, allows the application to enforce preemption, as per (V). This means that the loss of any inherited eligibility will indeed reflect on the execution of the unlocking task.

Therefore, we have shown that for all five basic scheduling operations, and thus for all scheduling operations in general, the framework protocol guarantees task execution within a band as if the application scheduler was the base scheduler. The only way for a band task to execute outside its band is to lock a resource with a preemption ceiling higher than the highest absolute preemption level assigned to the band. Priority inheritance enforces part of the FMSF protocol's eligibility inheritance. As has been shown in Section 4.5.5,

this task will be the only task executing outside the band and, thus, its application scheduler need not keep track of where it goes. Effectively, it has been proven that the application scheduler will not need to make a scheduling decision regarding the task, while its priority is outside the band. Also, as long as this band task executes outside the band, no other band task will be able to execute within the band due to the base scheduler's priority dispatching.

One consideration is the application scheduler of the locking band for a task that locks at a higher band. The FMSF Eligibility Inheritance rule (Section 4.5.3) specifies that the locking band's application scheduler will apply eligibility inheritance within its band, such that the locking task might acquire an effective eligibility. This will prevent eligibility inversion within the locking band. Tasks of the locking band with higher eligibilities than the effective eligibility of the locking task will be able to execute within or outside their band, as has been described.

Of the five basic operations only three are available to a task locking outside its band: *lock*, *unlock* and *suspend*. *release* is not applicable and *end* is not permitted while a task holds resources. Let us examine each of the three possible operations.

- A *lock* operation, performed while a task is already executing outside its own band, can only lock a resource r_2 with an even higher preemption ceiling than the already locked resource r_1. Therefore, the locking task is going to remain outside its own band. For every nested *lock* operation outside the band, Rule B applies. Therefore, the situation the task is going to be in, when executing its i^{th} nested *lock* operation outside its own band, is identical to that of the i^{th} resource being the first resource locked outside the task's own band. Without loss of generality, we can consider each nested locking outside the task's own band as the first locking outside the band. Therefore, performing a lock operation while a task is already executing outside its own band does not affect the correctness of the protocol.
- If an *unlock* operation is executed after the task has performed i nested lock operations outside its own band, then by unlocking its i^{th} locked resource it will be in a situation identical to that of locking the $(i-1)^{th}$ resource. Therefore, such an unlock operation does not affect the correctness of the protocol. If the task unlocks its last locked resource outside its band, then the *unL(τ,r).sbsc* is going to enforce preemption, if

necessary, as per observation (V). Execution will continue within the band as was described above.

- If the locking task suspends, the *pbsc* of the suspending operation will select the next most eligible task in its band to place on *medium*. This is so that other band tasks will get the opportunity to execute while the locking task is suspended, and also so that the first runnable task the dispatcher will find, if no runnable task exists above the band, will be at the *medium* priority and not the *low* priority of the band. As long as the task remains suspended, task execution within its band will proceed as described above. As soon as the task wakes up, it will, by default, preempt any band task that happens to be executing.

Finally, non-band tasks will be scheduled by the base scheduler according to their priority, as normal. Lock within a band will be treated the same as a lower band tasking locking at a higher band. The locking band does not distinguish between a lower band task and a non-band task locking at *medium_lock*.

Thus, for all possible situations, we have demonstrated that the framework protocol manages to enforce highest effective eligibility dispatching. QED■

4.7 Summary

This chapter presented a scheduling framework that adds support for flexible scheduling in real-time middleware. Its primary benefits are:

1. Its basic assumptions are simple and widely used.
2. It is designed to support arbitrary scheduling policies.
3. Its protocol supports resource sharing between tasks of different policies, guaranteeing deadlock avoidance and bounded eligibility inversion.
4. The protocol is deadlock-free even when tasks suspend while holding resources.
5. The protocol properties have been demonstrated with examples and proven.

In subsequent chapters a multi-faceted evaluation of the framework takes place, which further strengthens its proposal and demonstrates its viability.

Chapter 5

Framework Evaluation

Chapter 4 described a framework for achieving flexible scheduling in real-time middleware. This chapter provides a three-faceted evaluation of the protocol. The first part of the evaluation is the description of an application-defined scheduler used in conjunction with the framework. This essentially is a case study of how an arbitrary scheduler can be incorporated in the framework. Our choice is an EDF scheduler, which has gained significant support over the past several years as a valuable supplement to fixed-priority scheduling [Buttazzo 2005]. Then, a verification of the framework's operation is provided, which is achieved by modelling it as a system of timed automata in the UPPAAL model checker. This provides a strong indication that a correct implementation of the protocol is possible, by testing a number of characteristic cases. Finally, we present an examination of the types of different scheduling policies that can be supported by the framework. The extended range of policies covered can satisfy a plethora of application demands and diverse real-time needs. This demonstrates the full scope of the protocol.

Combining the three types of evaluation instills more confidence into our assertion that the Flexible Middleware Scheduling Framework is a viable and appropriate solution for flexible real-time scheduling in middleware.

5.1 An EDF application scheduler

The protocol presented in the previous chapter is the core element of a flexible middleware scheduling framework, but, on its own, is not enough to evaluate the approach. A crucial element, and indeed the reason for which the framework was created, is the presence of an application scheduler. This section describes how an EDF application scheduler could be implemented for use by the framework.

Implementing a basic EDF scheduler is fairly straightforward. To enforce the policy it is sufficient for the scheduler to keep a queue ordered by absolute deadline, where the head of the queue holds the task with the nearest absolute deadline while the back of the queue is the task whose deadline is furthest away in the future. It is important to note that, as a direct consequence of the observation on FMSF_PLT in Section 4.5.5, only band tasks executing within the band need to be placed in this queue.

However, the scheduler must also be able to cope with resource sharing within its band. In doing this, the EDF scheduler can implement a protocol of choice, but if it is to share resources with other schedulers it needs to support the FMSF_EI. It is the second case that is going to be examined here. Therefore, the scheduler must conform to the FMSF by performing the eligibility inheritance part of the protocol, as specified in Chapter 4. In light of this, the simple EDF queue model must be enhanced, so that the scheduler can easily apply eligibility inheritance when needed.

As stated in Rule 4.1, eligibility inheritance will need to be applied when a task blocks while executing within its own band. What makes eligibility inheritance easy to implement is the observation on the manner of resource locking, made in Section 4.5.5. By keeping a list of locking tasks within the band, like the one described in 4.5.5, the application scheduler can store within every link in the list the eligibility which the corresponding locking task had at the point of locking. If a band task τ_b blocks within the band due to resource sharing, the scheduler compares its eligibility against the active eligibility of the

first task in the locking list, τ_l. The active eligibility of τ_l is increased, if τ_b has higher execution eligibility. When τ_l unlocks the resource, its eligibility is restored to the value stored in the link and the link is removed from the list. Figure 5.X shows such a conceptual list, where $d_n(\tau_j)$ is the absolute deadline task τ_j had before locking its n^{th} nested resource.

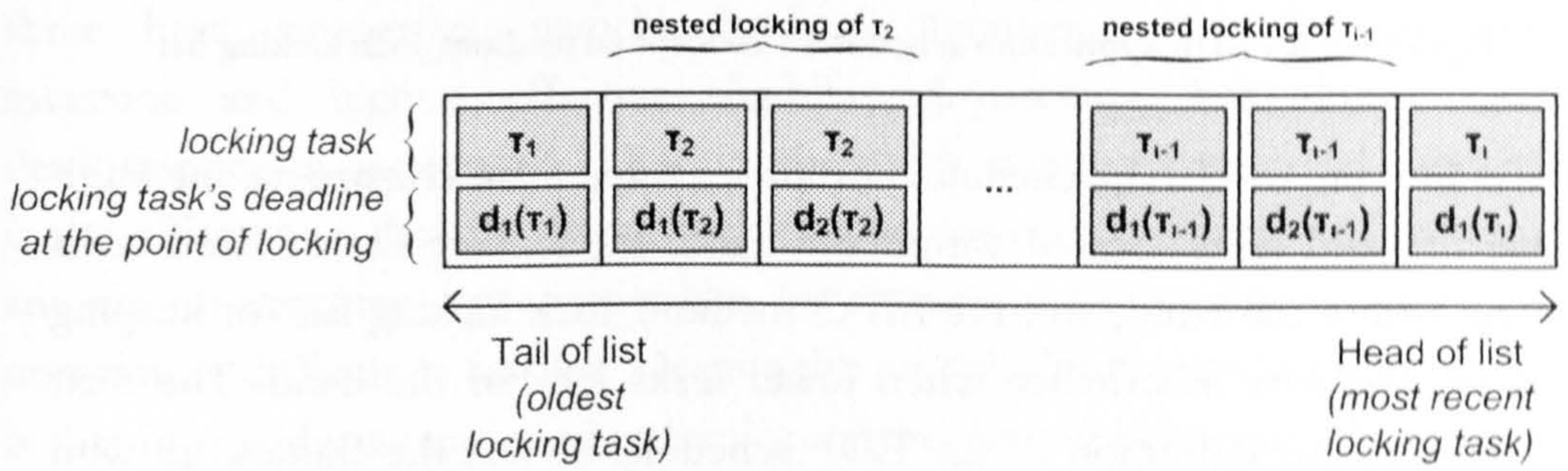


Figure 5.1: An EDF application scheduler's conceptual locking list

Note that no resource related information needs to be kept in the list. Furthermore, in removing links from this list the scheduler is not restricted to removing only the head of the list, but can remove the entry corresponding to the highest locked resource of any task currently locking. For example, in Figure 5.1 above the head of the list represents the highest locking resource for task τ_j and also the highest locked resource in the scheduler. The second link in the list represents the highest locked resource for task τ_{j-1} and, therefore, can be removed from the list, if τ_j suspends and τ_{j-1} executes and reaches the end of its innermost critical section.

The scheduler also has to cater for the situation where the locking task τ_L is of a lower band or fixed-priority level, as described in Section 4.5.3.2. Again, as explained in Section 4.5.5, because of the FMSF_PLT, at any one time there can only be one lower task using band resources and executing at the *medium_lock* level. However, because τ_L does not belong to the band, eligibility inheritance is not applicable to it and, hence, the locking queue is not suitable for keeping track of its locking history. Instead, the scheduler keeps a separate LIFO list for the task that happens to be executing at *medium_lock*. Every link in this list corresponds to a separate band resource that has been locked by τ_L in a nested way and holds the highest eligibility amongst band tasks that have blocked within the band due to τ_L. Each time one of these resources is unlocked, the corresponding link will be removed from the list. This list is shown in Figure 5.2.

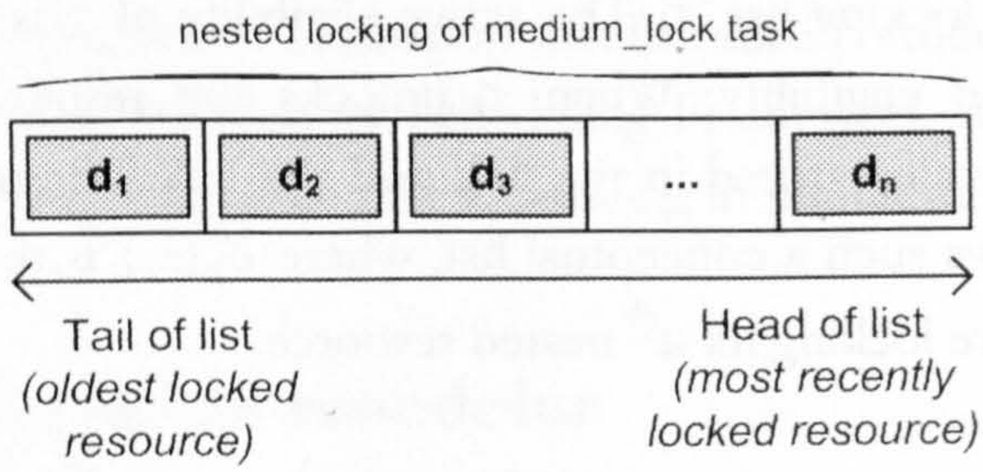


Figure 5.2: An EDF application scheduler's conceptual medium_lock locking list

To sum up, the EDF scheduler comprises three main constructs: the EDF queue of band tasks located within the band, the locking list of band tasks locking within the band, and the LIFO medium_lock locking list for keeping track of eligibility inheritance when lower tasks lock in the band. The next section uses this definition of the EDF scheduler to test the framework with model checking. What is more important, though, is the fact that these constructs are applicable to any application scheduler, regardless of its policy, since they are based on the general observations made in Section 4.5.5. The actual implementation of an EDF application-defined scheduler will be given in the next chapter, together with the framework implementation.

5.2 Modelling the framework

The formal verification presented in this section is based on *model checking*. Model checking is a method for algorithmically verifying formal systems. In order to apply the method, a system has to be modelled as a directed graph of vertices and edges. The set of vertices represents the state of the system at any time while the set of edges represents the set of possible transitions the system can take, going from one state to another. Given this model, model checking is simply an exhaustive search of the state space in order to see if a given formal specification is satisfied. However, as can be understood, this search is possible only when the model itself is finite. If the system that the model is representing is also finite then model checking can provide conclusive proof of the correctness of a property. When, however, the system to be modelled is not finite, any attempt to model it must, by default, limit the initial dimensions of the system.

This is true for the FMSF framework. Although the framework itself is finite, the possible situations under which it can be used are infinite. For example, theoretically we cannot put an upper limit on the number of tasks in the system, and the same is true for the number of resources and the number

of bands in the system. Furthermore, we cannot restrict the way tasks use resources, therefore the combinations of task resource usage are also infinite. Therefore, in building a model of the framework we must restrict these parameters. However, in the case of the FMSF framework we deem these limitations acceptable. The reason is that we have no desire of proving the correctness of the framework through model checking. The correctness of its three basic properties, namely deadlock avoidance, bounded eligibility inversion and highest effective eligibility dispatching, has already been demonstrated in Section 4.6.1. Rather, the result sought from model checking is the affirmation that the framework can be correctly implemented, without any run-time emergent properties like, for example, race conditions, which are notoriously difficult to test for. Despite the model's limitations, its importance is that it provides a very good indication of the desired behaviour. Essentially, it is a test bed for the framework, which has considerable advantages over a concrete implementation, such as:

- Rapid development of the model
- Ease of forming and trying different test cases
- Use of the model as the design specification of an implementation

In this case, model checking was done using the UPPAAL model checker. The following section briefly describes this tool, while Sections 5.2.2 to 5.2.9 describe the model. There are two application schedulers used, a fixed-priority scheduler and an EDF scheduler based on the description in Section 5.1. In the model, tasks are referred to as threads, because the model is seen as an abstraction of an actual implementation, where tasks are implemented as threads.

5.2.1 The UPPAAL tool

In this section we will give a brief introduction to the modelling tool UPPAAL. This is a freely available model checker based on the theory of timed automata. A timed automaton is a finite state machine extended with clock variables. UPPAAL defines a modelling language that extends timed automata with, amongst others: **constants; bounded integer variables** with which we can perform arithmetic operations; **binary synchronization channels,** where a transition labelled with `c!` synchronizes with only one (out of possibly multiple) transition labelled `c?`; **broadcast channels,** where one sender `c!` can synchronize with an arbitrary number of receivers `c?`; **urgent**

locations, where time is not allowed to pass when the system is in such a location; **committed locations**, where time is not allowed to pass *and* the next transition in the system must involve an outgoing transition from one of the committed locations; multi-dimensional **arrays**.

To express requirement specifications, UPPAAL has a query language that consists of **state formulae** and **path formulae**. A state formula translates to an individual state, a state being the set of the locations of all automata, all clock values and the values of all discrete variables. A path formula quantifies over paths in the model. Path formulae are classified into *reachability* ("can a particular state be reached?"), *safety* ("something will never happen") and *liveness* ("something will eventually happen"). To express path formulae we use the syntax `[A|E] ["[]"|"<>"]` φ, where φ is a state formula. `A` denotes that a given property should hold for all paths in the system. `E` denotes that there should be at least one path. `"[]"` denotes that all states in the path should satisfy the property, while `"<>"` denotes that at least one state in the path satisfies the property. So, for example, (`A[]` φ) means that invariantly φ should hold. UPPAAL offers the keyword `deadlock` to describe the state where no outgoing transitions are possible. The reader is referred to [Behrmann et al. 2004] and [Bengtsson and Yi 2004] for more information on UPPAAL and timed automata in general.

5.2.2 Architecture Description

In order to test our framework and evaluate its behaviour we have implemented it as a collection of timed automata in UPPAAL [Zerzelidis and Wellings 2006b]. In building our model we have identified 4 basic automata which are: Thread, PriQueue, BaseScheduler, Dispatcher. The model is rounded out with the EDFScheduler automaton, which helps demonstrate the ability of our framework to accommodate different application level schedulers. Our model's architecture can be seen in Figure 5.3. API calls (whether middleware or OS calls) are simulated with synchronization channels between automata, which can be seen as arrows in the figure, going from the automaton initiating the synchronization to the one receiving it. The channels on the arrows are just examples.

The Thread automaton represents a thread in the system. More than one instance of Thread can be specified and different threads can belong to different application schedulers (the dashed lines in Figure 5.3 between threads

and application schedulers demonstrate which scheduler each thread belongs to). PriQueue models the functionality associated with a base scheduler FIFO priority queue. Naturally, there can be more than one instance of PriQueue. The BaseScheduler automaton represents the middleware priority scheduler, which has the priority queues under its control.

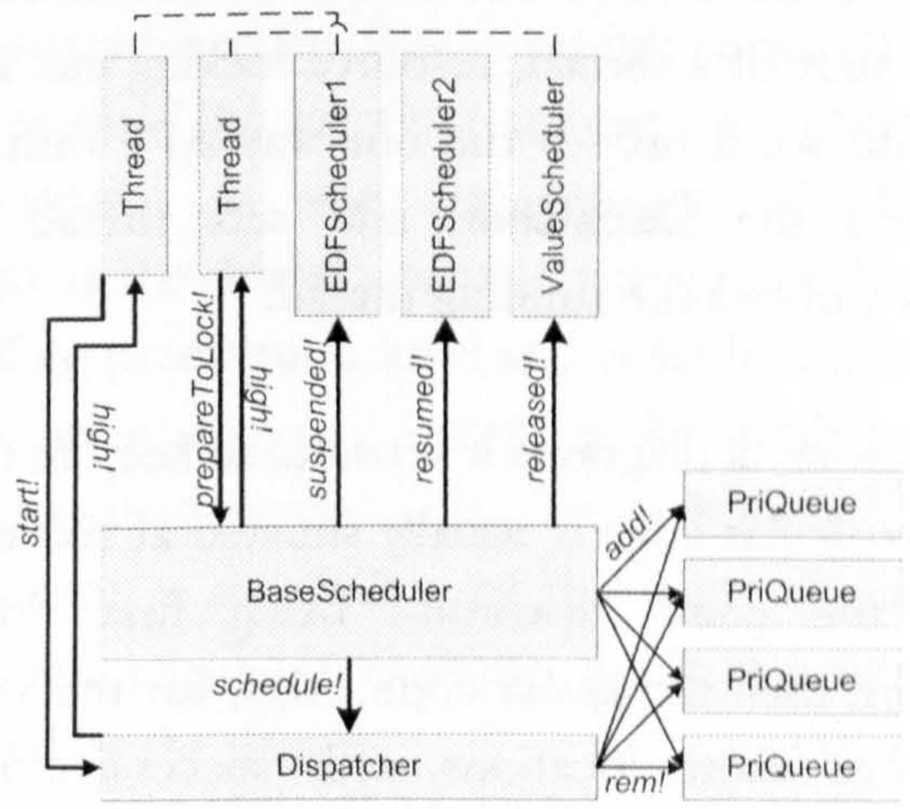


Figure 5.3: Model architecture

The Dispatcher automaton models the dispatching mechanism. It, too, interacts with the PriQueues, and is, in principle, part of the base scheduler, but has been modelled separately for clarity. Finally, the EDFScheduler automaton represents an application-defined EDF scheduler, with its private variables and data structures (e.g. internal queues). It contains only the basic parts of an EDF scheduler, those necessary to the model. The EDFScheduler automaton presents a special case since it is "pluggable". We can replace it, or use it in conjunction with other application scheduler automata, as long as we keep the same interface, i.e. the same synchronization channels. This is shown in Figure 5.3 with a ValueScheduler automaton. We can also have multiple instances of a particular application scheduler automaton.

It is worth pointing out that none of the automatons in the model makes use of clock variables. The reason for this is that the model does not represent the implementation of a particular system with known deadlines and costs. It is a model of a theoretical framework and our primary concern is the detection of race conditions, rather than meeting some kind of deadline or measuring execution costs. The only variable used to represent the passage of time is the integer `runtime` in the Thread automaton. This variable is incremented to simulate execution time passing for a particular Thread instance. Its practical use is to provide a limit to how long a thread can go on self-suspending or

locking an object. Thus, it guarantees that a Thread automaton will eventually end its run.

Every transition in the model is triggered as part of a chain of transitions that originate from the Thread automaton. In the figure we can see that the BaseScheduler plays a central role in our model. Its functionality is triggered by thread actions, e.g. start of a thread, resource locking etc. It then manages thread execution. To do so, it utilises the one-way API with the application defined schedulers and the Dispatcher, and sets thread priorities. The Dispatcher then selects and sets the running thread.

In describing the automata diagrams it is useful to keep in mind that all the code pertaining to a particular edge is usually situated at some point *above* the transition line, with the *guard* expression being first, followed by the *synchronization channel* and then the *update* code. Also, for the sake of brevity, a path containing four or more locations with successive numbering, e.g. $S22 \rightarrow S23 \rightarrow S24 \rightarrow S25$, will be written as $S22 \rightarrow ... \rightarrow S25$. This is different from $S22 \rightarrow S25$, which implies a transition on an edge directly linking locations $S22$ and $S25$.

5.2.3 Global declarations

The global declarations section of the model contains elements that can be used by any automaton. There are three main types of declarations: constants, variables and synchronization channels.

The constants are further subdivided into three categories: constants that specify maximum values (`MAX_THREAD`, `MAX_PRIORITY`, `MAX_PRELEVEL`, `MAX_RES`, `MAX_LOCK`), constants that are used for indexing the `thread[][]` array (`PERIOD=0`, `ADEAD=1`, `RDEAD=2`, `WAITING=3`, `RFH=4`, `PRIORITY=5`, `BAND=6`, `PRELEVEL=7`, `APL=8`, `ITER=9`, `COST=10`, `LAST_LOCKED=11`, `LOCKED=12`), and constants used to set the value of `thread[][RFH]` by specifying the reason a thread is at location *High*. `MAX_THREAD` is the maximum possible number of threads in the system, `MAX_PRIORITY` is the maximum number of priorities, `MAX_PRELEVEL` the maximum number of preemption levels per band, `MAX_RES` the maximum number of resources in the system, and `MAX_LOCK` the maximum number resources used by a thread.

The main global variable in the model is the **`thread[][]`** array, which contains all thread parameters, e.g. row `thread[1][]` contains all parameters

of thread 1 etc. Using the indexing constants we can access the different parameters for a thread. For example, to get the relative preemption level for thread 1 we access `thread[1][PRELEVEL]`. Column `thread[][PERIOD]` contains the thread periods. A thread's relative deadline is stored in column `thread[][RDEAD]`, while the absolute deadline is column `thread[][ADEAD]` and is set by the EDFScheduler. Column `thread[][WAITING]` indicates whether and why a thread is at the *Wait* location; `thread[][RFH]` indicates the reason a thread is at the *High* location; `thread[][PRIORITY]` is the thread's priority; `thread[][BAND]` is the *low* priority of the thread's band; `thread[][PRELEVEL]` is the thread's relative preemption level; `thread[][APL]` is the thread's absolute preemption level and is set by BaseScheduler during a *reschedule?* Synchronization; `thread[][ITER]` is the number of times the thread will execute; `thread[][COST]` is a thread's execution cost; `thread[][LAST_LOCKED]` is the ceiling of the latest resource the thread has locked (zero if none); `thread[][LOCKED]` holds the number of resources simultaneously locked by the thread.

The third set of constants, specifying the possible values that column `thread[][RFH]` can take, is:

- `AT_REL`, the thread has been released,
- `AT_BLOCK`, the thread is executing *prepareToSuspend?* in order to suspend itself,
- `AT_UNBLOCK`, the thread is executing *reschedule_resume?*,
- `AT_BLOCKLOCK`, the thread has to execute *prepareToLock?* again after being unblocked,
- `AT_LOCK`, the thread is executing *prepareToLock?*,
- `AT_RESLOCK`, the thread is executing *reschedule_lock?*,
- `AT_RESUNLOCK`, the thread is executing *reschedule_unlock?*,
- `AT_END`, the thread is executing *prepareToSuspend?* in order to end.

The **`bands[]`** array has as many cells as are priority levels and, if a priority level falls within a band, the corresponding cell contains the *low* priority of the band; otherwise it is zero.

The **`res[][]`** array contains the resources available in the model. The first row `res[0][]` contains the ceilings of the resources in the same format as `mutex`. The format is *xxxy*, where *xxx* is the ceiling and *y* is the low priority of the band (this clearly only works for bands with a *low* priority of 1 to 9 but its

enough for the purposes of our model). The second row `res[1][]` contains the lock for each resource, e.g. `res[1][3]`=1 means that resource 3 is locked, a value of 0 means it is unlocked.

Next is the two-dimensional array **`t_res[][]`**. This array contains the indexes of the resources (as they are allocated in the `res[][]` array) to be locked by each thread. Each row of `t_res[][]` contains the set of resources used by a different thread. For example, if `t_res[x][0]=1` this means that thread `x` will lock resource `res[0][1]`. Apart from specifying the resources, though, it also provides a schedule for locking the resources. That is, thread `x` will first lock `t_res[x][0]`, then it will lock `t_res[x][1]` in a nested way, and so on. However, the number of resources actually locked in each run is random, e.g. on one run thread `x` might lock only the first resource, while on another run it might perform a nested lock of the first two, and so on. Therefore, we are not specifying one particular scenario through the use of this array.

The last seven shared variables are used for communication between automata, much like parameters in a procedure call. **`run_thread`** holds the thread that is currently running. It is set and unset by the Dispatcher and used in transition guards by the Thread automatons to distinguish which one is currently executing. **`cur_thread`** is used by Thread for synchronization with the BaseScheduler and Dispatcher automata, to specify which thread performed the last synchronization with the BaseScheduler or Dispatcher. **`app_thread1`** and **`app_thread2`** are used to emulate parameter passing when the BaseScheduler synchronises with an application scheduler on one of its channels e.g. *isEligible!*. **`cur_queue`** is used by the BaseScheduler and Dispatcher, for synchronization with the PriQueues, and contains the id of the queue that must respond to a given synchronisation. **`cur_band`** is used between the BaseScheduler and the application scheduler automata and contains the id of the scheduler that must respond to a given synchronisation. **`mutex`** is used between the Thread and BaseScheduler automata and contains the ceiling of the resource to be locked or unlocked, in the form described for the `res[][]` array. .

Finally, there are a number of synchronization channels used between automata to guide the execution flow. These are divided according to the automaton which acts upon them. Unless stated otherwise, a channel is *not* a broadcast channel. For the rest of this chapter we will refer to a

synchronization channel by writing its name in *italics* followed by a `'!'` or `'?'`, depending on the way it appears in the automaton under question.

Dispatcher automaton: *start?* is equivalent to a thread's `start()` method; *schedule?* is called by the BaseScheduler whenever there is reason to re-select the running thread.

Thread automaton: *high?*, *medium?*, *medium_lock?*, *low?* and *fp?* tell Thread to go to the respective location, which signifies that the thread has acquired the corresponding priority.

BaseScheduler automaton: *reschedule?* is the equivalent of the framework's $\overrightarrow{rel(\tau)}.sbsc$ operation, as defined in Section 4.3; *prepareToLock?* is the equivalent of $\overrightarrow{loc(\tau,r)}.pbsc$; *reschedule_lock?* is the equivalent of $\overrightarrow{loc(\tau,r)}.sbsc$; *prepareToUnlock?* is the equivalent of $\overrightarrow{unL(\tau,r)}.pbsc$; *reschedule_unlock?* is the equivalent of $\overrightarrow{unL(\tau,r)}.sbsc$; depending on the situation, *prepareToSuspend?* is the equivalent of either $\overrightarrow{sus(\tau)}.pbsc$ or $\overrightarrow{end(\tau)}.pbsc$; *reschedule_resume?* is the equivalent of $\overrightarrow{sus(\tau)}.sbsc$.

EDFScheduler automaton: With a *released?*, *blocked?*, *suspended?* or *resumed?* synchronization the scheduler is informed that `app_thread1` has been respectively released, blocked, has suspended, or resumed after suspending, and takes appropriate action. *suspended?* in particular is a broadcast channel that at the same time instructs Thread to go to the *Suspended* location. *isEligible?* asks the scheduler to decide if `app_thread1` is eligible to execute. The scheduler sets `app_thread1` to the most eligible thread in the band and also sets `app_thread2` to the formerly executing thread in the band, if any. With *getMostEligible?* the scheduler places its current most eligible thread in `app_thread1` and also sets this thread as the thread executing at its *medium*. *lockedResource?* and *unlockedResource?* inform the scheduler that one of its threads has respectively locked or unlocked a resource, so that it can appropriately adjust its internal queues.

5.2.4 Thread

Local declarations: `cost` is `thread[][COST]`. `period` is `thread[][PERIOD]`. `rt` is `run_thread`. `ct` is `cur_thread`. `sus` is `thread[][RFH]`. `w` indicates whether a waiting thread should go to *High* or *Fp* once notified. `lc` counts the

number of times the thread has self-suspended. It is used to limit the number of times a thread can suspend, in order to restrict the state space and, thus, make the model more testable. `id` is the id number of the thread, which is also the index of the thread in the `thread[][]` array. `runtime` is the amount of CPU time consumed. `iterations` is the number of releases the thread has had. `locked` is `thread[][LOCKED]`. It counts the number of resources currently locked by the thread and is also used as an index for the `t_res[][]` array that contains the indexes of the resources to be locked. So, for example, when `locked` is 0, meaning that no resources have been locked, it points to the first resource to be locked, `t_res[id][0]`.

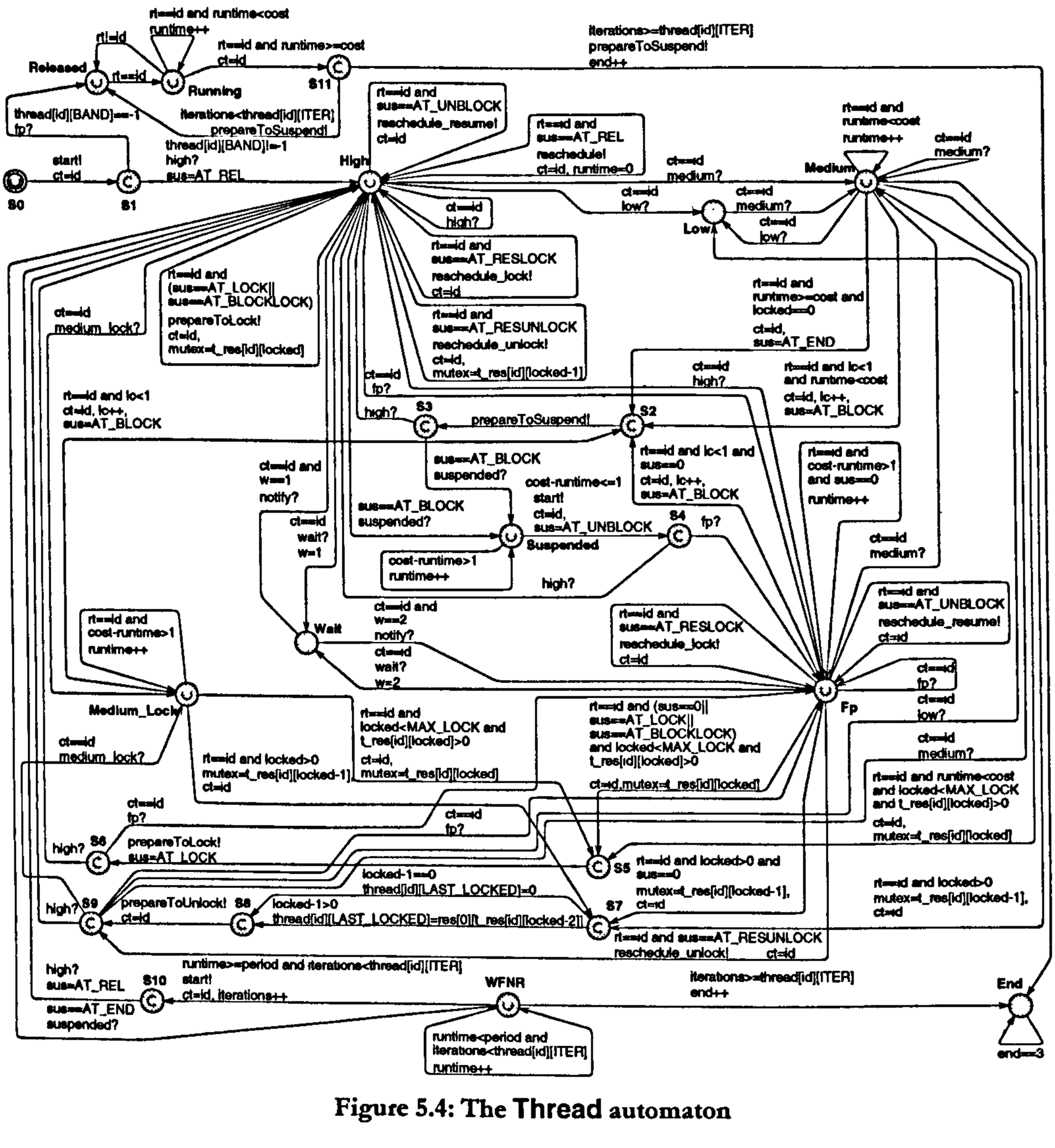


Figure 5.4: The Thread automaton

Description: The Thread automaton describes an implementation of both a band and a non-band task. However, due to a need to restrict the state space, we have limited this description in certain aspects. First, only band threads can perform scheduling operations other than *release* and *end*. Essentially, non-

band threads are modelled to only start, execute and finish. Secondly, the only other operations modelled for band threads are *lock*, *unlock* and *suspend*. These limitations have made our model testable, and yet they have not taken away from the ability of the model to test the principle of different threads executing different scheduling operations in various combinations, for reasons that will be presented in the BaseScheduler automaton.

Modelling a non-band thread consists of two main locations, *Released* and *Running*, which are self-explanatory. In the case of a band thread, there are eight main locations, *High*, *Medium*, *Medium_Lock*, *Low*, *Fp*, *Suspended*, *Wait* and *WFNR*. The first four correspond to the four priority levels of a scheduling band. The *Fp* location is reached when a band thread locks a resource at a fixed-priority level. Being at *High*, *Medium*, *Medium_Lock* or *Fp* means that the thread is either runnable or running. Being at *Low* means that the thread is runnable but not running. Being at *Suspended* signifies that the thread has suspended itself and, thus, lies outside the priority queues, waiting to resume. A thread is permitted to suspend both while it is holding resources and when it is not. The *Wait* location is reached whenever a thread blocks due to the system ceiling. There it waits to be notified that the resource it needed has been released. The wait-for-next-release (*WFNR*) location is reached whenever a thread finishes its current release. From there it can start a new release, or terminate. The two locations completing the set of primary thread locations are *S0* and *End*. *S0* is the initial location, where the thread has not started yet, while *End* is the final location in a thread's execution path, where the thread has terminated.

All primary locations, except *End*, *Low* and *Wait*, are urgent, which means that when a thread automaton is in one of them it has to take a valid outgoing transition without delay. This is to guarantee that the system will eventually progress. There are also a number of secondary locations, when moving between primary locations, which are transitory. Their importance lies with the transitions between them rather than with the locations themselves. All secondary locations are committed.

Generally speaking, transitions that start from one of the *High*, *Medium*, *Medium_Lock* and *Fp* locations are meant to be taken when the thread is running. These transitions might be the first in a path that starts from one of these locations and ends at another one of them, e.g. the path $Fp \rightarrow S5 \rightarrow S6 \rightarrow High$. As a rule, the first transition in such a path contains

the guard `rt==id`, which guarantees that only the automaton representing the running thread will take the transition. Similarly, transitions that receive synchronisations from other automatons (e.g. *Wait* → *High*) will, in most cases, have the guard `ct==id`, to guarantee that the correct thread receives the synchronisation. The automaton can be seen in Figure 5.4.

First, the available transition choices of the Thread automaton will be presented for each of its main locations. Then we will describe how a thread can initiate each of the three main scheduling operations supported, namely *lock*, *unlock* and *suspend*.

The first transition $S0 \rightarrow S1$ takes place when the thread is started. At *S1* we determine the type of thread. If it is a non-band thread (band=-1), it receives an *fp?* synchronization and is taken to *Released* and rotates between *Released* and *Running* until it has had a specified number of releases, at which point it goes to *End*. If it is a band thread it is taken to *High*. From there the thread calls for a system reschedule, by taking one of a number of cyclic transitions $High \rightarrow High$. From *High* it is taken to either *Medium* or *Low*, depending on whether the thread is the most eligible to run or not. From *Low* the only possible transition is $Low \rightarrow Medium$, which occurs when the thread is selected as the most eligible to run.

While at *Medium*, six transitions are possible: i) the thread could get preempted by a higher thread in the band and moved to *Low* ($Medium \rightarrow Low$), ii) it can execute normally, taking transition $Medium \rightarrow Medium$, during which the `runtime` local variable is incremented by 1, iii) it can try to lock a resource following the $Medium \rightarrow S5 \rightarrow S6 \rightarrow High$ path, iv) it can unlock a resource taking the $Medium \rightarrow S7 \rightarrow S8 \rightarrow S9 \rightarrow High$ path, v) it can suspend itself ($Medium \rightarrow S2 \rightarrow S3 \rightarrow High \rightarrow Suspended$), regardless of whether it is currently locking resources, and finally, vi) it can terminate via $Medium \rightarrow S2 \rightarrow S3 \rightarrow High \rightarrow WFNR \rightarrow End$.

When at *Medium_Lock*, a thread can do one of four things: i) execute, represented by the $Medium\_Lock \rightarrow Medium\_Lock$ transition, during which its `runtime` is again incremented, ii) the thread can perform a nested locking call, represented by transition $Medium\_Lock \rightarrow S5 \rightarrow S6 \rightarrow High$, iii) the thread can unlock the latest locked resource, following the

Medium_Lock → *S*7 → *S*8 → *S*9 → *High* path, or iv) it can suspend itself through *Medium_Lock* → *S*2 → *S*3 → *High* → *Suspended*.

Being at location *Fp* the thread has the following four choices: i) it can execute (*Fp* → *Fp*), ii) it can perform a nested lock via *Fp* → *S*5 → *S*6, iii) it can unlock a resource (*Fp* → *S*7 → *S*8 → *S*9), or iv) it can choose to suspend itself (*Fp* → *S*2 → *S*3 → *Suspended*).

In order to lock a resource, a thread must not have reached the limit on simultaneously held resources (MAX_LOCK). If this criterion is satisfied (*Medium* → *S*5, *Medium_Lock* → *S*5, *Fp* → *S*5), then the thread synchronises on *prepareToLock!* (*S*5 → *S*6), while at the same time setting the type of the scheduling operation (sus=AT_LOCK). From *S6* the thread is moved to either *High* or *Fp*, depending on whether the resource belongs to a band or to a non-band priority level. If the thread is blocked due to the system ceiling, it is moved to the *Wait* location (*High* → *Wait*, *Fp* → *Wait*). There it waits until the system ceiling is lowered, at which point it is notified and moves back to *High* or *Fp*. Once at *High* or *Fp*, the thread synchronises on *reschedule_lock!* with a *High* → *High* or *Fp* → *Fp* transition, respectively. Again, this is due to the sus variable being set to AT_RESLOCK by the BaseScheduler in the *prepareToLock?* synchronisation. If the thread is at *High*, it can next be moved to either *Medium* or *Medium_Lock*, depending on whether locking takes place in the thread's own band or in a higher band. If it is at *Fp* it remains there.

In order to unlock a resource, a thread takes one of the transitions *Medium* → *S*7, *Medium_Lock* → *S*7, or *Fp* → *S*7, depending on its current priority. In *S*7 → *S*8 the thread sets the thread[id][LAST_LOCKED] variable to the resource, if any, it was holding when it locked the resource it is now unlocking. Then, the thread synchronises on *prepareToUnlock!* (*S*8 → *S*9), which sets sus=AT_RESUNLOCK and takes the thread to either *High* or *Fp*, depending on whether was locking within a band or at a non-band level. At *High* and *Fp* the thread synchronises on *reschedule_unlock!*, through the *High* → *High* and *Fp* → *S*9 transitions, respectively. It is then moved to either *Medium*, *Medium_Lock*, *Fp* or *Low*, depending on the circumstances.

To suspend itself a thread takes one of the three transitions *Medium* → *S*2, *Medium_Lock* → *S*2, or *Fp* → *S*2, depending on its current

priority. The limit on the number of times a thread can self-suspend is set to 1 (`lc<1`). The thread then synchronises on *prepareToSuspend!* ($S2 \rightarrow S3$) and is moved to either *Suspended*, or *High* and then to *Suspended*, depending on whether it is executing outside or within a band. When at *Suspended*, a thread can spend a variable amount of time there taking the *Suspended* $\rightarrow$ *Suspended* transaction. Once the thread is ready to resume, it synchronises on *start!* (*Suspended* $\rightarrow S4$), setting `sus=AT_UNBLOCK`. It is then moved to *High* or *Fp*, depending on its priority, and synchronises on *reschedule_resume!* through *High* $\rightarrow$ *High* and *Fp* $\rightarrow$ *Fp* transitions, respectively. From *High* it can be then moved to *Medium*, *Medium_Lock*, or *Low*, depending on the circumstances. If at *Fp*, it remains there.

While at *WFNR* the thread can increase its `runtime` through *WFNR* $\rightarrow$ *WFNR*. If `runtime` equals or exceeds the thread's `period` parameter *and* if the number of performed thread iterations is less than the number of maximum thread invocations (`iterations<thread[id][ITER]`), the thread is released again. Otherwise, if the number of maximum invocations has been reached, the thread is moved to *End*. When all threads reach *End*, the system has finished its execution. The `end` variable takes its maximum value and the transition *End* $\rightarrow$ *End* is possible. This is done so that testing the model for the (`A[] not deadlock`) property will return true.

5.2.5 BaseScheduler

Local variables: The `PL[]` array contains pre-calculated values of the absolute preemption level for each priority level in the system. This is calculated according to Equation 4.1 for π=1. For example, for priority level i=5, given that the number of preemption levels per band is L=100, cell `PL[5]` equals 101. The `system_ceiling[][]` array keeps a stack of the system ceilings, past and present. Row 0 (`system_ceiling[0][]`) holds the actual system ceilings, row 1 (`system_ceiling[1][]`) holds the thread ids that have locked the resource that set the corresponding ceiling, while row 2 (`system_ceiling[2][]`) holds the ids of the locked resources. `system_ceiling[0][0]` holds the current system ceiling. The `wait_queue[][]` array holds all blocked threads that are waiting for a resource to be freed. There is one queue for each resource in the system. When a particular resource is unlocked all threads waiting on that resource are notified.

All other variables hold temporary values to assist in setting up transition guards etc. `i` and `j` are counters; `ceil` and `rs` hold resource ceilings. `next_band` holds the calculated value of the next band the thread is going to. `tmp_thread` holds thread ids. `BPreCP` is a "mock" boolean variable that is always true. It is meant to demonstrate what path *prepareToLock?* and *reschedule_ unlock?* would be taking if the resource being locked or unlocked was not governed by the BPreCP protocol.

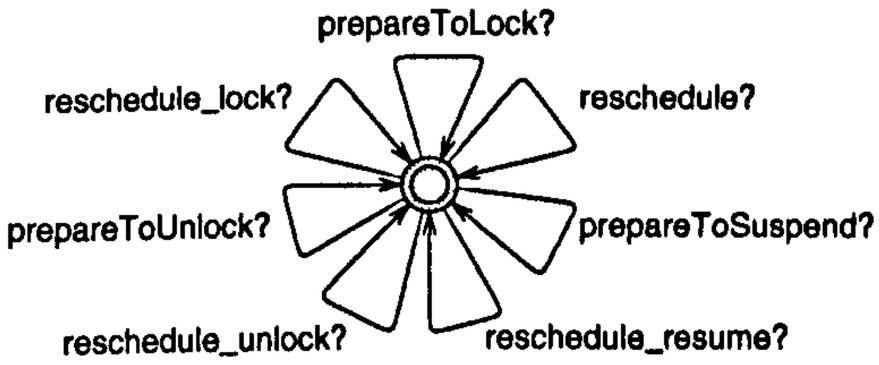


Figure 5.5: BaseScheduler abstract automaton

Description: The BaseScheduler automaton waits at the initial location *S0* for a Thread to synchronize on one of seven channels. This can be seen in Figure 5.5, which is an abstract depiction of the automaton. Following, the actions taken by the BaseScheduler for each of its synchronization channels will be presented, accompanied by the relevant portion of the BaseScheduler automaton. We will call each such portion of the BaseScheduler a "synchronisation". So, when a Thread automaton synchronises on e.g. *prepareToLock?*, it triggers a *prepareToLock?* "synchronisation" that starts with the given channel from location *S0* and traverses a path in the BaseScheduler that ends with *S0*. All locations in the automaton are specified as committed so as to guarantee that the execution of the scheduler will not be interrupted. The transition path in each such diagram begins and ends at the initial location *S0*.

A Thread automaton will synchronise on *reschedule?* (Figure 5.6) immediately after it has started. The base scheduler first calculates the absolute preemption level of the thread, based on the `PL[]` array ($S49 \rightarrow S50$), and then informs the application scheduler of the release ($S50 \rightarrow S51$). Following, the application scheduler is asked if the thread is eligible to execute in the band ($S51 \rightarrow S53$).

If the released thread is eligible to execute, the application scheduler will return the thread's id in `app_thread1` (`app_thread1==cur_thread`). If no other thread was previously executing in the band, the thread is put at *medium* priority ($S53 \rightarrow S60 \rightarrow ... \rightarrow S63$). If it is eligible, but there exists another

thread that was previously running at *medium*, the application scheduler will return the preempted thread's id in `app_thread2` (`app_thread2!=0`). The base scheduler will first put `app_thread2` at *low* ($S53 \rightarrow S55 \rightarrow ... \rightarrow S60$) and then put the released thread at medium ($S60 \rightarrow ... \rightarrow S63$). Finally, if the released thread is not eligible to execute (`app_thread1!=cur_thread`), it is put at *low* ($S53 \rightarrow S54 \rightarrow S62 \rightarrow S63$). In this case, the thread currently executing in the band is more eligible and will continue to execute. The base scheduler finally calls the dispatcher to schedule the system ($S63 \rightarrow S64 \rightarrow S0$).

When a thread is running, it can synchronize on *prepareToLock?*. This synchronization can be seen in Figure 5.7. After calculating the locking band (`next_band` in transition $S1 \rightarrow S2$), the base scheduler checks to see if this is the thread's first attempt to lock the resource (`AT_LOCK`), or if it is retrying after being blocked (`AT_BLOCKLOCK`). In the first case ($S2 \rightarrow S3$), the thread is moved to the appropriate priority level, i.e. if locking within a band (`next_band>=thread[cur_thread][BAND]`) it is placed on the locking band's *high* queue ($S3 \rightarrow S5$), otherwise (`next_band==0`) it is placed at the appropriate non-band priority level ($S3 \rightarrow S4 \rightarrow S5$). In the latter case, the thread will already be at the appropriate priority level, so no change in its priority takes place. Next, the preemption level test takes place, if the resource is governed by BPreCP (`BPreCP==true`). This is a mock guard, as this is always `true` in the model. If it was the case that `BPreCP==false`, the base scheduler would proceed with informing the application scheduler of the locking and it would be up to it to apply a resource sharing protocol ($S5 \rightarrow S32$). However, in the model the preemption level test is always taken. If the thread passes the test ($S5 \rightarrow S29$), we make sure its absolute preemption level is less or equal to the resource ceiling ($S29 \rightarrow S30$). We then update the system ceiling stack, saving, along with the system ceiling, also the thread id and the resource id ($S30 \rightarrow S31 \rightarrow S32$). Before the end of the synchronisation the base scheduler checks the thread's current band. If the thread is locking within a band (`((cur_band=bands[thread[cur_thread][PRIORITY]])>0)`), the base scheduler notifies the locking band's application scheduler of the locking ($S32 \rightarrow S33 \rightarrow S34$). If locking happened at a non-band priority (`cur_band==0`), then no notification is necessary. This is somewhat different from the actual framework protocol, which specifies that the thread's application scheduler must always be notified of the locking, whether this takes place within its own band, at a non-band priority, or within a higher band. However, Section 4.5.5 has made the point that the only reason for such notification is to cover the case where knowledge of the operation affects the

task's eligibility, according to some policy. In our model we are only using an EDF scheduler whose policy is not affected by a thread's locking history. Therefore, in order to keep the model complexity low, we do not inform a thread's own scheduler, when locking occurs outside its own band. Finally, the base scheduler indicates, by setting `thread[cur_thread][RFH]=AT_RESLOCK`, that the next action taken by the thread (once it locks the actual resource) should be to call *reschedule_lock?* ($S34 \rightarrow S0$).

If the thread failed the preemption level test ($S5 \rightarrow S6$), its application scheduler is informed that it has blocked through the *blocked!* channel ($S6 \rightarrow ... \rightarrow S9$). As parameters to this synchronisation the base scheduler passes the blocked thread (`app_thread1=cur_thread`) and whether the thread's band is the locking band (`app_thread2==0`) or not (`app_thread2==-1`). Then the thread is instructed to *wait!* and is added to the `wait_queue[][]` for the resource it blocks on. The threads application scheduler is further informed that the thread has suspended ($S11 \rightarrow S12 \rightarrow S13$), meaning that it has relinquished the CPU, and is asked for its next most eligible thread ($S13 \rightarrow S14 \rightarrow S15$). If no thread is returned, the system is scheduled ($S15 \rightarrow S27 \rightarrow S28$). If there is such thread, it is removed from *low* and placed on *medium* ($S15 \rightarrow ... \rightarrow S21$). If, additionally, the application scheduler also returns a thread in `app_thread2`, then this thread has been preempted by `app_thread1` and must be put at *low* ($S21 \rightarrow ... \rightarrow S27$). If `app_thread2` is zero, meaning that no thread has been preempted, the base scheduler immediately asks the dispatcher to schedule the system ($S21 \rightarrow S27 \rightarrow S28$). Finally, the base scheduler indicates, by setting `thread[cur_thread][RFH]=AT_BLOCKLOCK`, that the next action of the thread should be to again synchronise on *prepareToLock?* (i.e. once it has been notified that the resource has been unlocked).

After locking takes place, a thread synchronises on *reschedule_lock?* (Figure 5.8). The thread is placed at the appropriate priority level, depending on where it is executing. If the locking band is not its own band, the thread is taken to its *medium_lock* priority ($S36 \rightarrow S46 \rightarrow S47$). If the thread is at a non-band level, its priority remains unaltered ($S36 \rightarrow S46 \rightarrow S47$). If the thread is locking in its own band, an eligibility check is asked for ($S36 \rightarrow ... \rightarrow S39$), exactly as described for *reschedule?*. If the thread is eligible, it will either preempt another band thread ($S39 \rightarrow S40$) or not ($S39 \rightarrow S44$). Either case, it is moved to *medium* ($S44 \rightarrow S46 \rightarrow S47$). If the thread is not eligible, it is moved to *low* ($S39 \rightarrow S45 \rightarrow S46 \rightarrow S47$). At the

end of the synchronization the dispatcher is called once more to *schedule!* the system ($S47 \rightarrow S48 \rightarrow S0$).

Before unlocking a resource, a thread synchronizes on *prepareToUnlock?* (Figure 5.9), which is charged with putting the thread on the right priority queue. If the thread is executing within a band (`cur_band=bands[thread[cur_thread][PRIORITY]]`, `cur_band>=thread[cur_thread][BAND]`), it is put at the band's *high* priority ($S66 \rightarrow S67$). Otherwise, it remains at the fixed priority it currently has ($S66 \rightarrow S67$). Finally, the thread is instructed to synchronise on *reschedule_unlock?* next (`thread[cur_thread][RFH]=AT_RESUNLOCK`).

After actually unlocking a resource, a thread will synchronise on *reschedule_unlock?* (Figure 5.10). If the thread is executing within its own band, it will continue to run in it and `next_band` is set to `next_band=cur_band` in $S105 \rightarrow S106$. If the thread is executing within a band, that band is notified of the *unlock* operation ($S106 \rightarrow S107 \rightarrow 108$). Again, as explained in *reschedule_lock?*, the thread's own band is not necessarily notified. Next, the calculation of the band the thread will next go to is concluded. If the thread is currently executing outside its own band and it has additional locked resources, then the `next_band` is calculated from the ceiling of its next locked resource (`(next_band==0 and thread[cur_thread][LAST_LOCKED]>0)` in $S108 \rightarrow S109$). If the thread is currently executing outside its own band but has no additional locked resources, then the `next_band` will be its own band (`(next_band==0 and thread[cur_thread][LAST_LOCKED]==0)` in $S108 \rightarrow S109$). The next transition is again guarded by the `BPreCP` flag. In the hypothetical case that the unlocked resource is not governed by BPreCP (`BPreCP==false`), the base scheduler leaves the system ceiling as is and proceeds with transition $S9 \rightarrow S115$. However, this again is never the case and the base scheduler proceeds to find the unlocked resource's ceiling among the system ceiling values in the `system_ceiling[][]` array ($S109 \rightarrow S110 \rightarrow S111$). Then, all threads blocked on the unlocked resource are notified and placed on the queue for their respective priority ($S111 \rightarrow S112 \rightarrow S113 \rightarrow S111$). Immediately after that, the system ceiling value corresponding to the unlocked resource ceiling is removed from the `system_ceiling[][]` array ($S111 \rightarrow S114 \rightarrow S115$). This value can be at any position in the array. After that, the thread is moved to the appropriate priority level to execute after the unlocking. If the thread is going to execute within a band other than its own, it is moved to that band's *medium_lock* priority

($S115 \rightarrow S125 \rightarrow S126 \rightarrow S127$). If it is going to execute at a non-band level, that level is calculated using the resource ceiling and the `PL[]` array, and the thread is moved to it ($S115 \rightarrow S124 \rightarrow S126 \rightarrow S127$). If the thread is going to execute within its own band, the base scheduler asks the thread's application scheduler to perform an eligibility check ($S115 \rightarrow S116 \rightarrow S117$). Then the paths are the same as in *reschedule?* and *reschedule_lock?*. If the thread is the most eligible and if it preempts another thread, the path is ($S117 \rightarrow ... \rightarrow S122 \rightarrow S126 \rightarrow S127$). If the thread is the most eligible without preempting another thread, the path is ($S117 \rightarrow S122 \rightarrow S126 \rightarrow S127$). If the thread is not eligible, the path is ($S117 \rightarrow S123 \rightarrow S126 \rightarrow S127$). Finally, the dispatcher is again called to schedule the system ($S127 \rightarrow S128 \rightarrow S0$). The final transition in the synchronisation ($S128 \rightarrow S0$) sets the entry in the `t_res[][]` array corresponding to the unlocked resource to zero, so that it will not be locked by the thread again. This is again a way of minimising the size of the model state set.

The final two scheduling operations included in the model, *suspend* and *end*, are modelled using the same synchronisation, namely *prepareToSuspend?* (Figure 5.11). As it has already been pointed out in the previous chapter, the *end* operation is a special case of the more general *suspend* operation, their differences being that *end* has no *sbsc* and can be executed only if a thread holds no resources. In the model, *end* is the only operation that is also executed by non-band threads. *prepareToSuspend?* is the same for both operations (in essence modelling their *pbsc*), the semantic difference between the two operations being evident only in the Thread automaton. If the suspending thread is a simple fixed-priority thread (`cur_band==0` or `cur_band==-1`), then the base scheduler just calls for a system schedule ($S0 \rightarrow S68 \rightarrow S69 \rightarrow S70 \rightarrow S103 \rightarrow S104 \rightarrow S0$). If the thread belongs to a band, the base scheduler initially informs the thread's scheduler that its thread has been suspended ($S70 \rightarrow S71 \rightarrow S72$). Then the scheduler is asked to provide its next most eligible thread ($S72 \rightarrow S73 \rightarrow S74$). If no thread is returned (`app_thread1==0`) then the path is $S74 \rightarrow S87$. If a thread is returned (`app_thread1!=0`), but with no preemption taking place (`app_thread2==0`), then it is simply removed from *low* and put to *medium* ($S74 \rightarrow ... \rightarrow S80 \rightarrow S86 \rightarrow S87$). If `app_thread1` preempted `app_thread2` (`app_thread2!=0`), then the preempted thread is put to *low* ($S80 \rightarrow ... \rightarrow S86$). If the suspending thread is performing a general *suspend* operation (not an *end*) and is executing within another band, that band is also

informed of the suspension and asked for its most eligible thread ($S87 \rightarrow ... \rightarrow S91$). Again, there could be no returned thread ($S91 \rightarrow S103$), or there could be a returned thread with no preemption ($S91 \rightarrow ... \rightarrow S97 \rightarrow S103$), or with preemption ($S91 \rightarrow ... \rightarrow S103$). Finally, the base scheduler calls for a system schedule ($S103 \rightarrow S104 \rightarrow S0$).

The final part of the BaseScheduler automaton is the *reschedule_resume?* synchronisation (Figure 5.12). A thread synchronises on this channel when it is ready to resume execution after having suspended itself. Depending on what its priority is at the time it wakes up (`cur_band=bands[thread[cur_thread][PRIORITY]]`), the base scheduler moves the thread at the appropriate level. If the thread is executing within a band higher than its own, it is placed on that band's *medium_lock* priority, since in order to execute outside its band it must be locking a resource. Then the locking band's scheduler is notified of the resumption ($S130 \rightarrow S141 \rightarrow S142 \rightarrow S143 \rightarrow S145 \rightarrow S146$). If the thread's priority is a non-band priority, it is placed on the queue for that priority ($S130 \rightarrow S144 \rightarrow S145 \rightarrow S146$). Finally, if the thread is executing within its own band, the base scheduler asks the band's application scheduler to perform an eligibility test ($S130 \rightarrow S131 \rightarrow S132$). If the thread is eligible, it will either preempt another band thread ($S132 \rightarrow ... \rightarrow S138$) or not ($S132 \rightarrow S138$). Either case, it is moved to *medium* ($S138 \rightarrow S139 \rightarrow S145 \rightarrow S146$). If the thread is not eligible, it is moved to *low* ($S132 \rightarrow S140 \rightarrow S145 \rightarrow S146$). At the end of the synchronization the dispatcher is called once more to *schedule!* the system ($S146 \rightarrow S147 \rightarrow S0$).

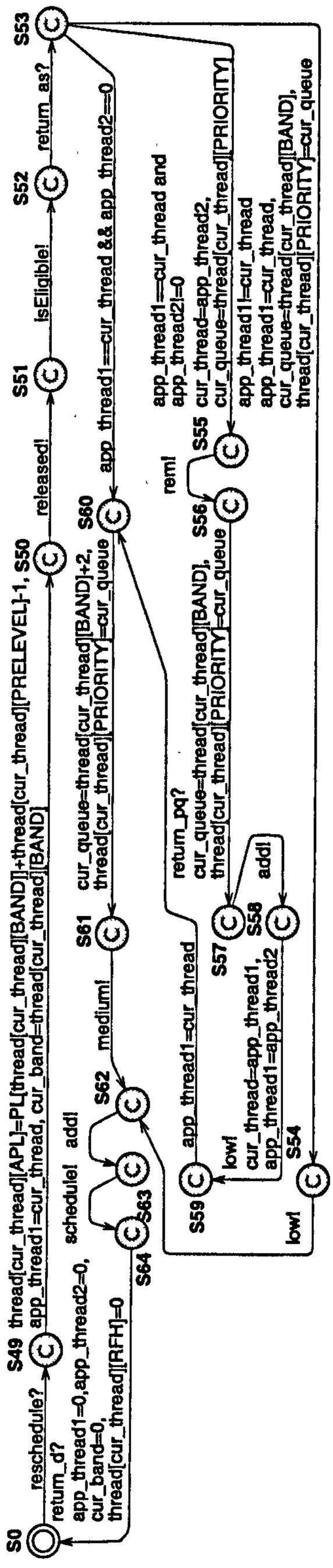


Figure 5.6: The *reschedule?* synchronization

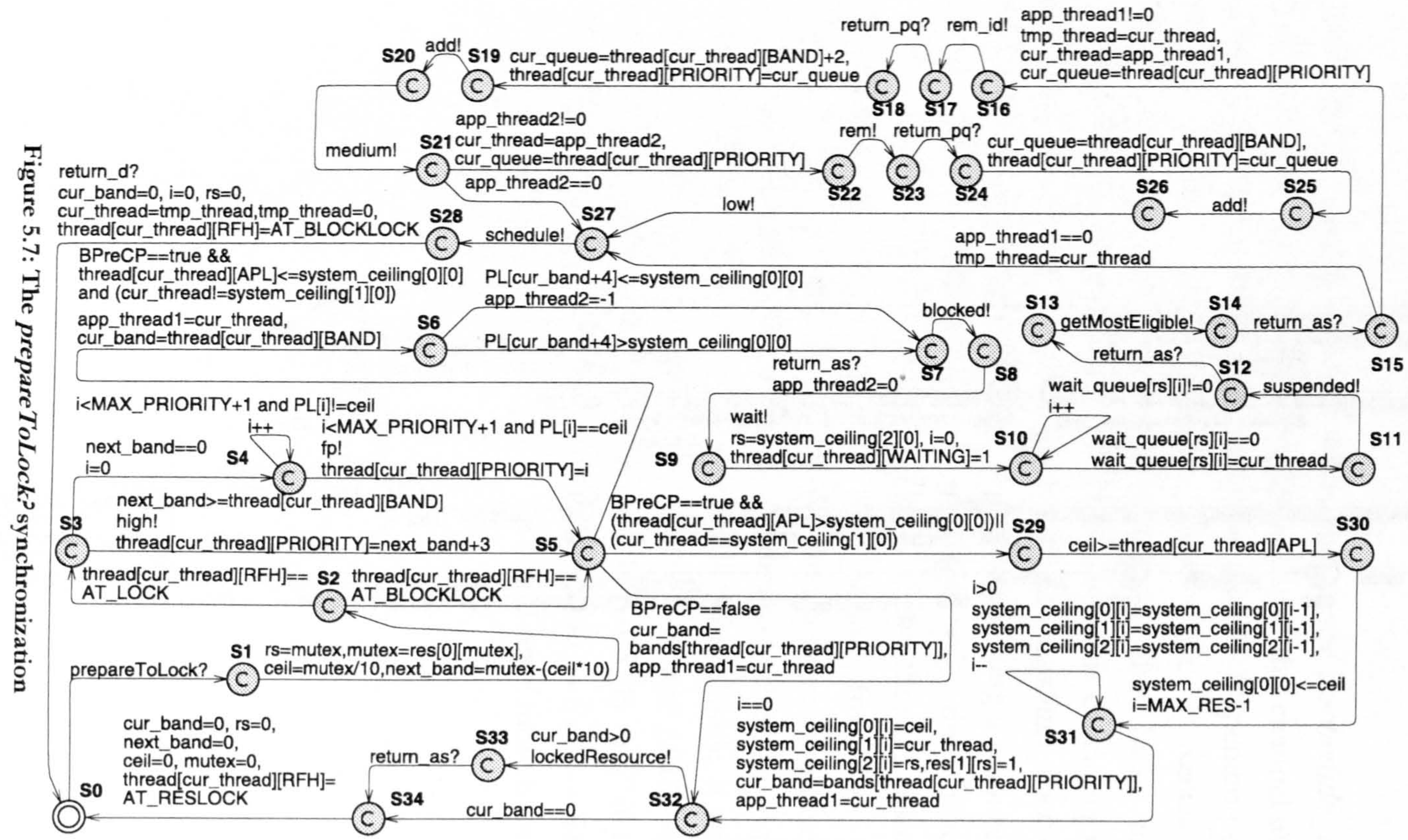


Figure 5.7: The *prepareToLock?* synchronization

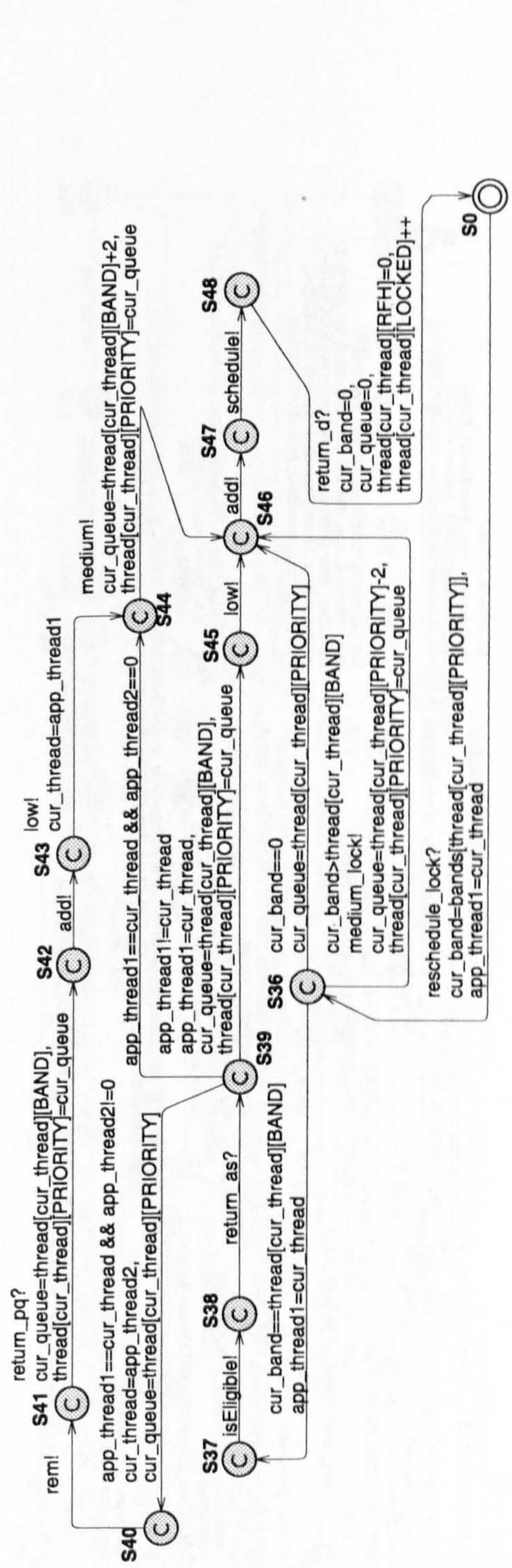


Figure 5.8: The *reschedule_lock?* synchronization

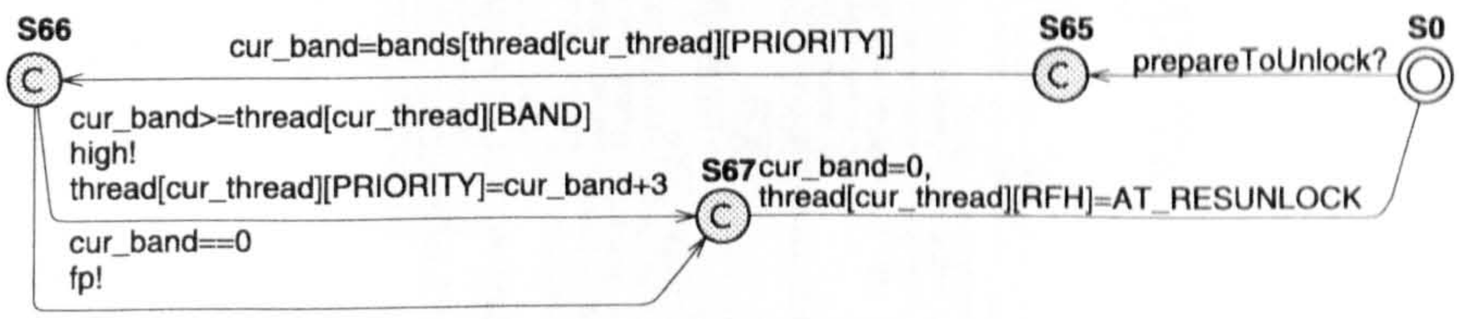


Figure 5.9: The *prepareToUnlock?* Synchronization

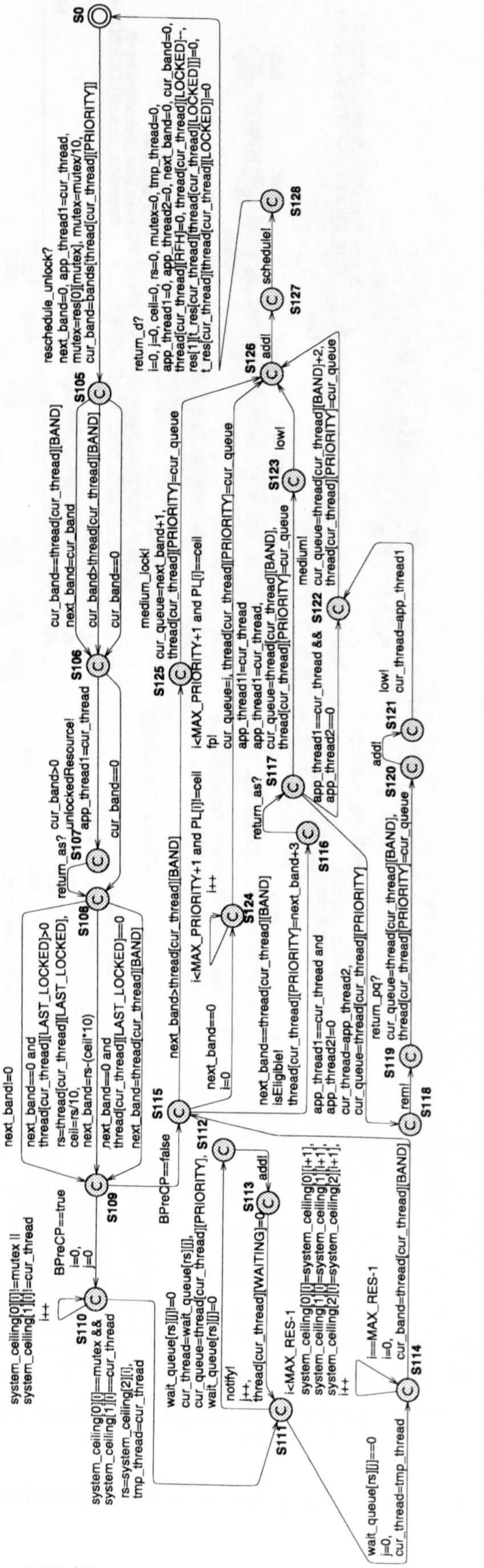


Figure 5.10: The *reschedule_unlock?* synchronization

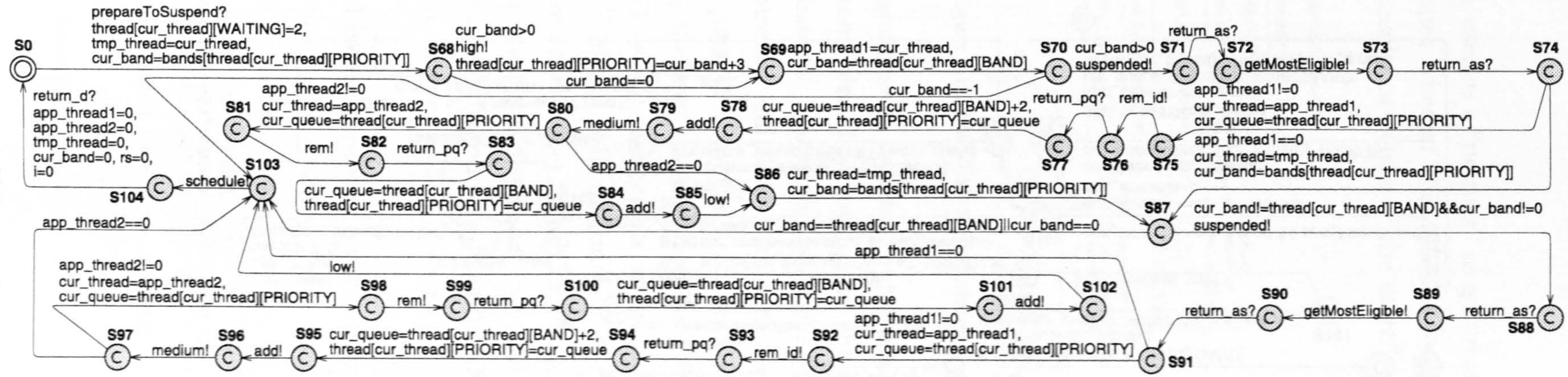


Figure 5.11: The *prepareToSuspend?* synchronization

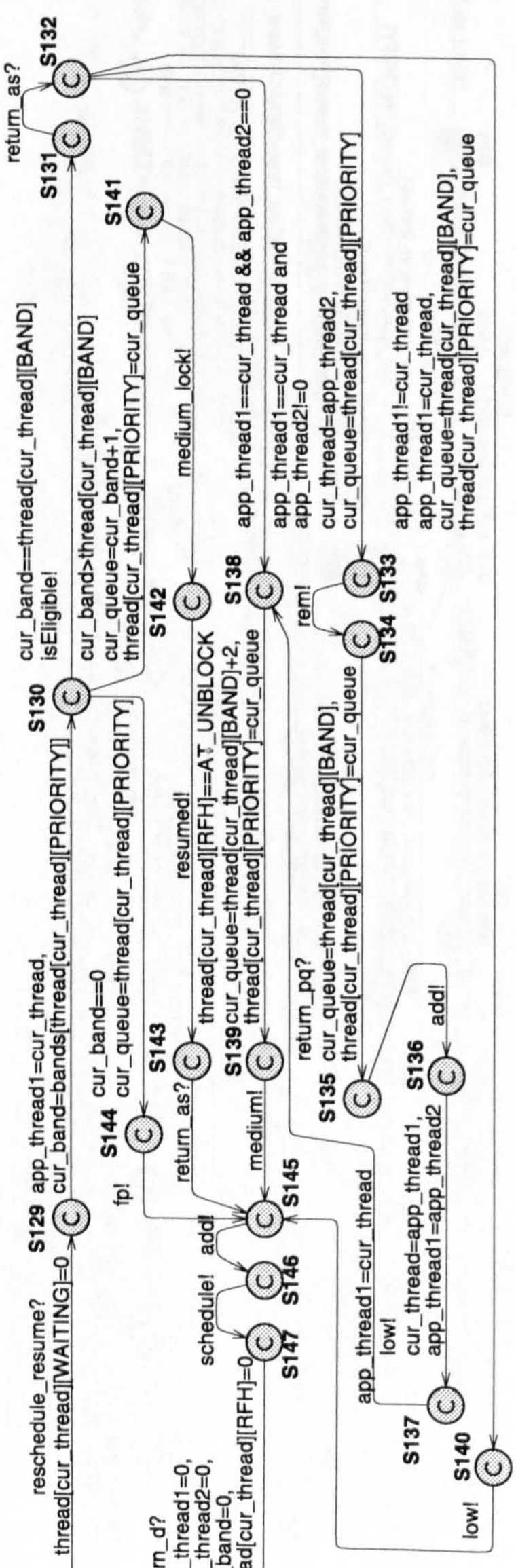


Figure 5.12: The *reschedule_resume?* synchronization

5.2.6 Dispatcher

Local variables: `pq` is used as a "pointer" to go through the priority queues. `cft` is a flag that indicates whether the *schedule?* synchronisation has been initiated by the base scheduler or as part of the *start?* synchronisation.

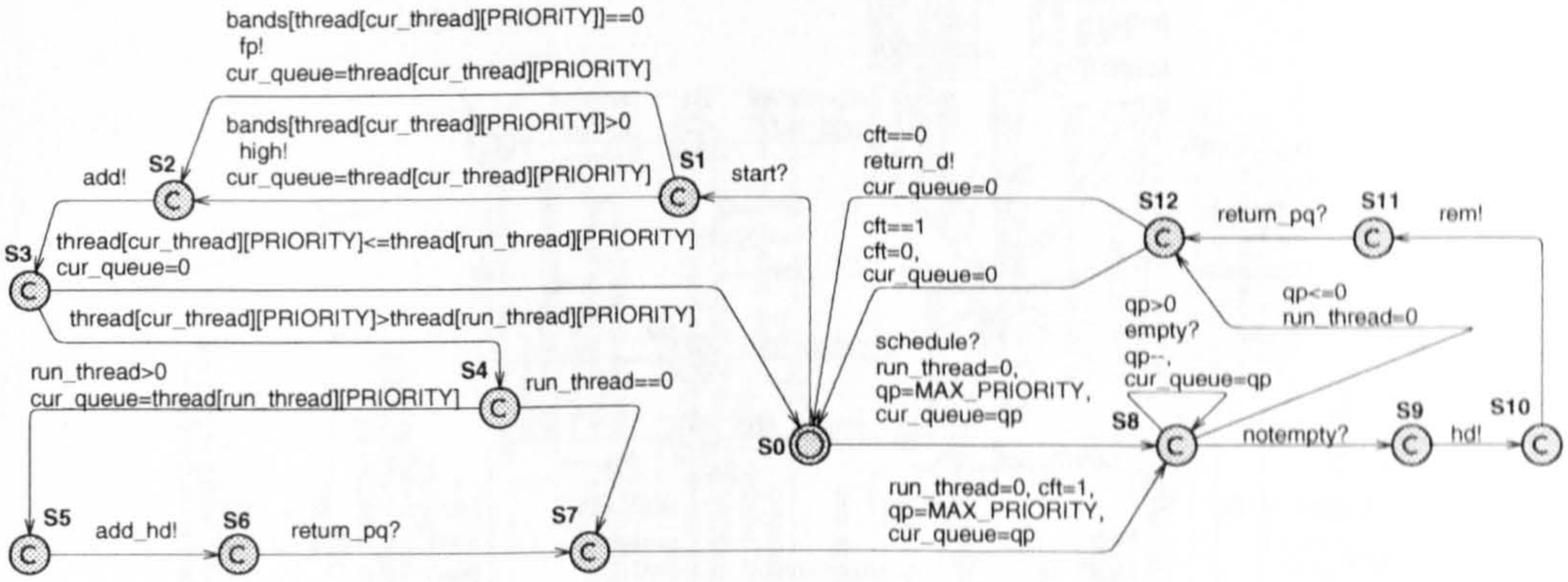


Figure 5.13: The Dispatcher automaton

Description: The Dispatcher is, together with PriQueue, represent the low-level part of the base scheduler. The automaton (Figure 5.13) contains only two synchronization channels *start?* and *schedule?*. *start?* is synchronised upon by a Thread automaton and deals with a thread's release. It takes the released thread and puts it on the priority queue for its priority ($S0 \rightarrow ... \rightarrow S3$). Then, if there is no thread running, the synchronisation immediately returns ($S3 \rightarrow S0$). If there is a thread already running in the system and its priority is higher than the priority of the released thread, the synchronisation simply returns ($S3 \rightarrow S0$). If there is a thread already running and its priority is lower, then the dispatcher puts the running thread at the head of the queue for its priority and schedules the system, as if a *schedule?* synchronisation had been received ($S3 \rightarrow ... \rightarrow S8$).

When the base scheduler synchronises on *schedule?*, the dispatcher tries to find a non-empty priority queue, starting from the highest priority in the system ($S8 \rightarrow S8$). If it finds one, it asks for the thread at the head of the queue and removes it from the queue ($S8 \rightarrow ... \rightarrow S12$). If all queues are empty then the running thread is set to zero and it returns ($S8 \rightarrow S12$). The synchronisation returns in one of two ways, depending on whether it was the base scheduler that initiated the *schedule?* synchronisation (`cft==0`), or a thread via the *start?* synchronisation (`cft==1`).

5.2.7 PriQueue

Local variables: `list[]` is the actual FIFO-ordered queue. `len` holds the current length of the queue, i.e. how many threads it currently holds. `i` is a counter.

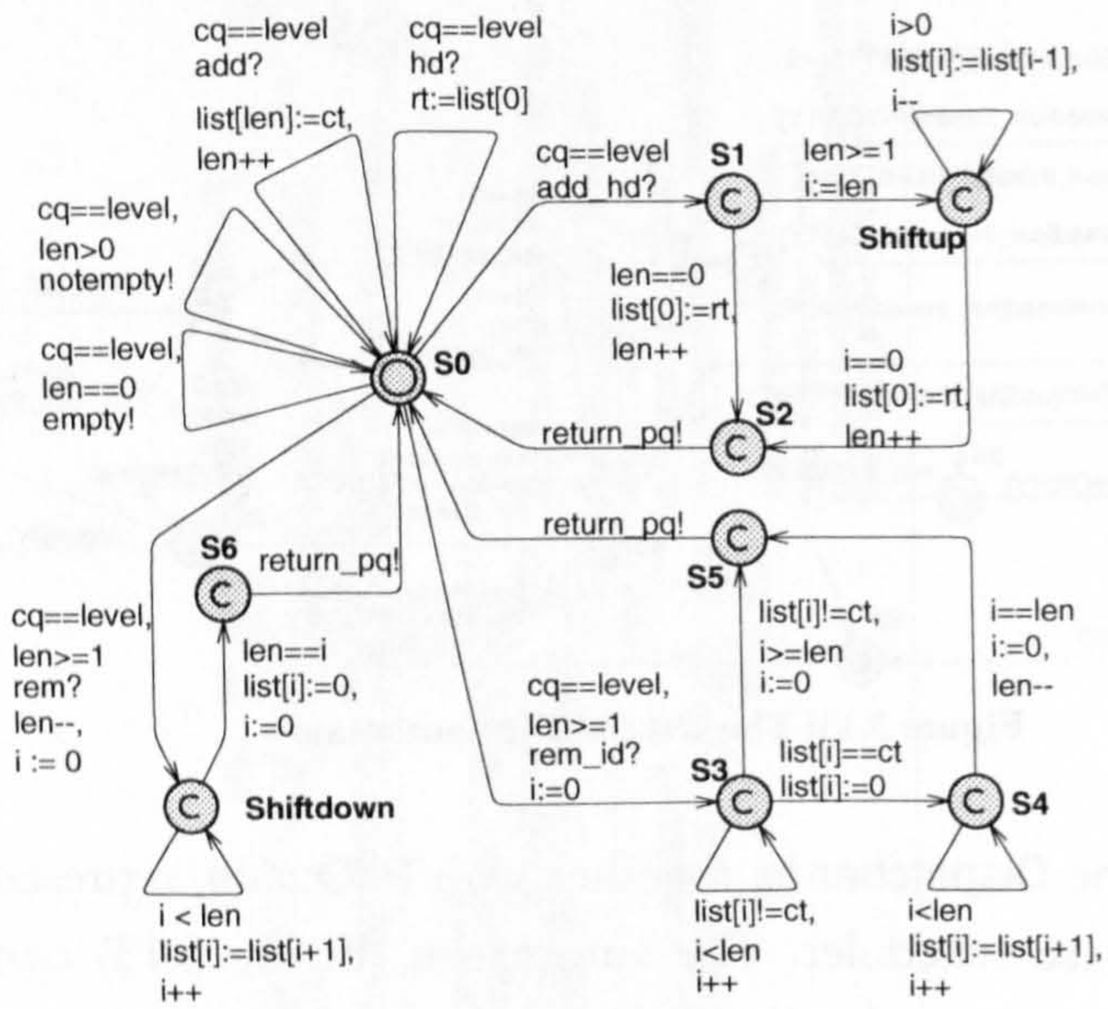


Figure 5.14: The PriQueue automaton

Description: Each PriQueue represents a separate base scheduler priority level. It maintains a FIFO queue, which both the BaseScheduler and the Dispatcher manipulate. The defined queue operations are to add a thread at the head (*add_hd?*) or the tail (*add?*) of the queue, to remove a thread from the head (*rem?*) or from wherever it is in the queue (*rem_id?*), to get the head of the queue (*hd?*), and finally to ask if the queue is *empty!* or *notempty!*. The automaton can be seen in Figure 5.14.

5.2.8 EDFScheduler

The EDFScheduler automaton is not part of the core framework. As we have already mentioned, it is a pluggable component of the model, which can be replaced by any other automaton implementing an application-defined scheduler, as long as that automaton conforms to the same interface, i.e. defines the same synchronisations: *released?*, *blocked?*, *suspended?*, *resumed?*, *isEligible?*, *getMostEligible?*, *lockedResource?* and *unlockedResource?*.

Local variables: The main scheduler data structures are the three described in Section 5.1. The `app_queue[][]` array is the scheduler's internal EDF queue. It is a simple EDF-ordered queue that keeps those band threads that

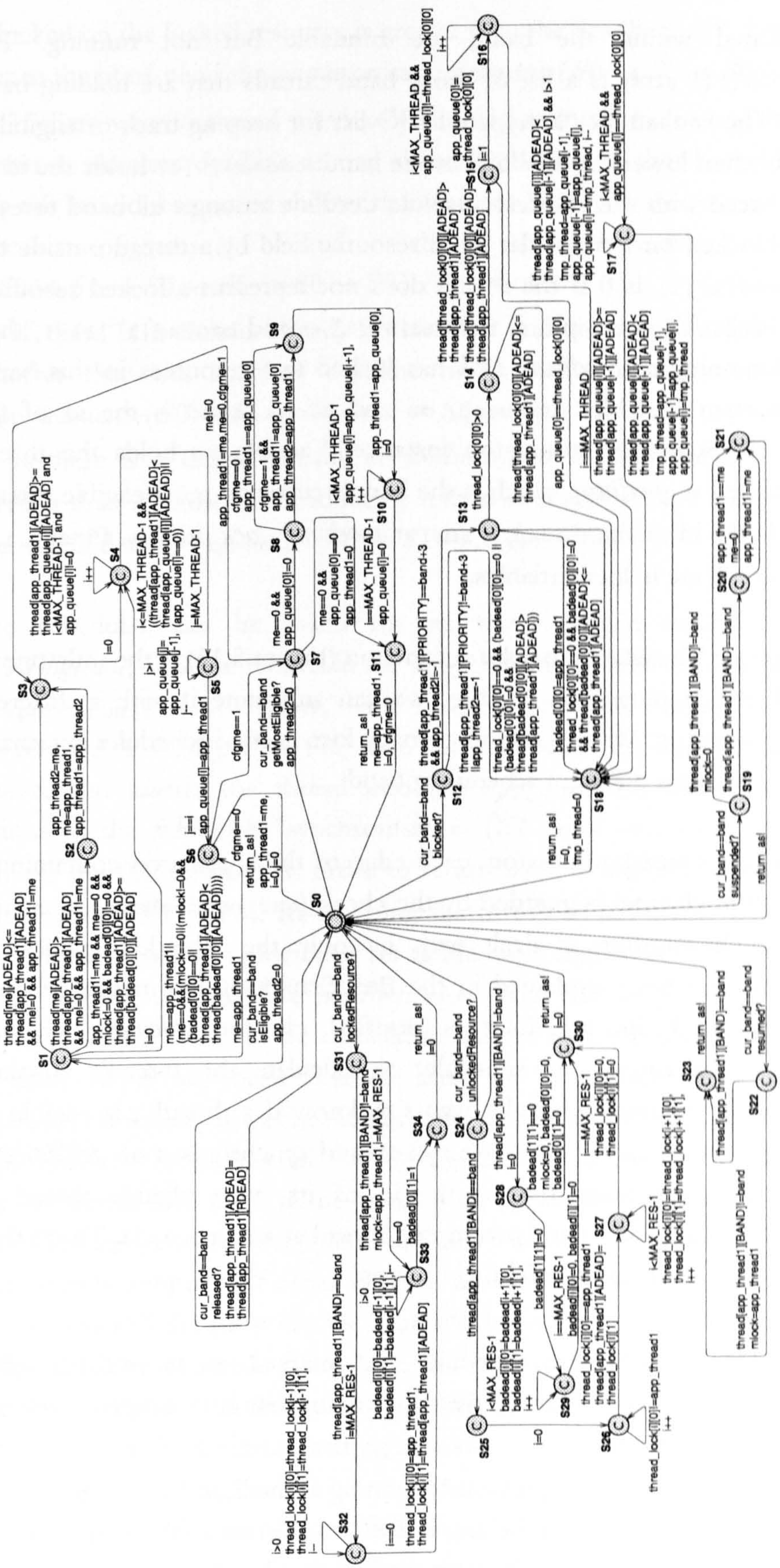


Figure 5.15: The EDFScheduler automaton

are located within the band, are runnable but not running. The `thread_lock[][]` array is a list of those band threads that are holding band resources. The `badead[][]` array is a LIFO list for keeping track of eligibility inheritance when lower threads lock in the band. `badead[][0]` holds the id of the band thread with the shortest absolute deadline amongst all band threads that have blocked on a particular band resource held by a thread outside the band. `badead[i][1]` is 0 if the i^{th} cell does not represent a locked resource, and 1 if it does. For example, if `badead[0][1]=1` and `badead[1][1]=1`, then the thread running at *medium_lock* has locked two resources in this band. Three important variables are `band`, `me` and `mlock`. `band` is the id of the particular EDFScheduler automaton instance. Variable `me` holds the thread currently placed at *medium*, which is the band's currently most eligible thread. `mlock` holds the id of the thread, if any, at *medium_lock* priority. Finally, `i`, `j` and `tmp_thread` are helper variables.

Description: The EDFScheduler automaton (Figure 5.15) is the only one in the model that is pluggable. That is, we can substitute it with a different scheduler automaton. We can also have many instances of scheduler automata, each one managing a different scheduling band.

As with the Thread automaton, each edge of the automaton containing a synchronisation channel is guarded by the check (`cur_band==band`) to ensure that the synchronisation is dealt with by only the intended application scheduler. As has been explained in the BaseScheduler automaton, when a thread is released, its scheduler is notified via the *released?* channel ($S0 \rightarrow S0$). This causes the scheduler to calculate the thread's absolute deadline. When the base scheduler wants to know if a thread τ is eligible to run, it places the thread id in `app_thread1` and synchronises on *isEligible?*. The application scheduler returns the id of its most eligible thread in `app_thread1`, and the id of any preempted thread at `app_thread2`. To do this, the EDF scheduler checks the following cases:

- If τ is the one the scheduler has marked as its most eligible (`app_thread1==me`) then the synchronisation simply returns ($S0 \rightarrow S1 \rightarrow S6 \rightarrow S0$).
- If there was no thread previously running at medium (`me==0`) and either there is no outside thread locking in the band (`mlock==0`), or there is but the shortest absolute deadline among those band threads that have

blocked on the locked resource is greater than the deadline of τ, then me is set to the id of τ and the synchronisation returns ($S0 \rightarrow S1 \rightarrow S6 \rightarrow S0$).

- If there was no thread previously running at medium (me==0) and the shortest absolute deadline among those band threads that have blocked on the locked resource is shorter than the deadline of τ, then τ is added to the app_queue[] and the synchronisation returns ($S0 \rightarrow S1 \rightarrow S4 \rightarrow S5 \rightarrow S6 \rightarrow S0$).
- If there is a thread already at *medium* and its absolute deadline is shorter than the absolute deadline of τ, then τ is again added to app_queue[] ($S0 \rightarrow S1 \rightarrow S3 \rightarrow ... \rightarrow S6 \rightarrow S0$).
- If there is a thread already at *medium* and its absolute deadline is greater than the absolute deadline of τ, then me is set to τ and the previous me is added to app_queue[] ($S0 \rightarrow S1 \rightarrow ... \rightarrow S6 \rightarrow S0$).

The scheduler can be asked for its most eligible thread through *getMostEligible?*, which it returns in the app_thread1 variable. If me==0 and if the app_queue[] is empty, the scheduler sets app_thread1=0 ($S0 \rightarrow S7 \rightarrow S11 \rightarrow S0$). If me!=0, then the scheduler sets app_thread1=me and me=0, and inserts the thread into the app_queue[], using the same transitions as the *isEligible?* synchronisation ($S7 \rightarrow S3 \rightarrow ... \rightarrow S7$). To do this, it sets the flag cfgme=1 in order to return back to the *getMostEligible?* synchronisation ($S6 \rightarrow S7$). Re-inserting the thread in the queue guarantees that the thread which was previously set as the most eligible in the band is still the most eligible. A third case is when me==0 and the app_queue[] is not empty ($S7 \rightarrow S8$). This can be reached in one of two ways: either it is the initial case, or it has been reached after me was placed back in the app_queue[]. If initially it was me!=0 and now that thread is not at the front of the app_queue[] (cfgme==1 in $S8 \rightarrow S9$), the scheduler knows that the former me thread will need to be preempted by another thread, so it sets app_thread2=app_thread1 (app_thread1 now holds the previous me thread) to indicate its preemption. If cfgme==0 or if the me thread is at the front of the queue (i.e. it is still the most eligible), the synchronisation simply proceeds to $S9$. The scheduler proceeds to set app_thread1 to the first thread on the app_queue[] ($S9 \rightarrow S10$) and moves all threads one place up in the queue ($S10 \rightarrow S10$). Finally, it returns, setting cfgme=0.

The EDF scheduler can be notified of a suspending thread through the *suspended?* synchronisation, both if the thread is a band thread and if it is an outside thread locking within the band. If the thread is a band thread and was

the `me` thread, the scheduler just resets `me` to zero ($S0 \rightarrow S19 \rightarrow S20 \rightarrow S21 \rightarrow S0$). If it was an outside thread, it sets `mlock` to zero ($S0 \rightarrow S19 \rightarrow S21 \rightarrow S0$). When a thread wakes up from suspension, the base scheduler synchronises on *resumed?* to notify the EDF scheduler that the thread is ready to execute again. If the resuming thread does not belong to the scheduler's band, the scheduler will know that it is a thread locking at *medium_lock* and therefore, will set `mlock` to the thread's id (`mlock==app_thread1` in $S22 \rightarrow S23$). If the thread is of the scheduler's band, the scheduler does nothing. This second case never occurs but is included for completeness.

The base scheduler can inform the EDF scheduler of a thread locking a resource within its band by synchronising on its *lockedResource?* channel. As usual, `app_thread1` holds the id of the thread performing the operation. If the locking thread belongs to the scheduler's band, it is added to the top of the `thread_lock[][]` list, together with the absolute deadline it has at the time of locking, and the synchronisation ends ($S0 \rightarrow S31 \rightarrow S32 \rightarrow S34 \rightarrow S0$). If the locking thread is not a band thread, the scheduler saves the thread's id in `mlock` and pushes the `badead[][]` stack down, to create space for the newly locked resource ($S33$). `badead[0][0]` holds the id of the band thread with the shortest absolute deadline that has blocked on the band resource most recently locked by the thread locking at *medium_lock*. The initial value of this thread id is the same as the one for the previous resource, if any. That is, initially `badead[0][0]==badead[1][0]`. Before returning, the scheduler sets `badead[0][1]=1` to indicate that the *medium_lock* thread has locked a resource.

When a thread unlocks a resource, the base scheduler synchronises on *unlockedResource?*. If the thread is a band thread ($S24 \rightarrow S25$), the EDF scheduler searches through the `thread_lock[][]` array to find the first entry for the particular thread ($S26$). It then undoes eligibility inheritance by setting the unlocking thread's absolute deadline to the value it had at the time of locking the resource (`thread[app_thread1][ADEAD]=thread_lock[i][1]` in $S26 \rightarrow S27$), and proceeds to remove the entry ($S27 \rightarrow S30$). If the thread is not of this band ($S24 \rightarrow S28$), the scheduler checks to see if the resource it is unlocking is the last one it holds within the band. If it is (`badead[1][1]==0`), then the scheduler sets `mlock`, `badead[0][0]` and `badead[0][1]` to zero and returns ($S28 \rightarrow S30 \rightarrow S0$). If `badead[1][1]!=0` (the thread holds other resources as well), the scheduler removes the top of the `badead[][]` stack and returns ($S28 \rightarrow S29 \rightarrow S30 \rightarrow S0$).

Finally, the EDF scheduler can be informed that one of its threads has blocked when trying to lock a resource through the *blocked?* synchronisation channel. If the thread blocked outside the band, the synchronisation simply returns ($S0 \rightarrow S12 \rightarrow S18 \rightarrow S0$). If the thread blocked inside the band ($S12 \rightarrow S13$), the scheduler checks the `thread_lock[][]` array. If `thread_lock[0][0]==0`, then a *medium_lock* thread must have set the system ceiling. If the current inherited absolute deadline of the *medium_lock* thread is zero or greater than the absolute deadline of the blocked thread, then `badead[0][0]` is set to hold the id of the blocked thread (`badead[0][0]=app_thread1` in $S13 \rightarrow S18$). If the current inherited absolute deadline of the *medium_lock* thread is less than the absolute deadline of the blocked thread, the synchronisation returns ($S13 \rightarrow S18 \rightarrow S0$). If `thread_lock[0][0]!=0`, then it is a band thread that has set the system ceiling ($S13 \rightarrow S14$). If the locking thread's absolute deadline is shorter than the blocking thread's deadline, the synchronisation returns ($S14 \rightarrow S18 \rightarrow S0$). If the locking thread's deadline is greater, then it acquires the deadline of the blocking thread ($S14 \rightarrow S15$). Then, if the locking thread is the first in the `app_queue[]`, the synchronisation returns ($S15 \rightarrow S18 \rightarrow S0$). If not, then the scheduler rearranges the `app_queue[]` according to the locking thread's new deadline ($S15 \rightarrow ... \rightarrow S18 \rightarrow S0$).

5.2.9 Formal analysis of the model

In this section we specify certain properties both to evaluate the correctness of our model and to explore its behaviour. To test these properties we have defined a system with 10 priority levels (i.e. 10 PriQueue automata) and two EDF bands with $low_{EDF1} = 1$ and $low_{EDF2} = 6$ (Figure 5.16).

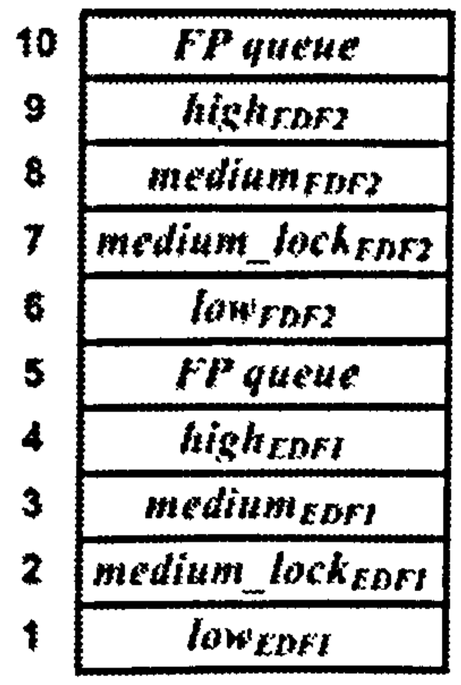


Figure 5.16: The test system

It follows that priority levels 5 and 10 are left under direct base scheduler control. To test the system, seven threads are specified, only three of which are used at any one time (3 Thread automata), in order to keep the system analysable. Threads τ_1, τ_3 and τ_7 belong to band B_1, threads τ_2 and τ_5 belong to band B_6, while threads τ_4 and τ_6 are non-band threads with priorities 5 and 10 respectively. Based on this we have specified a number of scenarios to test against, which can be seen in Table 5.1.

Table 5.1: Number of threads per band

Test case	B_1	*Priority 5*	B_6	*Priority 10*
A	τ_3	τ_4	τ_2	0
B	τ_3	τ_4	0	τ_6
C	τ_3	0	τ_2	τ_6
D	τ_1, τ_3	τ_4	0	0
E	0	τ_4	τ_2, τ_5	0
F	τ_1, τ_3	0	τ_2	0
H	τ_1, τ_3, τ_7	0	0	0
I	τ_3	0	τ_2, τ_5	0

We also specify 6 resources. Resource usage is shown in Table 5.2 below. The aim is to cover as many different situations as possible by making combinations of tasks, bands and resources used. At the same time queries need to be kept small enough to be able to run them. This is why only a subset of the resources is used in each test case. For example, in case A thread τ_3 is using 3 resources because this test case runs only two band threads. Test case H runs three band threads, which means more computations, and hence thread τ_3 uses only two resources.

Table 5.2: Resource usage per test case

Test case	*Band 1*		*Priority 5*	*Band 6*		*Priority 10*
	r_1	r_6	r_2	r_3	r_4	r_5
A	τ_3		τ_3	τ_2, τ_3	τ_2	
B	τ_3		τ_3			τ_3
C	τ_3			τ_2, τ_3		τ_3
D	τ_1, τ_3		τ_1, τ_3			
E				τ_2	τ_2, τ_5	
F				τ_1, τ_2, τ_3	τ_1	
H	τ_1, τ_3	τ_3, τ_7				
I				τ_2, τ_3	τ_2, τ_3, τ_5	

Model Consistency First we specify three properties to check the correctness of our model.

Property 1: *Throughout its execution a thread will only be placed on either its own band's low, medium or high queue, or on a higher band's medium_lock or high queue, or on a normal priority queue above its band.*

We specify a safety property for each thread. For thread τ_3 in test case A the property is written:

```
A[] (thread[3][PRIORITY]!=2 && thread[3][PRIORITY]!=6 &&
thread[3][PRIORITY]!=8 && thread[3][PRIORITY]!=10 &&
thread[3][PRIORITY]<11 && thread[3][PRIORITY]>=0)
```

Here we check that the priority of thread τ_i, given by `thread[i][PRIORITY]`, never takes an invalid value. The property is satisfied for all threads.

Property 2: *A thread cannot be selected to run if it is on a low queue.*

We specify the following safety property:

```
A[] !(
(Dispatcher.S10 && run_thread==1 && thread[1][PRIORITY]==1) ||
(Dispatcher.S10 && run_thread==2 && thread[2][PRIORITY]==6) ||
(Dispatcher.S10 && run_thread==3 && thread[3][PRIORITY]==1) ||
(Dispatcher.S10 && run_thread==5 && thread[5][PRIORITY]==6) ||
(Dispatcher.S10 && run_thread==7 && thread[7][PRIORITY]==6)
)
```

which tests that under no circumstances is any of the band threads selected to run while its priority is the *low* priority of its band. Threads 4 and 6 are not band threads and are not checked. This property is also satisfied.

Property 3: `me` *always holds the most eligible runnable thread in a band.*

This property is expressed as follows, using the ability to define functions in UPPAAL:

```
A[] !( BaseScheduler.S0 && (
(thread[EDFScheduler1.me][ADEAD]>thread[EDFScheduler1.app_queue[0]
][ADEAD] && run_thread==EDFScheduler1.me && EDFScheduler1.me!=0 &&
EDFScheduler1.app_queue[0]!=0) ||
!queue_check(EDFScheduler1.app_queue) ||
```

```
(thread[EDFScheduler2.me][ADEAD]>thread[EDFScheduler2.app_queue[0]
][ADEAD] && run_thread==EDFScheduler2.me && EDFScheduler2.me!=0 &&
EDFScheduler2.app_queue[0]!=0) ||
!queue_check(EDFScheduler2.app_queue)))
```

queue_check() is a function defined in the global variables section of the model that has the following body:

```
bool queue_check(int queue[MAX_THREAD])
{
  return forall (i: int[0,MAX_THREAD-2])
        ((thread[queue[i]][ADEAD]<=thread[queue[i+1]][ADEAD] &&
          queue[i]!=0 && queue[i+1]!=0) ||
         (queue[i]==0 || queue[i+1]==0));
}
```

The <forall(i:Type) exp> expression evaluates to true, if and only if exp evaluates to true for all values of i in the range of Type. So, here the whole queue_check() function will return true if the expression in brackets is true for every i in [0,MAX_THREAD-2]. What the expression, in effect, says is that when two consecutive cells in the queue both contain thread ids, the thread closer to the head of the queue should have a shorter absolute deadline. Otherwise the queue cells can be zero.

Using this function, we construct our query, which states that, when the BaseScheduler automaton is at $S0$ (i.e. no scheduling operation is taking place) and when the me thread of a band is the actual running thread (run_thread==me), for all states in the system the following should hold true:

- me holds the most eligible runnable thread in the band,
- threads in each band's EDF queue are in absolute deadline order

The property tests true.

Exploring the behaviour The next two properties guarantee the unhindered progress of the system.

Property 4: *The system is livelock free.*

This property checks to see whether all threads reach the end of their execution. It is written as a liveness property:

```
A<> (Thread2.End && Thread3.End && Thread4.End)
```

This, essentially, tests that all paths of execution contain a state where all threads have reached their *End* node. However, since there are no outgoing transitions from the *End* node, this state must be the final state for all paths in the system. This is proof that there are no race conditions in the system. The above is how this property is written for test case A. It is written in a similar fashion for every test case. This property is also satisfied.

Property 5: *The system can never deadlock.*
This safety property can be expressed as

```
A[] (not deadlock)
```

using the UPPAAL built-in keyword `deadlock`. The property checks that under no conditions does the system come to a halt. This is extra proof of the lack of race conditions. This property is also satisfied.

5.3 Accommodating diverse scheduling policies

The essence of the Flexible Middleware Scheduling Framework is to facilitate the use of diverse scheduling policies under the preemptive fixed priority base scheduler. This can be seen on two levels: supporting the execution of tasks under a particular scheduling policy within a band, and allowing the sharing of resources within and across bands. Therefore, fully accommodating other scheduling policies means providing support for both these aspects. The following sections address these issues.

5.3.1 Supporting application of a policy within a band

To support a scheduling policy a system must be capable of two things: i) providing the necessary functionality for the policy to be able to make its scheduling decision, and ii) being compatible with the policy's scheduling triggering mechanism, i.e. allowing the policy to make a scheduling decision when it needs to. Section 4.1 briefly commented on the fact that the framework does not de facto support every scheduling policy. It presented two major reasons for this. Firstly, the presence of the framework mandates no additional functionality other than what is provided by the standard fixed-priority preemptive scheduler. Consequently, functionality that might be required for the implementation of certain policies will not be necessarily

present. Secondly, the framework's reliance on the fixed-priority scheduling inevitably means that for every application-defined scheduler the underlying scheduling model will be:

- preemptive scheduling;
- event-triggered scheduling; application scheduling decisions can be taken only on the occurrence of certain events, which have been described with the notion of a scheduling point.

Therefore, all preemptive event-triggered schedulers can easily be implemented in the framework. However, not all scheduling policies are preemptive or event-triggered. Three other types of policies stand out: non-preemptive policies, time-driven scheduling and job-level dynamic eligibility algorithms.

Non-preemptive scheduling .

Non-preemptive scheduling policies are the simplest of the three to implement. The application scheduler will select a new task to run in only the following situations:

- during a *release* operation, if no task is currently running,
- during any operation where the running task voluntarily suspends itself (e.g. *suspend*, *wait*, *yield*),
- during an *end* operation

During all other scheduling operations the application scheduler, when asked, simply returns the currently running task.

Time-triggered scheduling

In contrast to event-triggered scheduling schemes, time-triggered scheduling policies apply scheduling decisions at specific time instants. The most typical examples of this category are off-line scheduling, the round-robin scheme, and synchronous reactive systems. Under the preemptive fixed-priority scheduling the framework is based on, the passage of time is not factored into making a scheduling decision. As a result, time-triggered approaches are not directly supported by the framework. Nonetheless, there are ways of implementing time-triggered scheduling by introducing "time events" in the system. This way we can transform time-triggered policies to event-triggered. Since round-robin is a well known and easily understood case

of time-triggered scheduling, its emulation is provided as an example. As we will see, the solution given can only be implemented if CPU time clocks are available.

To enforce the round-robin policy the application scheduler could make use of some sort of timer to introduce scheduling points at regular intervals. Since scheduling points are linked to the behaviour of tasks, this scheme will have to involve a "round-robin scheduling task" running in the band on behalf of the scheduler. A possible behaviour for this task can be seen in Figure 5.17 below.

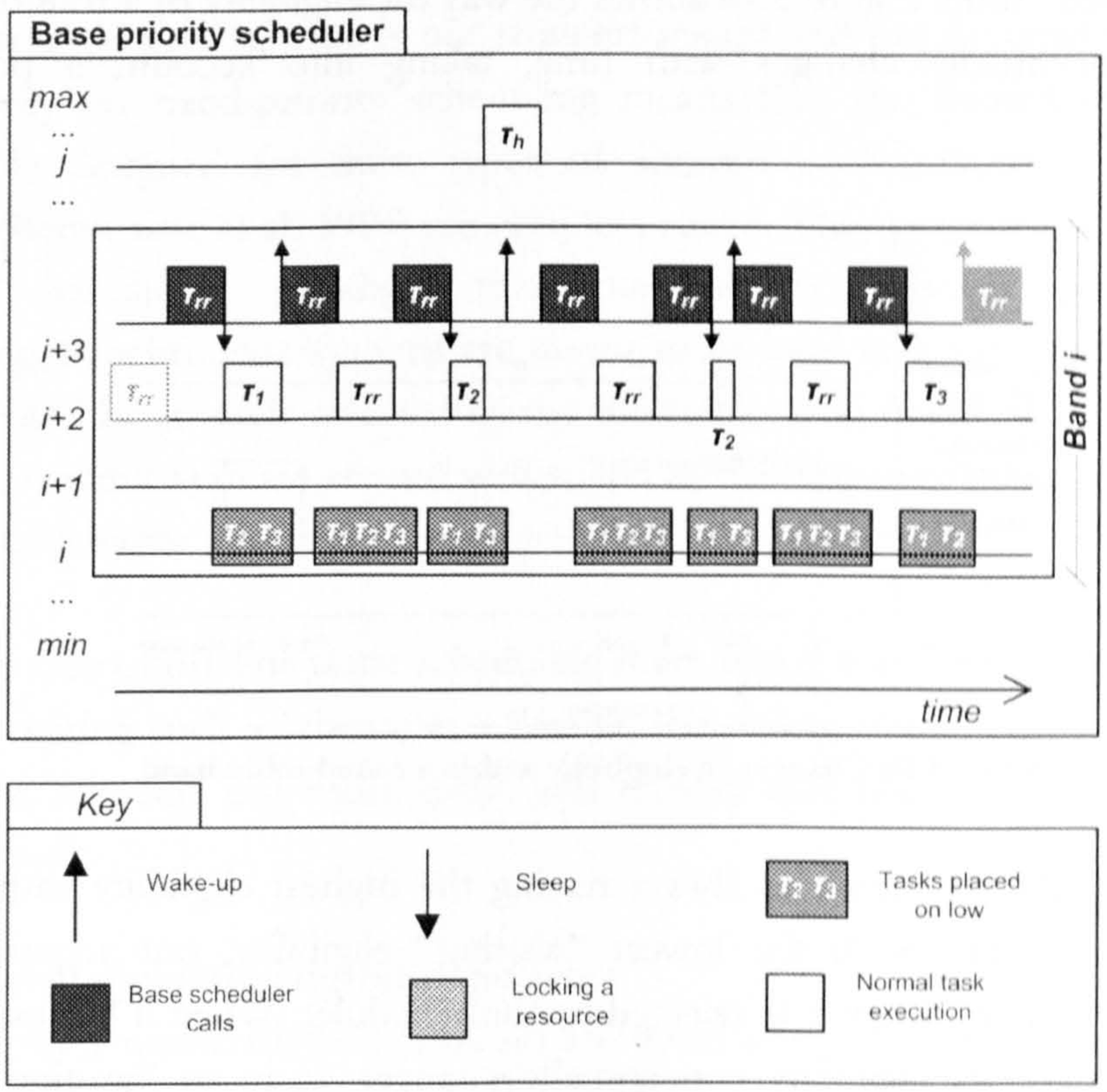


Figure 5.17: Round-robin scheduling within a band

In this figure task τ_{rr} is the round-robin scheduling task for band B_i and has the highest eligibility in the band. This task executes in an endless loop. All other band tasks have the lowest eligibility in the band. Therefore, task τ_{rr} will always be selected to run in preference to all other tasks. The only action this task takes is to call for a suspending sleep operation. Before suspending, it asks the application scheduler, as per normal framework protocol rules, to designate the next task to run. The application scheduler determines, with the help of facilities equivalent to the execution time monitoring found in POSIX [IEEE 2004], how much time its most eligible task has left to execute in order

to complete one quantum, and directs τ_{rr} to "sleep" for that duration. The selected task executes until τ_{rr} wakes up again, preempting it. At that point, τ_{rr} executes again, just enough so that it can call its next sleep operation, at which point the procedure repeats itself until all tasks have finished. This technique also addresses the case where a round-robin task might get preempted by higher priority level tasks before finishing its quantum. This can be seen in Figure 5.17, where task τ_2 gets preempted by τ_h. τ_2 did not have the chance to execute part of its quantum and this part must be given back to it. Therefore, the next time τ_{rr} wakes up and asks the round-robin scheduler for its next task, the scheduler again specifies task τ_2 and instructs τ_{rr} to sleep for the duration of the "stolen" time. Figure 5.18 shows the way the eligibility of a round-robin task τ conceptually changes with time, taking into account a possible preemption.

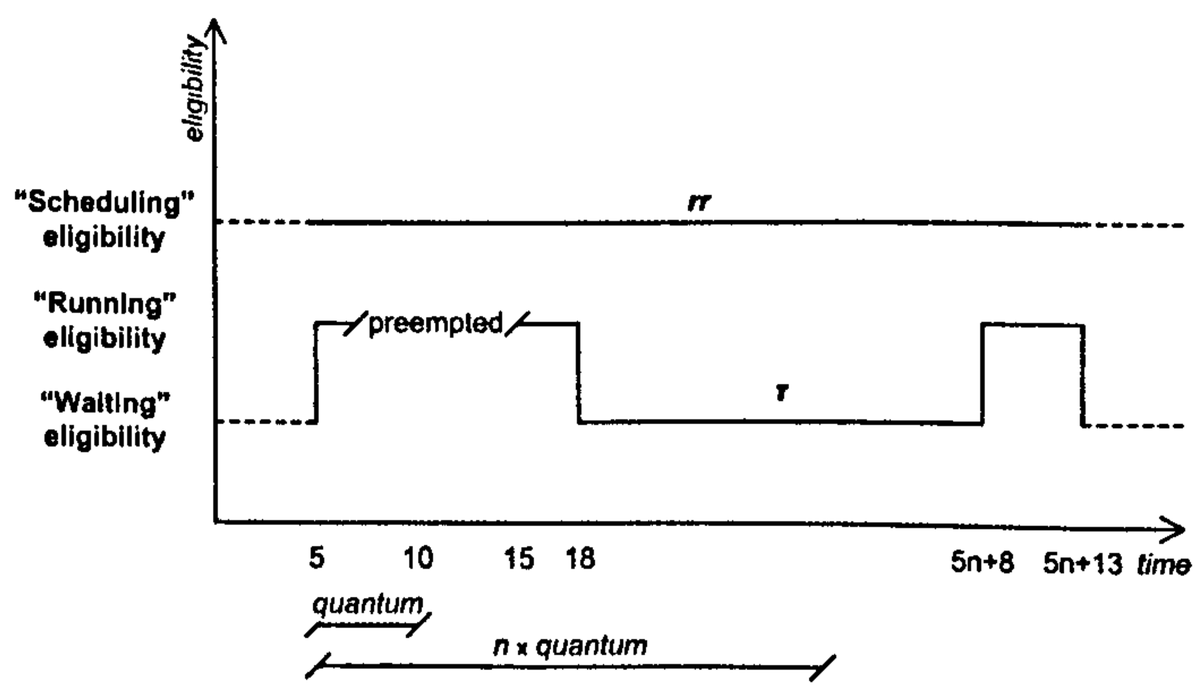


Figure 5.18: Changes in eligibility within a round-robin band

Here, task τ_{rr} is shown as always having the highest eligibility within the band. Task τ starts with the lowest "waiting" eligibility, but acquires the "running" eligibility when it is selected by the scheduler. When it has executed for a quantum, its eligibility conceptually changes again to "waiting", and another task is selected to run. This is true even if the task gets preempted, as can be seen in Figure 5.18. At time 5, when τ is selected to run, τ_{rr} calls a sleep for 5 units, since that is the size of the quantum. In the meantime, the band gets preempted by higher priority tasks and, therefore, task τ does not have a chance to finish its quantum. Neither is τ_{rr} able to execute at time 10. However, τ_{rr} executes at time 15, when the preemption ends, since its priority is the *high* priority of the band. τ_{rr} asks the round-robin scheduler for its next task and the scheduler returns τ, because it did not have the chance to finish its quantum. τ now executes until time 18 when it is preempted by τ_{rr}. After all round-robin tasks have executed in the CPU for $n \times quantum$ time units, where n is the number of tasks in the band, task τ will again be selected to run.

Round-robin scheduling finds application when tasks need to progress at a similar rate. It is part of the POSIX standard [IEEE 2004] and has also been introduced in the Ada language [Burns et al. 2003]. If CPU time clocks are available, it can be accommodated by the framework, as demonstrated above, but at an overhead cost. Of course, if the framework was to be implemented in a system where the preemptive fixed-priority scheduler could use either FIFO or round-robin within priorities, then round-robin scheduling could be provided directly by the base scheduler outside of a band.

In general, implementing even a simple time-triggered scheduling policy like the round-robin scheme is not straightforward and can be unwieldy. The dependency on fixed-priority scheduling means that the framework is not specifically designed for these types of algorithms. Synchronous reactive systems [Benveniste et al. 1994] are even less suited. They present a completely different scheduling paradigm than that of fixed-priority preemptive scheduling. Synchronous systems are meant to be hard real-time systems and, hence, would be severely affected by the inevitable extra overheads. Generally speaking, reactive systems are not well-suited to the framework and will not be further considered.

In the case of off-line static scheduling there is arguably little point, if at all, in incorporating such a scheme in a flexible scheduling framework. Execution of a static schedule can more easily and reliably take place using non-band priority levels.

Job-level dynamic eligibility policies

Job-level dynamic eligibility policies are those where the eligibility of a task changes throughout the course of the task's current release. An example of such a policy is the least slack time first algorithm (LST). Slack time s is defined as $s=d-t-C^{remaining}$, where d is the absolute deadline, t the current time and $C^{remaining}$ is the remaining task execution time of a task for its current job. The less a task's slack time the greater its eligibility. As is evident from the equation, a task's eligibility is monotonically non-decreasing with time. More importantly, eligibility can increase even though a task might not be executing.

There are three versions for every job-level dynamic eligibility policy, the *strict*, the *non-strict* and the *static* version [Liu 2000]. Enforcing the strict version means constantly monitoring the eligibility of all tasks and comparing them

with the eligibility of the running task. A context switch has to take place if the eligibility of the running task falls below the eligibility of another task. On the other hand, enforcing the non-strict version means making scheduling decisions only when specific events arise, such as task release or completion. Finally, the static version of a job-level dynamic policy is not really job-level dynamic, but rather a translation of the policy to a job-level static policy. This is achieved by considering the eligibility of a task for the whole duration of its release to be equal to its initial eligibility at the start of its release. In the case of LST, for example, this would mean that the eligibilities equation is written $s=d-t_0-C^{initial}$. Without loss of generality, we can consider the initial release of each task to be at $t_0=0$. The equation can now be written $s=d-C^{initial}$ which is the same as $s=D-C^{initial}$, where D is the task's relative deadline. The static version is, by default, also non-strict.

As we can see, the strict version means that the scheduler should be constantly aware of the eligibilities of its tasks and preempt tasks as needed. Scheduling decisions are not just triggered due to specified events, but could happen at any time, as required by the current eligibilities of tasks. As mentioned in [Liu 2000:p.121] the strict version of LST leads to tasks being scheduled in round-robin manner, when a runnable task's eligibility reaches the eligibility of the running task. To implement such a policy, an application scheduler would have to actively keep track of the eligibilities of its tasks. However, an application scheduler is a passive entity in the framework and cannot carry out such a task. Therefore, the only option would be to have a dedicated task, as in the case of round-robin illustrated above, which would cause a system schedule at regular intervals, emulating the application scheduler's "tick". This, again, would be an awkward, far from ideal solution.

The non-strict version and static versions, on the other hand, apply scheduling decisions at specific events. This coincides with the event-triggered scheduling model of the base scheduler. Therefore, as far as the scheduling triggering mechanism is concerned, the non-strict and static versions of job-level dynamic eligibility algorithms can be implemented as an application scheduler in the framework. It is a matter of the middleware if the necessary functionality for implementing the policy is present (e.g. CPU timers).

5.3.2 Sharing resources under a new policy

With respect to sharing resources, there are two possibilities for an application-defined policy: it can either use the framework's BPreCP protocol

or it can make use of its own resource sharing protocol, provided that no tasks outside its band will want to use resources that belong to its band. We will examine both cases.

5.3.2.1 Using an application-defined resource sharing protocol

Section 4.5.4 explained that it is possible for an application scheduler to use its own policy-specific resource sharing protocol for those resources that are used solely by tasks of its own band. As seen in Section 4.6, the base scheduler does not perform the preemption level test when locking takes place on a resource that it identifies to be governed by a protocol other than the framework's BPreCP protocol. Resources could also not be associated with any protocol, if that is deemed appropriate. This means that resource sharing is not an issue, when it comes to a stand-alone implementation of any policy that can be supported according to the guidelines of the previous section.

Of course, as already pointed out, maximum integration with the framework means being also able to use resources of other bands, and this requires the use of preemption levels. However, this does not mean that the application scheduler needs to fully implement the BPreCP. To understand this we need to understand the mechanism that is effected when a task uses a resource outside its own band. When accessing such a resource, the task's own application scheduler does not need to perform any function. Indeed, the FMSF_PLT is carried out by the base scheduler, as is priority inheritance. Eligibility inheritance is carried out solely by the scheduler of the locking band. Therefore, an application scheduler could be using its own application-defined protocol for resources used solely by its own tasks, while at the same time its tasks would be able to lock outside their own band. The only requirement is that each task locking outside its own band must have a relative preemption level assigned, even if the band's own protocol does not make use of preemption levels. The reason for this is that the base scheduler must be able to compute the task's absolute preemption level, in order to carry out the FMSF_PLT for the resource in question. Here it is worth pointing out that a band always has a range of absolute preemption levels statically assigned to it at the point of its creation, even if it does not make use of the BPreCP.

The assignment of preemption levels to tasks is almost trivial for a band that uses its own resource sharing protocol. As explained in Chapter 3, the preemption level test guarantees deadlock avoidance and assists in minimising eligibility inversion. However, in the case of a band that specifies its own

resource sharing protocol, the FMSF_PLT is not used when locking takes place within the band (hence, not used for minimising inversion), but only to guarantee deadlock avoidance when tasks use resources outside the band.

To perform the FMSF_PLT the base scheduler must know the absolute preemption level $|\pi|$ of the locking task. The value of $|\pi|$ for a task, in turn, is calculated based on the task's relative preemption level π and the band the task belongs to. The scheduling policy governing a band does not affect the position of the band in the priority range. Therefore, the only issue to consider with respect to guaranteeing that tasks will not deadlock is the way relative preemption levels are assigned.

Section 3.2 presented Baker's preemption level assignment rule (Rule 3.1). We pointed out that this rule is an *optimal assignment rule.* This means that it provides for minimum eligibility inversion. However, this assignment is irrelevant in our case, since we have explained that we are only interested in the deadlock avoidance aspect of the FMSF_PLT. Moreover, we will show the stronger assertion that there is no assignment rule that needs to be followed in order to achieve deadlock avoidance. In other words, the ability of the protocol to avoid deadlocks is totally independent of the particular assignment of relative preemption levels.

It is easy to understand this by considering the function that preemption levels fulfill. Relative preemption levels essentially provide a total ordering of a band's task set based on a particular metric. Since there is a "1-to-1" relation between a relative preemption level within a band and an absolute preemption level, and since a task can only belong to one band at a time, it follows that the total ordering of tasks within each band translates to a total ordering of all band tasks in the system. Furthermore, since we have assigned absolute preemption level values to non-band priority levels as well, we have a *total ordering of all tasks in the system* based on absolute preemption levels. That is to say that for two arbitrary tasks in the system τ_1 and τ_2, irrespective of the bands they belong to (and thus irrespective of the policy that schedules them), exactly one of the relations $\pi(\tau_1) < \pi(\tau_2)$, $\pi(\tau_1) = \pi(\tau_2)$, $\pi(\tau_1) > \pi(\tau_2)$ holds. Furthermore, resource ceilings, according to Definition 4.7, are set to the highest absolute preemption level amongst those of the tasks that use them. As a result, the set of resources in the system that are used by band tasks also becomes a totally ordered set.

Let us consider, now, that relative preemption levels are assigned to tasks in a random manner. These preemption levels will still correspond to a single absolute preemption level each. Therefore, the task set will still be totally ordered, even though the order will not have any particular meaning. Moreover, Definition 4.7 still applying, the case will still be that if $\pi(\tau) > \lceil r \rceil$ then τ must not be using r. Since the definition of the system ceiling (Definition 3.3) also remains unchanged, the FMSF_PLT will succeed or fail for the exact same reasons as before. That is, a task passing the test will never need any of the locked resources. Therefore, even with arbitrary relative preemption levels, the FMSF_PLT guarantees deadlock-free execution of tasks.

It should be clear that this result applies to the way relative preemption levels are assigned in all bands, regardless of the resource sharing protocol used within the band. Therefore, all band tasks can avoid deadlock through the use of the FMSF_PLT, irrespective of their preemption levels. For bands that do not use the FMSF, however, it has the particular implication that they can be using the FMSF_PLT to avoid deadlocks in resource sharing, while using another resource sharing protocol to minimise eligibility inversion, or even while not using any protocol.

5.3.2.2 Using the FMSF

In using the FMSF eligibility inversion protocol a scheduling policy will seek two guarantees: deadlock avoidance and minimal eligibility inversion. As has been pointed out in the previous section, a band could choose to use only the FMSF_PLT for deadlock avoidance. However, if it also needs minimal eligibility inversion then it must combine use of the FMSF_PLT with eligibility inheritance at the application scheduler level. Although deadlock avoidance can be achieved with an arbitrary assignment of preemption levels, minimising eligibility inversion through the FMSF_PLT necessitates an optimal assignment of preemption levels. Moreover, the application scheduler must implement the FMSF eligibility inheritance rule. Therefore, examining the ability of different scheduling policies to bound eligibility inversion needs to be done on two levels: i) examine how effective eligibility inheritance can be under a specific policy and ii) examine if the optimal preemption level assignment rule is applicable under the specific policy.

The key in examining these two factors is the observation that they are both linked to eligibility, since Rule 3.1 assigns preemption levels according to the eligibilities of tasks at time t=0. Under any policy, eligibility is a function of time and of certain task parameters, like a task's relative deadline, its period, its release time etc. These task parameters typically stay fixed throughout a task's execution. As far as their time parameter is concerned, we can distinguish three types of eligibility functions: fixed eligibility, task-level dynamic eligibility, job-level dynamic eligibility. A fixed eligibility function does not change its value over time, or, in other words, the coefficient of time is zero, e.g. any form of fixed-priority scheduling, like shortest execution time first (assuming the execution time of a job does not change between releases). A task-level dynamic eligibility function is static with regard to time during a task's release (job) but might change between different releases, e.g. EDF scheduling. Therefore, task-level dynamic eligibilities are always step functions of time. Job-level dynamic eligibilities change constantly with time, e.g. strict least slack time first (LST), and can theoretically be either non-decreasing, non-increasing or non-monotonic.

Eligibility inversion happens when a task τ_2 preempts a locking task τ_1, while a task τ_3 blocked on the system ceiling has the highest eligibility. For static eligibilities we avoid this by elevating the eligibility of τ_1 to that of τ_3. Section 3.3 demonstrated that BPreCP with fixed task eligibilities amounts to the PCP. Therefore, policies that apply fixed eligibilities (i.e. fixed priorities) to tasks will get the same behaviour from the protocol as if it was PCP. In task-level dynamic eligibility policies there can again be no violation of the protocol because the eligibilities of tasks τ_1, τ_2, τ_3 remain static for as long as the tasks participate in the eligibility inheritance situation.

Job-level dynamic eligibility policies can be split into two categories:

a) policies where the ordering of tasks according to their eligibilities remains static throughout their release (even though eligibilities change),

b) policies where the ordering of tasks according to their eligibilities changes during their release.

Policies belonging to the first group are able to use the protocol because they provide the certainty that a task will always be either more or less eligible than another task for the whole duration of its release. This is the same guarantee that task-level dynamic eligibility policies provide. Job-level dynamic

eligibility policies with static task ordering produce eligibility graphs that never intersect for two different tasks. Or, to state it differently, if equation $e(\tau_i,t)-e(\tau_j,t)=0$ for any i,j has any solutions, it will be for $t \leq 0$. Three imaginary examples can be seen in Figure 5.19.

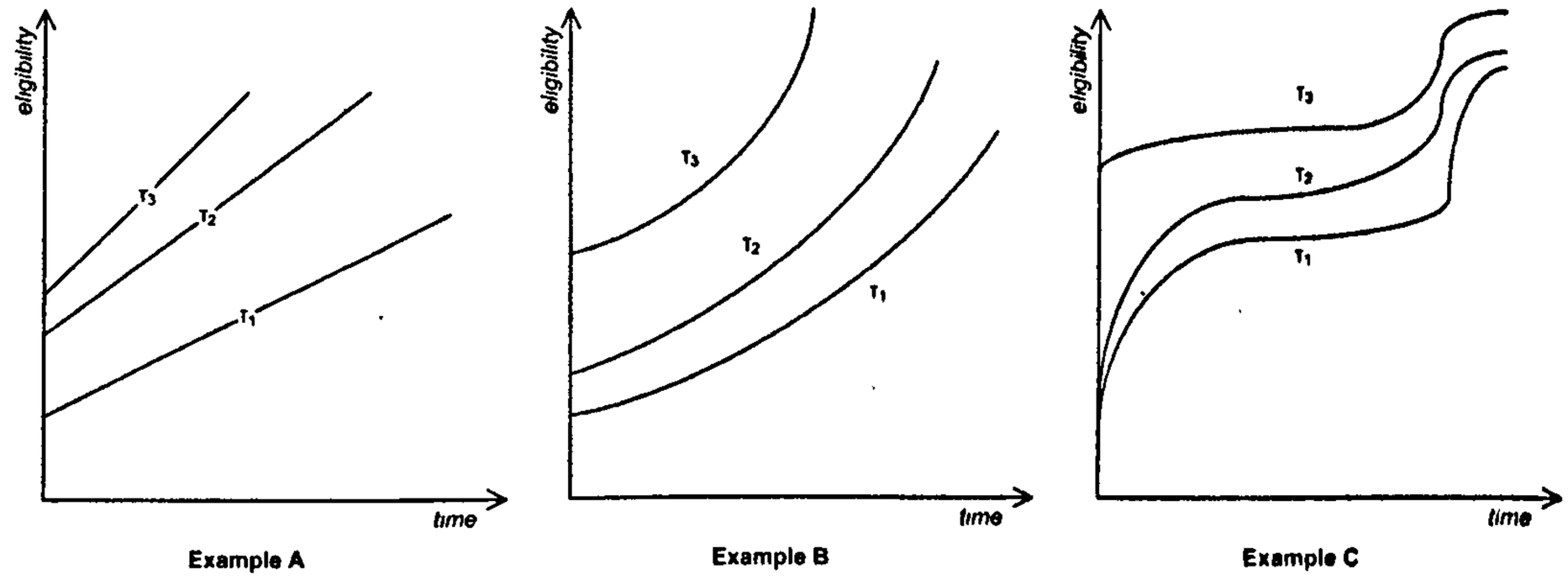


Figure 5.19: Eligibility in job-level dynamic policies with static task ordering

Next, let us examine the effectiveness of the eligibility inheritance rule in the case of job-level dynamic eligibilities with dynamic task ordering. Such policies produce eligibility graphs that can intersect for two different tasks. In other words, eligibilities of different tasks can change at different rates. This can be seen in Figure 5.20.

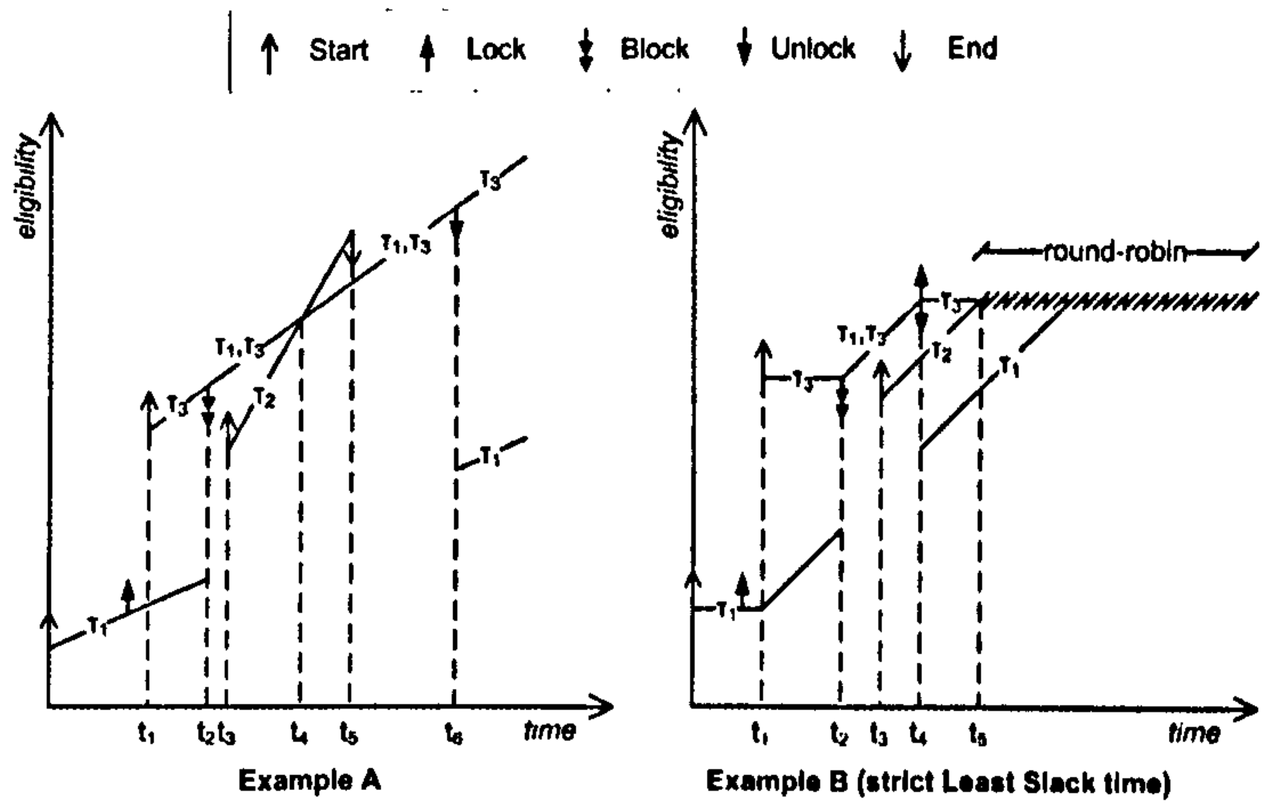


Figure 5.20: Eligibility in job-level dynamic policies with dynamic task ordering

To understand how eligibility inheritance would work in such a case, let us consider the three tasks τ_1, τ_2, τ_3, in Example A of Figure 5.20. The relation between their initial eligibilities, if they were all to be released at t=0, is $e_o(\tau_1) < e_o(\tau_2) < e_o(\tau_3)$. Furthermore, their eligibilities are increasing with time. Let us assume tasks τ_1 and τ_3 use the same resource, τ_1 is released first and proceeds to lock the resource. Task τ_3 is released at time t_1 and preempts τ_1.

At time t_2, task τ_3 blocks trying to lock the resource. Let us suppose that τ_1 inherits the eligibility of τ_3. In other words, the active eligibility of τ_1 is equal to the eligibility of τ_3. During the time from t_2, when τ_3 blocks, to t_6, when the resource is unlocked, τ_1 can be preempted by only two sets of tasks: i) tasks that are released during this time and have greater eligibility than τ_3, and tasks whose eligibility exceeds that of τ_3 during this time. Both sets of tasks are, obviously, eligible to preempt and moreover, they can be included in any worst-case response time analysis for τ_3. Therefore, their preempting of τ_1 would not constitute eligibility inversion. In the example, this is true of task τ_2, which is released at t_3, but cannot preempt. However, at t_4 the eligibility of τ_2 exceeds the eligibility of τ_3 and, therefore, also the active eligibility of τ_1. Therefore τ_2 preempts τ_1. Any task that would be able to preempt τ_1 but not τ_3 is prevented from introducing eligibility inversion by means of the eligibility inheritance.

The same is true for the LST algorithm, which is shown as Example B in Figure 5.20. LST is a special case of a job-level dynamic policy. Tasks can be released with different eligibilities, but eventually they will all acquire the same eligibility and the policy is reduced to the round-robin scheme, as is the case in the example at time t_5. Before entering the round-robin phase, the algorithm produces a static task ordering. This can be seen in the figure in the interval $[t_2, t_4]$, where τ_1 acquires the eligibility of τ_3, which is blocked and thus has an increasing eligibility. Without this eligibility inheritance τ_2 would be able to preempt τ_1, while having lower eligibility than τ_3, thus introducing eligibility inversion. Once in the round-robin phase, the task ordering becomes dynamic. Since the only time that the ordering is dynamic is during the round-robin phase, there can be no eligibility inversion, because all tasks participating in the round-robin essentially have the same eligibility. Again, it is possible to include this preemption in a worst-case response time analysis.

Therefore, for any type of eligibility, application of eligibility inheritance bounds the extent of eligibility inversion. Let us now examine the effect of eligibility on the preemption level assignment rule.

It has already been mentioned that the assignment of preemption levels under Rule 3.1 is optimal. Clearly, an arbitrary assignment of preemption levels would undermine this optimality. As an example, let us assume we have two tasks τ_1 and τ_2, with eligibilities $e(\tau_1) < e(\tau_2)$, and we assign preemption levels to them such that $\pi(\tau_1) > \pi(\tau_2)$. Furthermore, we assume that τ_1 uses resource

r_1 while τ_2 uses r_2. If τ_2 is released while τ_1 is locking r_1, it will block due to the FMSF_PLT when it tries to lock r_2, even though it has a higher eligibility and is not using r_1. Therefore, because preemption levels are static, their inappropriate assignment to tasks introduces unwanted eligibility inversion.

Rule 3.1 assigns preemption levels according to task eligibilities at the instance of their release, which can be taken to be t=0 for all tasks. If we hypothesise that all tasks are released at t=0, then higher eligibility for a task at that instance must also mean a higher preemption level. Thus, a total ordering is created for all tasks under a particular scheduling policy. The rule is not concerned with absolute values of eligibility but with the relative ordering of tasks based on their eligibilities. So, in order to determine if and when the rule can be broken, we must ask whether there is a way for this ordering to change during a task's release.

We have already seen that such a change is possible in job-level dynamic eligibility policies with dynamic task eligibility ordering. For such eligibility preemption levels can not always be allocated optimally. This is because there is always a possibility that two tasks τ_1, τ_2 with initial eligibilities $e(\tau_1) < e(\tau_2)$ will have their eligibilities reversed, as time progresses. However, due to the preemption level assignment rule, their preemption levels will be $\pi(\tau_1) > \pi(\tau_2)$ for the whole duration of their release which leads to the problem outlined above. Using dynamic preemption levels, i.e. preemption levels that increase or decrease in value as the task's eligibility increases or decreases, would help avoid eligibility inversion in some situations, but at the cost of breaking the FMSF_PLT and allowing deadlocks. To understand this we can look at Figure 5.21. Tasks τ_1 and τ_2 have job-level dynamic eligibilities with the task ordering dynamically changing at time t_2. Let us assume that their preemption levels also change dynamically, proportionately to their eligibilities. Now, let us assume that τ_1 locks a resource r_1 at time t_1, setting the system ceiling to $\bar{\pi}=\pi(\tau_1,t_1)$. At time t_4 task τ_2 tries to lock a resource r_2, not used by τ_1, and succeeds, because its pre-emption level at that point is greater than the system ceiling $\pi(\tau_2,t_4)>\pi(\tau_1,t_1)$. This allows τ_2 to lock r_2 whereas with static pre-emption levels it would have blocked. However, let us now consider the situation where the two tasks share the two resources r_1 and r_2. Let us assume that after τ_1 locking r_1, τ_2 locks r_2 at time t_3.

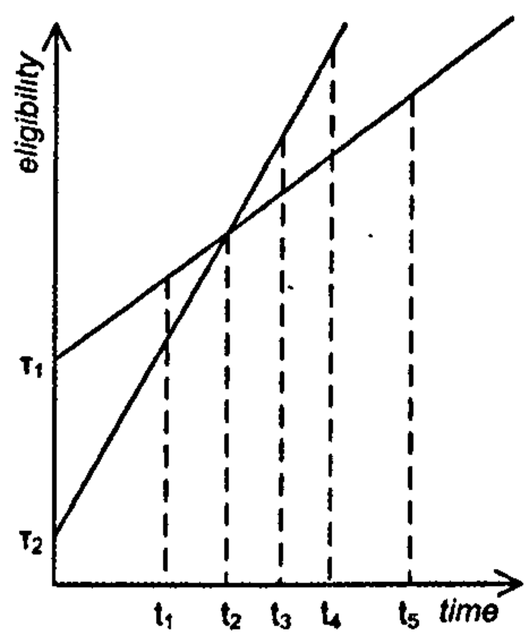


Figure 5.21: Two tasks with job-level dynamic eligibilities

It will be able to do this, because $\pi(\tau_2,t_3)>\pi(\tau_1,t_1)$. Then, at time t_4, τ_2 tries to lock r_1. It will again pass the FMSF_PLT, only now r_1 is locked. If there is no other mechanism for managing the sharing of resources, τ_1 and τ_2 will be using r_1 simultaneously, a critical error. If there is another lower-level mechanism present that keeps τ_2 from using r_1, then τ_2 will block, τ_1 will execute, and when at time t_5 it tries to lock τ_2 a deadlock will occur. Therefore, dynamic preemption levels are not a solution.

From the above we can understand that as long as a policy produces a static ordering of tasks based on their eligibility, the optimal assignment rule (Rule 3.1) will always produce preemption levels that do not produce unnecessary blocking. In the case of dynamic task ordering policies the optimal assignment rule is broken.

5.4 Summary

The contributions of this chapter are to demonstrate:

1. How an application-defined scheduler can be constructed, in particular an EDF scheduler, according to the requirements set by the framework. It has been shown that three main constructs are adequate to support the framework's protocol.
2. That the framework protocol can be implemented free of race conditions.
3. The range of scheduling policies able to be incorporated in the framework. More specifically, it has been shown that all scheduling policies can be supported, as far as supporting their scheduling decisions is concerned, but strict job-level dynamic eligibility policies are awkward and not as efficient to implement. As far as being able to use a resource sharing protocol, it has been shown that new policies can introduce their own protocol, with an option to use the FMSF_PLT for deadlock avoidance, or they can choose to comply with the framework's protocol. The only problematic category is job-level dynamic eligibility policies with dynamic task ordering, for which the optimal assignment of preemption levels is not valid.

The chapter has helped to show the applicability and viability of the flexible middleware scheduling framework in addressing a wide range of scheduling needs.

Chapter 6

Implementing the Framework

This chapter continues the evaluation of the Flexible Middleware Scheduling Framework (FMSF), taking over from the previous chapter. However, whereas Chapter 5 concentrated on a theoretical evaluation of the framework, this chapter provides a practical evaluation based on a simulated implementation in the Real-Time Specification for Java (RTSJ). The rational for choosing RTSJ has been the wide acceptance of the Java platform, in general, and the growing relevance of the RTSJ for real-time computing. This guarantees that the evaluation will be relevant to a significantly broad review audience.

This chapter's purpose is three-fold. Firstly, it demonstrates that the FMSF framework can be feasibly implemented. Secondly, and more importantly, by describing how the framework can enhance the RTSJ with the addition of flexible scheduling this chapter highlights the framework's merits and ability to be easily introduced to an existing middleware system. Last but not least, it presents measurements on the execution time overhead that the framework

introduces, proving that the framework can viably support real-time applications.

Section 6.1 gives a brief description of the RTSJ and its deficiencies with regard to scheduling. Section 6.2 presents the implementation design for the incorporation of the FMSF into the RTSJ, the changes needed in the specification to fully support our approach, and the impact on feasibility analysis. The description is mainly kept at the class level, avoiding cumbersome implementation details. Section 6.3 describes a lightweight simulated implementation of the framework, which is used in Section 6.4 to calculate execution time overheads. Section 6.4 discusses the timing results from a set of test applications. It is deemed that these results are satisfying and reinforce the viability of a full framework implementation.

6.1 The Real-Time Specification for Java

The Real-Time Speciation for Java (RTSJ) [Bollella et al. 2000], [Belliardi et al. 2006] provides a framework from within which real-time scheduling can be performed for single and multi-processor systems[14]. The provided API forms the `javax.realtime` package. The RTSJ enhances Java in seven areas: thread scheduling and dispatching, memory management, synchronization and resource sharing, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, and physical memory access. Of these areas thread scheduling and synchronization are the two most relevant with regard to the FMSF framework. It is important to note that the intention of the RTSJ is to support a range of schedulers, all of them conforming to the `abstract Scheduler` class. However, the current specification defines only a base scheduler, the `PriorityScheduler`. The RTSJ provides a series of classes to specify the scheduling needs and behaviour of a real-time thread. The `SchedulingParameters` class and its subclasses (most notably the `PriorityParameters` class) provide the parameters to be used in scheduling the thread. The `ReleaseParameters` class and its subclasses (`PeriodicParameters`, `AperiodicParameters`, `SporadicParameters`) provide information about the execution behaviour of a real-time thread, although some parameters, like the relative thread deadline, can also be used

[14] Although the RTSJ is intended for use on both single and multiprocessor systems, the specification is silent on specific multiprocessor issues. In this thesis we will consider only single processor implementations.

for scheduling purposes by particular schedulers, e.g. an EDF scheduler. Also of particular importance is the functionality offered by the `ProcessingGroupParameters` class, which groups real-time threads together, guaranteeing that they will not be collectively given more time per period than indicated by a `cost` parameter, and which could be used in an implementation of temporal isolation between threads of different scheduling bands in the FMSF. As far as thread synchronization is concerned, the RTSJ allows the setting of a priority inversion control policy either as the default or for specific objects. Priority inheritance is the default protocol while priority ceiling emulation is supported as optional.

The RTSJ scheduling framework can be summarized by describing how it addresses the three components of scheduling: the scheduling policy, the scheduling mechanism and the feasibility analysis [Burns and Wellings 2001].

Scheduling Policy: The RTSJ uses the notion of "execution eligibility" for "schedulable objects" to determine their execution order. Execution eligibility is encapsulated in the `SchedulingParameters` class and its subclasses `PriorityParameters` and `ImportanceParameters`. For the base scheduler, priorities are assigned by the programmer and the scheduler enforces priority inheritance algorithms when resources are accessed. Hence, it supports the notion of *base* and *active* priorities.

Scheduling Mechanism: For the base scheduler, the RTSJ requires pre-emptive priority-based dispatching of schedulable objects. An executable schedulable object with the highest active priority is always executing on the processor at any given time. However, the RTSJ makes no statement on whether it supports "pre-emptive *execution eligibility* dispatching" in general.

Feasibility Analysis: The RTSJ requires no specific feasibility analysis to be implemented. The default analysis always returns true if the application contains only periodic and sporadic schedulable objects, and returns false if aperiodic schedulable objects are present.

Whilst it is clear that the RTSJ's intention is to support different (and possibly multiple) schedulers, it is far from clear that the provided framework is adequate for this purpose. Furthermore, it is unclear the extent to which priority-based dispatching is so ingrained in the specification that all other schedulers must express "execution eligibility" in terms of priority. In part, this

is due to the variety of execution environments in which an application may execute. There are at least three ways by which an RTSJ application can be executed [Lindholm and Yellin 1999]:

1. The application runs as a process on top of a real-time operating system. The RTSJ library and virtual machine (VM) supports a native threads model with each schedulable object having an associated operating system real-time thread (although not necessarily a one-to-one mapping). Run-time dispatching (the scheduling mechanism) is provided by the operating system.
2. It runs on top of bare hardware. The RTSJ library and virtual machine have full control over the hardware resources and implement their own scheduling mechanism.
3. It runs on top of a hardware-implemented real-time virtual machine. Again the scheduling mechanism can be implemented by the RTSJ library and virtual machine.

Whatever the execution environment, the "write-once carefully, run-anywhere conditionally" principle (a more realistic version of the "write once, run anywhere" principle) dictates that the RTSJ should define its scheduling mechanism. Most real-time operating systems support pre-emptive priority-based dispatching. Consequently, the RTSJ should arguably explicitly define this as the only scheduling mechanism. However, many modern applications require more flexible scheduling [Brandt et al. 2003], [Regehr et al. 2000]. Furthermore, some applications may need to be scheduled by one policy while others may need a different policy; e.g. fixed priority for hard real-time threads and EDF for soft real-time threads. Hence, state-of-the-art real-time OS nowadays move towards supporting application-defined scheduling [Gwinn 2004], often in the form of hierarchical scheduling [FIRST 2005]. Support for these kinds of applications is not readily available in the RTSJ.

In this chapter we propose that the RTSJ adopts the Flexible Middleware Scheduling Framework's (FMSF) hierarchical scheduling model, described in Chapter 4, on top of its own fixed-priority scheduler. This will transform the existing RTSJ scheduling model to a two-level scheduling scheme, with the RTSJ's priority scheduler performing the actual dispatching. Under this scheme an application will be able to introduce its own scheduler, which can have its own notion of execution eligibility. Adopting this approach also provides a framework within which multiple application-defined schedulers

can be integrated. It is also sympathetic to the notion that priority-based scheduling is more ingrained in the RTSJ than intended, and that a more general scheduling mechanism would require more fundamental changes to the RTSJ than is acceptable to the community. This way, the RTSJ's scheduling deficit can be suitably addressed, thus fully realising the RTSJ's scope for supporting "both hard and soft real-time applications" [RTSJ 2007].

The next section describes the augmented RTSJ scheduling model. The remainder of this chapter assumes that priority changes that require OS intervention occur immediately and are not deferred. Also, the terms "thread" and "schedulable object" are used interchangeably, as are the terms "application-defined scheduler" and "user-defined scheduler".

6.2 Applying the FMSF to RTSJ

To apply the Flexible Middleware Scheduling Framework in the RTSJ a number of classes must be introduced in the RTSJ API accompanied by changes to existing classes and to the virtual machine [Zerzelidis and Wellings 2006]. The following sections explain these modifications in detail.

6.2.1 Changes to the `PriorityScheduler` class

The `Scheduler` class is, perhaps, the most fundamental class of the scheduling part of the RTSJ API. It is an `abstract` class intended to be the base class for any implementation of any scheduling policy. In other words, all scheduler classes should inherit from this class [Belliardi et al. 2006]. The current RTSJ `Scheduler` class hierarchy is shown in Figure 6.1 below. The RTSJ defines only one scheduler, the `PriorityScheduler`.

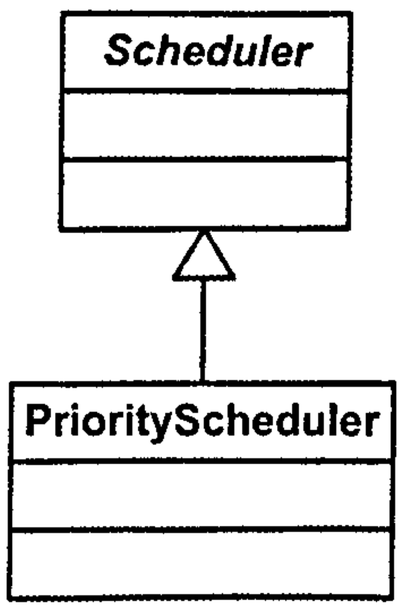


Figure 6.1: Existing RTSJ `Scheduler` class hierarchy

Currently the `Scheduler` class is defined as follows:

```
package javax.realtime;
public abstract class Scheduler {
  // constructors
  protected Scheduler();
  // methods
  protected abstract boolean addToFeasibility(
      Schedulable schedulable);
  protected abstract boolean removeFromFeasibility(
    Schedulable schedulable);
public abstract boolean isFeasible();
public abstract boolean setIfFeasible(
    Schedulable schedulable, ReleaseParameters release,
    MemoryParameters memory);
public abstract boolean setIfFeasible(
    Schedulable schedulable, ReleaseParameters release,
    MemoryParameters memory,
    ProcessingGroupParameters group);
public abstract boolean setIfFeasible(
    Schedulable schedulable, SchedulingParameters scheduling,
    ReleaseParameters release, MemoryParameters memory,
    ProcessingGroupParameters group);
public abstract void fireSchedulable(
    Schedulable schedulable);
  public static Scheduler getDefaultScheduler();
  public abstract java.lang.String getPolicyName();
  public static void setDefaultScheduler(
          Scheduler scheduler);
}
```

As can be seen from this specification, the `Scheduler` is mainly concerned with manipulating the feasibility set and performing feasibility analysis. Only the `fireSchedulable()` method is concerned with scheduling and the `PriorityScheduler` does not support this method[15]. In other words, although the scheduler is responsible for releasing schedulable objects, monitoring deadline misses and cost overruns, implementing the required priority inheritance algorithm etc, there is no API support for these. Most of the semantics of scheduling in the RTSJ are defined to be for the priority scheduler and they are carried out under the hood. This was to allow for greater flexibility in an RTSJ implementation that would want to support other schedulers[16]. However, this now means that, in order to expose the underlying

---

[15] The intention of this method is to allow other schedulers to support schedulable objects of a different type to `RealtimeThread` and `AsyncEventHandler`.

[16] Since the semantics for methods like `waitForNextPeriod()` are only defined for the `PriorityScheduler`, other schedulers can support different semantics.

mechanisms, a radical overhaul of the RTSJ scheduling API would be required.

The FMSF framework enables a different approach. The framework protocol is such that it can rely on the priority-based dispatching of the RTSJ to carry out application-defined scheduling policy decisions. This means that changes to the API can be kept at a minimum, because the scheduling mechanism is kept as is and invisible to applications. In order to implement the framework scheduling band operations must be supported. That is, scheduling points must be identified and base scheduler calls must be added. To this end seven new methods are introduced in the `PriorityScheduler` class[17]. They can be seen in following class specification:

```
package javax.realtime;
public class PriorityScheduler extends Scheduler{
  ...
  // constants for "reason" argument
  public static final int WAIT_FOR_NEXT_RELEASE;
  public static final int SLEEP;
  public static final int IO_WAIT;
  public static final int MUTEX_WAIT;
  public static final int SCOPED_MEMORY_ENTRY;
  public static final int YIELD;
  public static final int THREAD_END;
  ...
  // new methods[18]
  protected static final void reschedule(Schedulable sched);
  protected static final void prepareToLock(Schedulable sched,
                        java.lang.Object lock);
  protected static final void rescheduleLock(
                        Schedulable sched);
  protected static final void prepareToUnlock(
                        Schedulable sched);
  protected static final void rescheduleUnlock(
                      Schedulable sched, java.lang.Object lock);
  protected static final void prepareToSuspend(
                        Schedulable sched, int reason);
  protected static final void rescheduleResume(
                        Schedulable sched);
  public static final int getPreLevelsPerBand();
  public static final void setPreLevelsPerBand(int levels);
  public static final void setScheduler(
```

---

[17] Currently the RTSJ (informally) defines that other scheduling mechanisms can be supported. With the scheme proposed in this thesis this can still be achieved if other base schedulers implement their own such methods, tailored to their own dispatching mechanism.

[18] These methods must be implemented as thread-safe.

```
                ApplicationDefinedScheduler appSched, int band);
}
```

The first seven new `PriorityScheduler` methods are, essentially, the *pbsc* and *sbsc* base scheduler calls, as described in Chapter 4. The `prepareTo` methods constitute the *pbsc* part of the scheduling band operation and the `reschedule` methods are the *sbsc* part. It is evident from their names which scheduling band operation they correspond to. Only `prepareToSuspend()` begs some additional clarification. This method is the *pbsc* for all potentially suspending situations, apart from starting a thread, locking a resource and unlocking a resource. The `reason` argument specifies the actual scheduling band operation for which the method is called for, e.g. `WAIT_FOR_NEXT_RELEASE`, `SLEEP`, `MUTEX_WAIT`, `THREAD_END` etc. Therefore, `prepareToSuspend()` covers five of the eight scheduling band operations: *suspension*, *wait*, *yield*, *change* and *end*.

The last three new `public` methods of the `PriorityScheduler` class are available to the application programmer. `getPreLevelsPerBand()` and `setPreLevelsPerBand()` get and set the number of preemption levels available to each scheduling band. Finally, the `setScheduler()` method tells the base scheduler to register the given application-defined scheduler with the given priority level as the band's *low* priority.

6.2.2 Supporting scheduling band operations

The key is identifying all possibly suspending situations in the RTSJ and Java libraries, e.g. the `sleep()` methods in `javax.realtime.RealtimeThread` and `java.lang.Thread` classes. The goal is to pass control to the base scheduler (`PriorityScheduler`) during each of the potentially suspending calls, once before the scheduling point and once after. We can distinguish three possibilities. The first is a potentially blocking library call, such as `RealtimeThread.sleep()`. In essence, the bytecode of such a method contains a particular bytecode instruction (or series of instructions) that can cause thread blocking. All such methods should fall into one of the eight scheduling band operations identified in Chapter 4 and essentially contain the scheduling point of the operation. To complete the scheduling band operation model the implementation must also provide the base scheduler calls before and after the scheduling point. Adding the base scheduler calls entails the addition of a call to `PriorityScheduler.prepareToSuspend(rt,SLEEP)` just before the

potentially blocking instruction (or instructions), and a call to `PriorityScheduler.rescheduleResume(rt)` right after that instruction, where `rt` is the executing real-time thread. The second possibility is a potentially blocking bytecode instruction, such as `monitorenter`. In this case the RTSJ compiler could add the calls to the `PriorityScheduler` at compile time before and after the instruction. Alternatively, the RTSJ virtual machine (RTSJVM) could be modified to perform the appropriate calls, or a suitable `ClassLoader` could perform bytecode re-writing to add the calls. The third case is the when the RTSJVM needs to perform a context switch due to an unforeseen situation; for example, if there is an uncaught exception that terminates the `run()` method of a thread. In this case the runtime environment must be appropriately modified to perform the *end* scheduling band operation by having the thread calling `prepareToSuspend(curThread,THREAD_END)`. For an example, let us consider one of the `RealtimeThread.sleep()` methods again. Conceptually, it can be rewritten as:

```
public static void sleep(HighResolutionTime time)
                    throws java.lang.InterruptedException {
  ...
  PriorityScheduler.prepareToSuspend(
                RealtimeThread.currentRealtimeThread(), SLEEP);
  // code that constitutes the scheduling point e.g. a POSIX sleep()
  PriorityScheduler.rescheduleResume(
                RealtimeThread.currentRealtimeThread());
  ...
}
```

Figures 6.2, 6.3 and 6.4 show how the execution of a scheduling band operation would take place in an RTSJ implementation of the FMSF framework. The first figure is for a potentially blocking method, the second for a potentially blocking bytecode instruction, and the third for a possibly suspending situation arising within the RTSJ virtual machine.

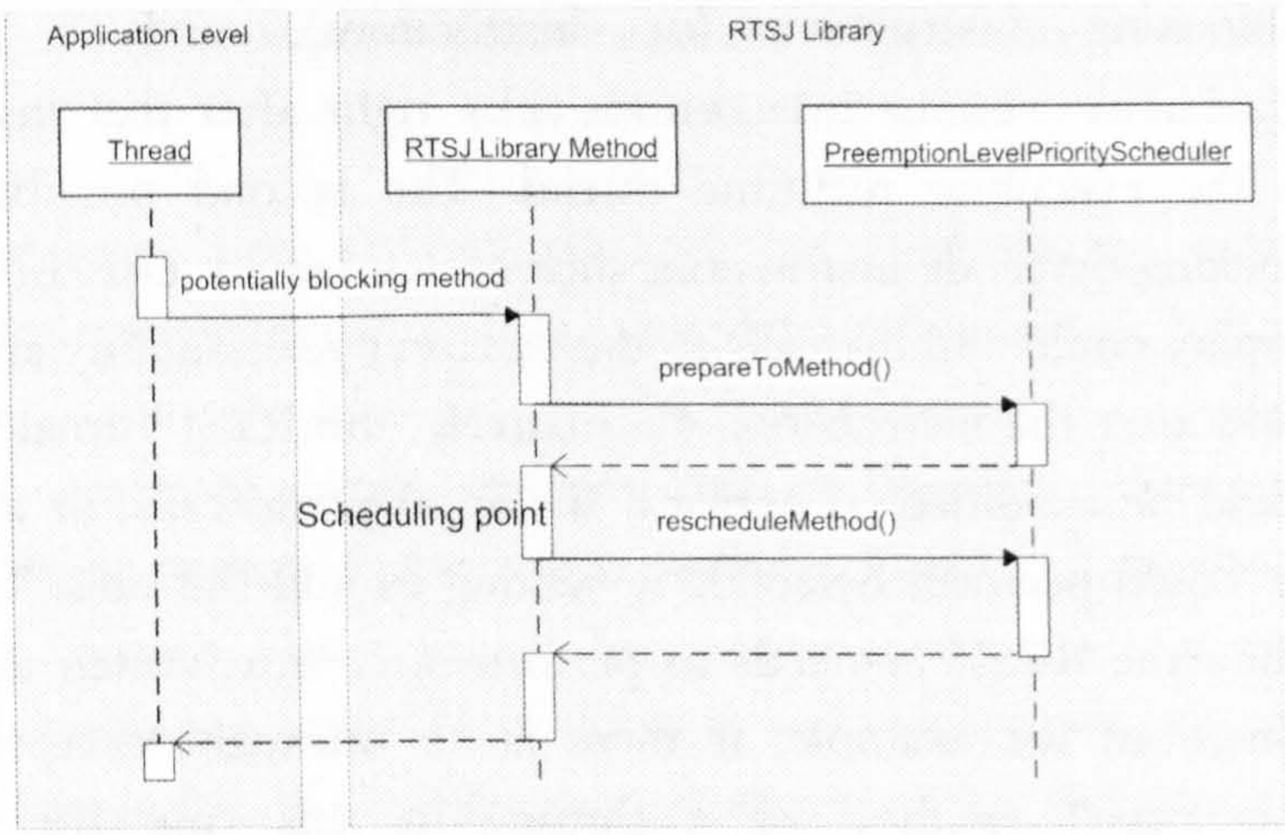


Figure 6.2: Adding base scheduler calls to a potentially blocking library method

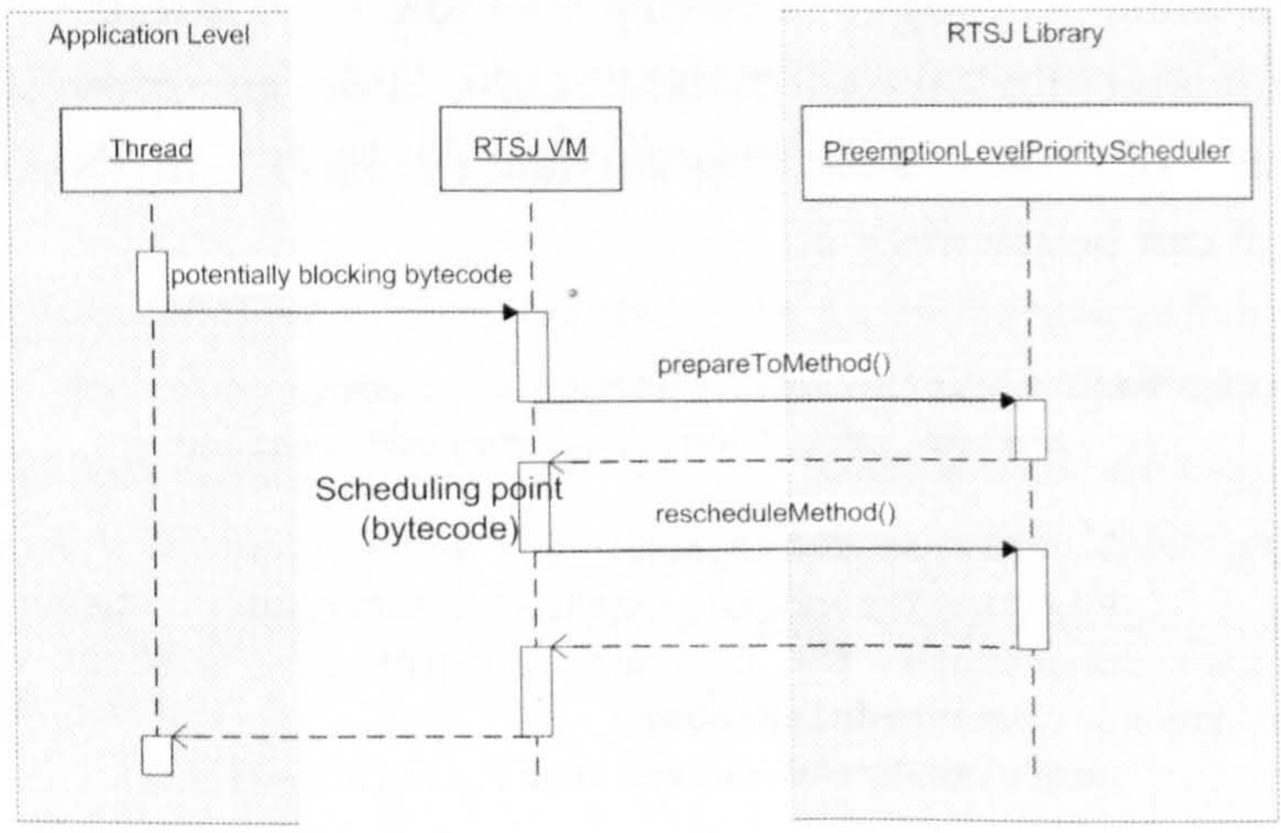


Figure 6.3: Adding base scheduler calls to a potentially suspending bytecode instruction

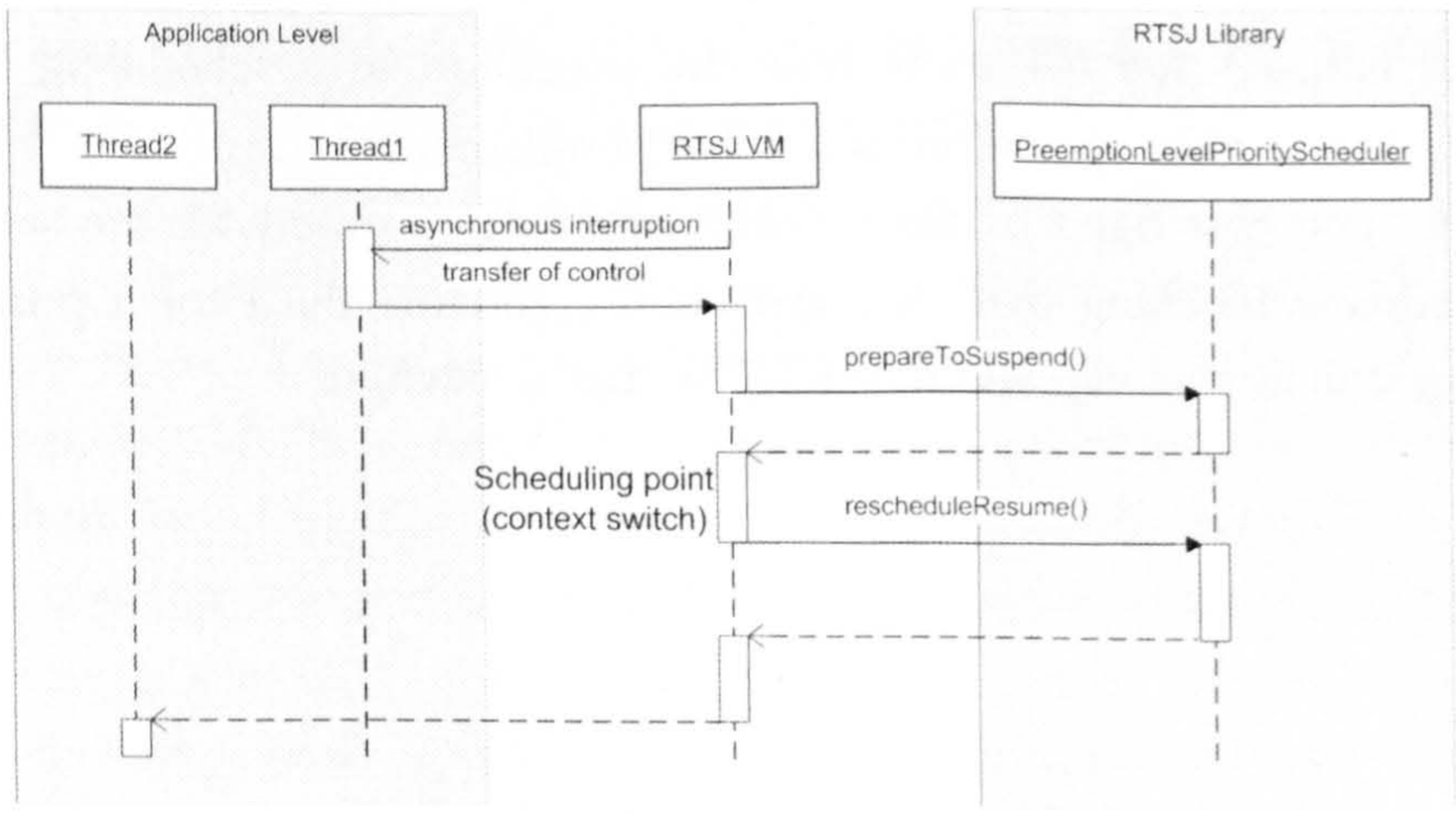


Figure 6.4: Adding base scheduler calls within the virtual machine

6.2.3 The `PreemptionLevelParameters` class

As we have seen in Chapter 4, a key part of the FMSF framework is the use of preemption levels for the sharing of resources. Each thread must, therefore, be associated with its own preemption level. This is supported by extending the `PriorityParameters` class. This way the `SchedulingParameters` object associated with each thread will contain both a priority, which is essential for the dispatching of the thread by the base scheduler, and a preemption level. The new `SchedulingParameters` hierarchy is shown below, followed by the description of the `PreemptionLevelParameters` class.

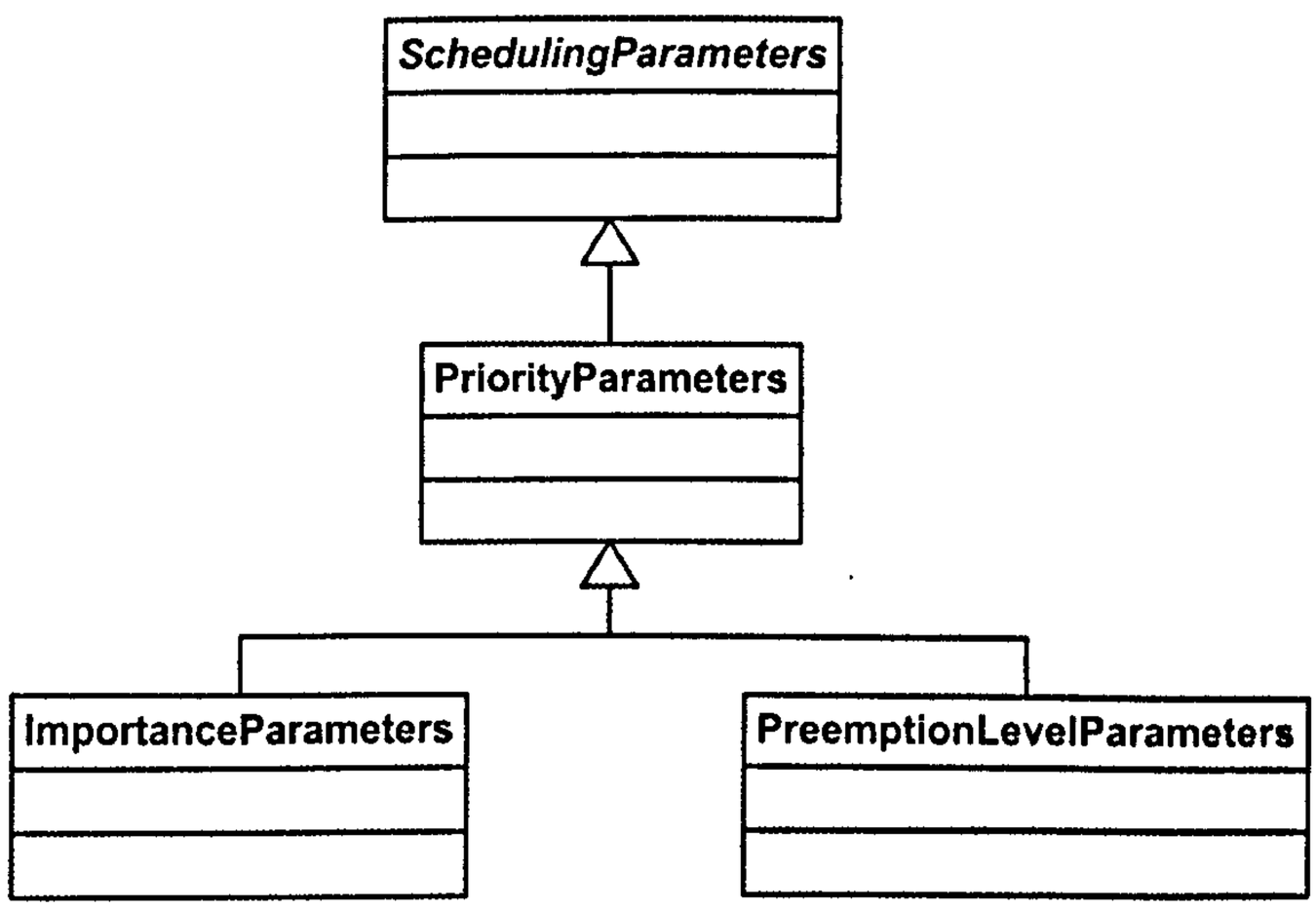


Figure 6.5: The `SchedulingParameters` class hierarchy

```
package javax.realtime;
public class PreemptionLevelParameters extends
                                              PriorityParameters {
  public PreemptionLevelParameters(
                         int band, int relativePreemptionLevel);
  //methods
  public int getBand();
  public int getPreemptionLevel();
  public int getAbsPreemptionLevel();
  public void setBand(int newBand);
  public void setPreemptionLevel(int relativePL);
  protected void setAbsPreemptionLevel(int absolutePL);
  public java.lang.String toString();
}
```

An object of the class is initialised with two values, the band to which the `PreemptionLevelParameters` object will correspond (i.e. the *low* priority of the band) and a relative preemption level to be used within that band. These

values can be retrieved and changed with the provided `get` and `set` methods. A third field is also part of a `PreemptionLevelParameters` object, the absolute preemption level, corresponding to the given relative preemption level. This field's value is set by the base scheduler using the available `set` method.

6.2.4 Application-defined schedulers

To allow application-defined schedulers, a new subclass of `Scheduler` is introduced. The position of the `ApplicationDefinedScheduler` class in the scheduler hierarchy is shown in Figure 6.6.

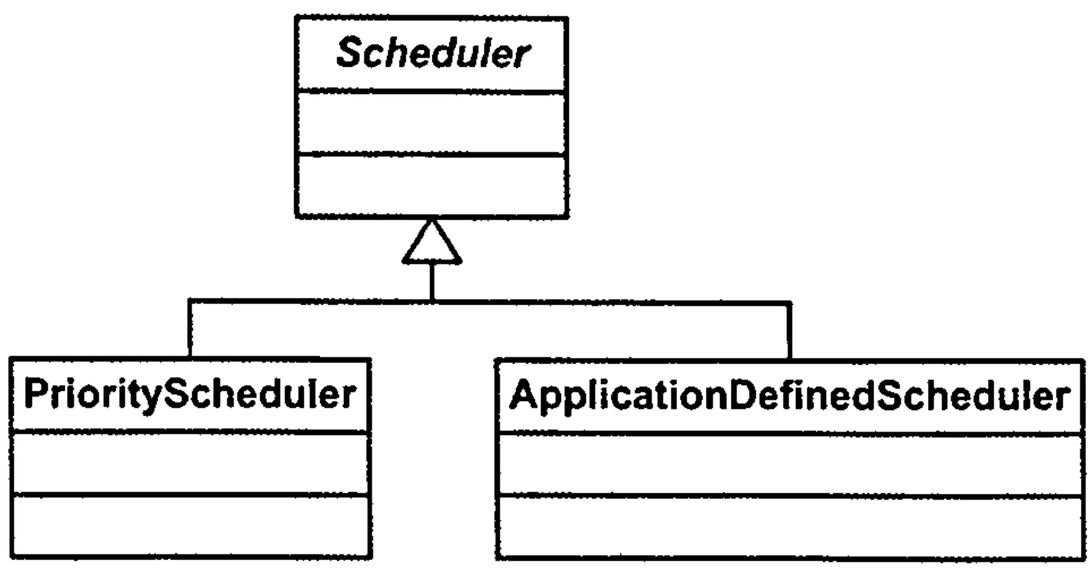


Figure 6.6: The `ApplicationDefinedScheduler` class diagram

The class definition is the following:

```
package javax.realtime;
public abstract class ApplicationDefinedScheduler extends
                                                    Scheduler {
  public ApplicationDefinedScheduler(int band,
                          ProcessingGroupParameters capacity);
  // abstract methods
  protected abstract void released(Schedulable sched);
  protected abstract void lockedObject(Schedulable sched);
  protected abstract void unlockedObject(Schedulable sched);
  protected abstract void suspended(Schedulable sched,
                                    int reason);
  protected abstract void blocked(Schedulable sched);
  protected abstract void resumed(Schedulable sched);
  protected abstract Schedulable[] getMostEligible();
  protected abstract Schedulable[] isEligible(
                                     Schedulable sched);
  protected abstract boolean setScheduler(Schedulable sched);

  // instance methods
  public int getBand();
}
```

As we can see in the constructor, each newly constructed application-defined scheduler is assigned a `ProcessingGroupParameters` object. This object represents the CPU percentage available to all the scheduler's threads. This effectively allows the threads to be scheduled within a server and will be further explained in Section 6.2.7.

Through its abstract methods this class sets the API that all user-defined schedulers must provide in order to be compatible with the FMSF. In other words, an application-defined scheduler must inherit from this class and implement all abstract methods, which form the one-way API between it and the `PriorityScheduler`. In this scheme the application-defined scheduler is a passive entity. Only the base scheduler can pass messages to other schedulers. For most of the methods the name is indicative of the method's purpose:

- `released()` informs the application-defined scheduler that a new thread in the scheduler's band has been started
- `lockedObject()` tells the scheduler that one of its threads has locked an object (i.e. entered a `synchronized` region)
- `unlockedObject()` informs the scheduler that one of its threads has released an object lock (i.e. exited a `synchronized` region)
- `blocked()` informs of a thread being blocked while trying to lock a resource
- `suspended()` informs the scheduler that a thread has been suspended (in reality, this method is called right before the thread is actually suspended)
- `resumed()` informs that a thread has resumed after a suspension
- `getMostEligible()` asks from the application scheduler to return its currently most eligible thread; the scheduler must return a two-cell array (`Schedulable[2]`) that contains the most eligible thread in the first cell (cell '0') and a preempted thread, if any, in the second cell (cell '1')
- `isEligible()` asks the application-defined scheduler whether the current thread is eligible to run; the scheduler again returns a two-cell array, as in `getMostEligible()`.
- `setScheduler()` asks the scheduler to accept a thread to be scheduled by it, while also associating the scheduler's `ProcessingGroup Parameters` object with the thread.

- lastly, the `getBand()` instance method returns the *low* priority of `this` band.

As we have already seen in the previous section, new application-defined schedulers can be added to a system by registering them using the `PriorityScheduler.setScheduler()` method. This way multiple user-defined schedulers can coexist in the system, occupying non-overlapping bands in the RTSJ priority range. Hence, the proposal introduces two-level scheduling to RTSJ. The first level is the `PriorityScheduler`, the second level is user-defined.

6.2.5 Execution eligibility inversions

Priority inversion can occur in the RTSJ whenever a schedulable object is blocked waiting to enter a `synchronized` region or method. In order to limit the length of time of that blocking, the RTSJ requires that the priority scheduler maintain all queues used by the real-time virtual machine in priority order. So, for example, the queue of schedulable objects waiting for an object monitor, as a result of a `synchronized` method call or the execution of a `synchronized` statement, must be priority ordered. Where there is more than one schedulable object in the queue at the same priority, the order between them is defined to be first-in-first-out (FIFO). Similarly, the queues resulting from calls to the `wait()` methods in the `Object` class should be priority FIFO ordered.

The RTSJ provides facilities for the programmer to specify the use of different priority inversion control algorithms. By default, the RTSJ requires priority inheritance to occur whenever a schedulable object is blocked waiting for a resource. However, an implementation of the FMSF needs to enforce its own eligibility inversion avoidance protocol, which, as seen in Section 4.5, is based on the Basic Preemption-Ceiling Protocol (BPreCP). Conveniently, the RTSJ provides the ability to change the default priority inversion control algorithm for any and all objects in the system via the `MonitorControl` class hierarchy. At the root of this hierarchy is the following `abstract` class:

```
package javax.realtime;
public abstract class MonitorControl {
  // constructors
  protected MonitorControl();

  // methods
```

```
  public static MonitorControl getMonitorControl();
  public static MonitorControl getMonitorControl(
                               java.lang.Object obj);
  public static MonitorControl setMonitorControl(
                               MonitorControl policy);
  public static MonitorControl setMonitorControl(
            java.lang.Object obj, MonitorControl policy);
}
```

The four static methods allow the getting/setting of the default policy and the getting/setting for an individual object (the methods return the old policy). The RTSJ defines two policies, subclasses of `MonitorControl`: `PriorityInheritance` (default policy) and `PriorityCeilingEmulation`. All we have to do to extend the RTSJ and support preemption levels is to introduce the following new class:

```
package javax.realtime;
public class BPreCPResourcePolicy extends MonitorControl {
  private BPreCPResourcePolicy(int band, int ceiling);
  // methods
  public int getSchedulingBand();
  public int getCeiling();
  public static BPreCPResourcePolicy getMaxPreemptionLevel();
  public static BPreCPResourcePolicy instance(int band,
                                              int ceiling);
}
```

The `MonitorControl` class hierarchy thus becomes the following:

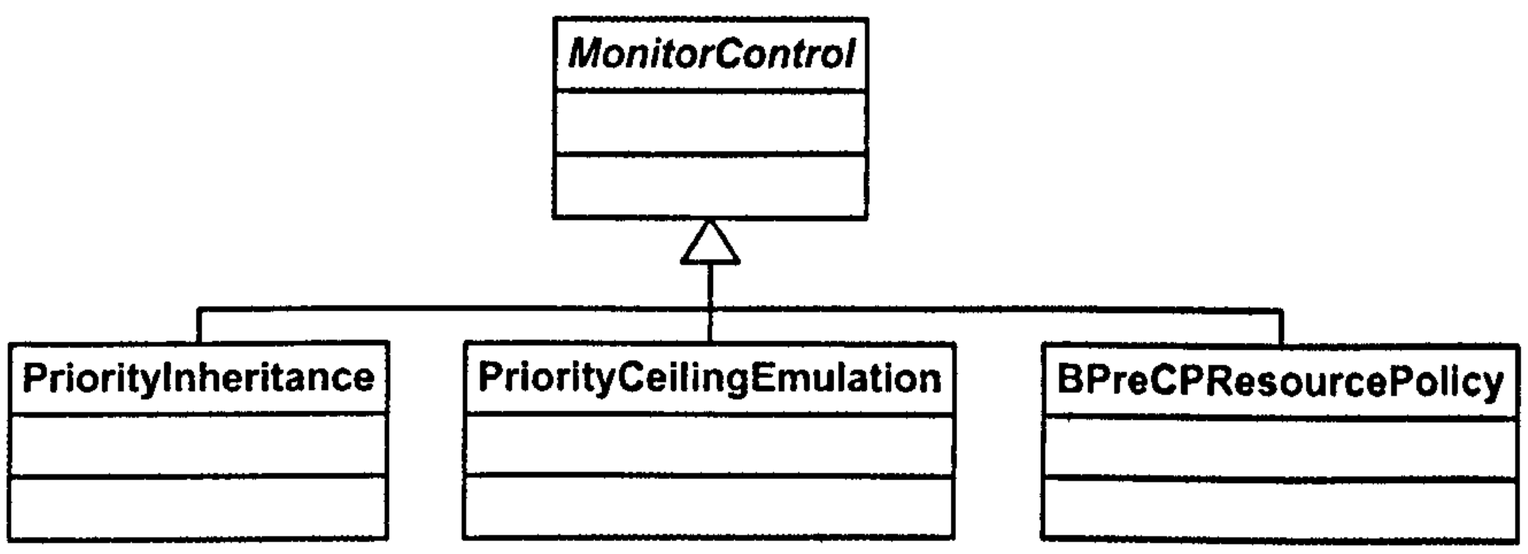


Figure 6.7: The `MonitorControl` class hierarchy

At this point it is worth repeating what was said in Section 4.5.5, that implementation of the eligibility inversion control policy is done at the middleware layer and is transparent to the OS. The OS can carry out its own priority inversion control algorithm. Also, it is important to note that every resource accessed by threads (one or more) running under an application-

defined scheduler should be governed by a `MonitorControl` object of type `BPreCPResourcePolicy`.

6.2.6 `EDFScheduler`: an application-defined scheduler

As an example and proof of concept for our user-defined scheduling scheme, we can implement an EDF scheduler.

```
public class EDFScheduler
        extends ApplicationDefinedScheduler {
    public EDFScheduler(int band,
                        ProcessingGroupParameters capacity);
    ...
}
```

The class implements all abstract `Scheduler` and `ApplicationDefinedScheduler` methods. In addition, it has three internal structures for the enforcement of the EDF policy within a scheduling band, as specified in Section 5.1. These are: i) the EDF queue of band tasks currently residing within the band, ii) the locking list of band tasks currently locking within the band, and iii) the LIFO medium_lock locking list for keeping track of eligibility inheritance when lower tasks lock in the band. Figure 6.8 shows the position of the `EDFScheduler` class in the `Scheduler` hierarchy.

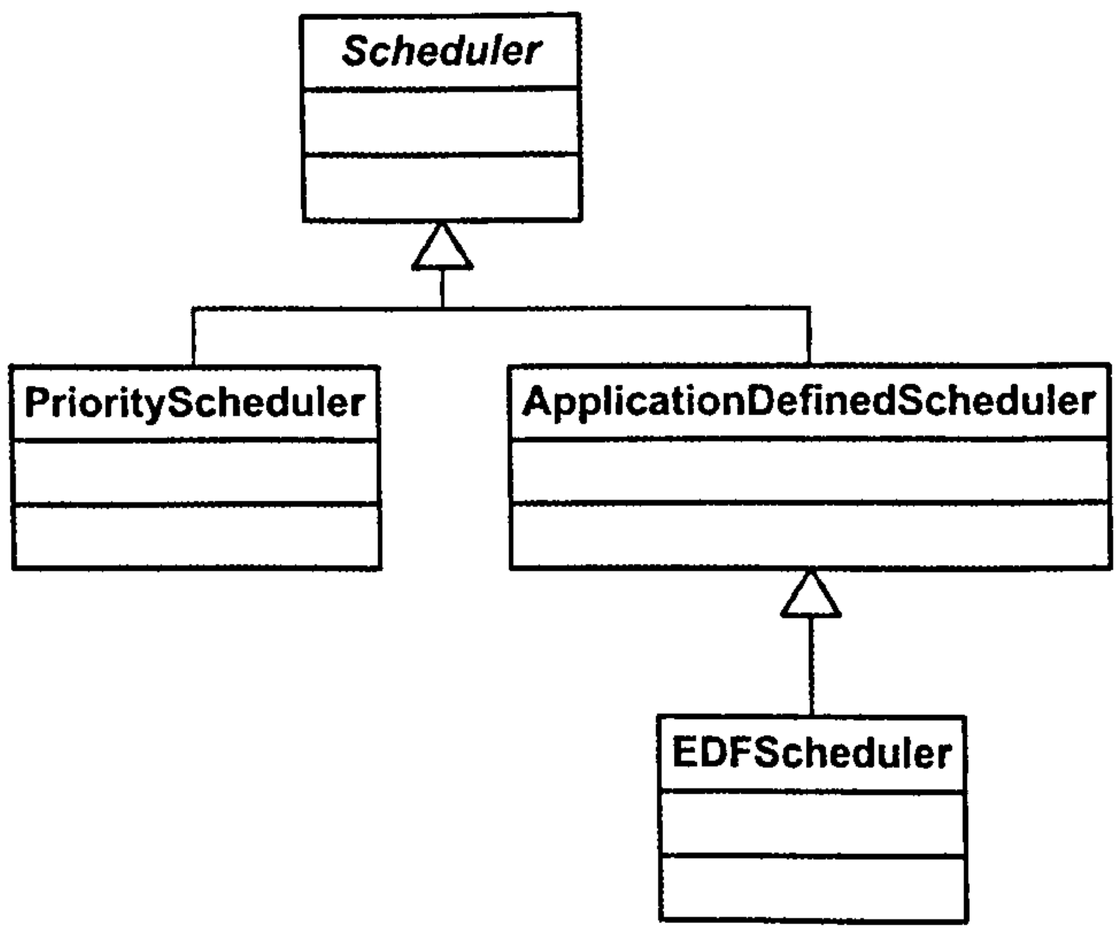


Figure 6.8: `EDFScheduler` in the `Scheduler` class hierarchy

Construction of the `EDFScheduler` is fairly straightforward, given the description of Section 5.1. However, there is one vital element missing: the RTSJ API does not include the notion of an absolute thread deadline. To

rectify this situation we introduce a subclass of `PeriodicParameters`, `EDFPeriodicParameters`. The only new element this class introduces is methods that get and set an absolute deadline. This is the absolute deadline of a thread associated with an instance of this class. The class description is given below.

```
import javax.realtime.*;
public class EDFPeriodicParameters extends PeriodicParameters {
    public     EDFPeriodicParameters(HighResolutionTime     start,
RelativeTime period, RelativeTime cost, RelativeTime deadline,
AsyncEventHandler overrunHandler, AsyncEventHandler missHandler);

    protected void setAbsDeadline();
    protected void setAbsDeadline(RelativeTime rt);
    protected void setAbsDeadline(AbsoluteTime at);
    public AbsoluteTime getAbsDeadline();
}
```

The `setAbsDeadline()` method sets the absolute deadline to an absolute time equal to the current time plus the `deadline` parameter passed to the constructor. The `setAbsDeadline(RelativeTime rt)` method sets the absolute deadline to `rt` amount of time past the previous absolute deadline, or past the current time, if the previous absolute deadline is `null`. The `setAbsDeadline(AbsoluteTime at)` just sets the absolute deadline to `at`. The `getAbsDeadline()` method returns the absolute deadline of the thread associated with `this`. The class diagram is shown in Figure 6.9.

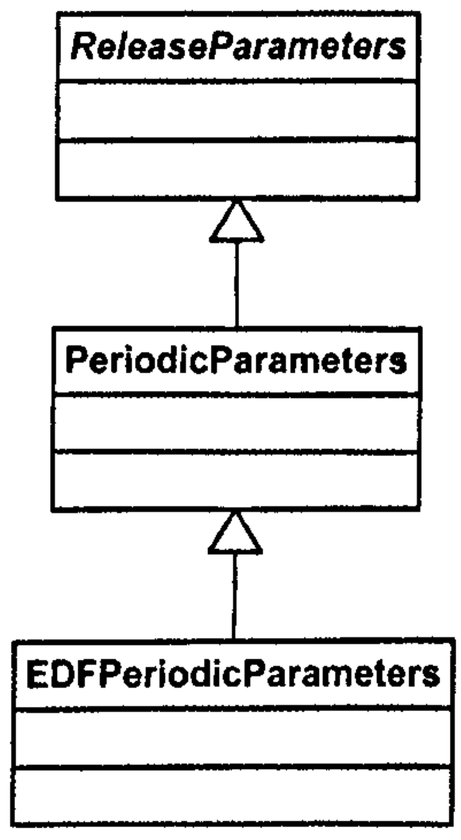


Figure 6.9: `EDFPeriodicParameters` in the `ReleaseParameters` class hierarchy

This class is, of course, meant to be used for periodic threads. In order to use aperiodic or sporadic threads with the `EDFScheduler` we can simply

construct EDFAperiodicParameters and EDFSporadicParameters, inheriting from AperiodicParameters and SporadicParameters, respectively. Alternatively, we could construct EDFNonPeriodicParameters, inheriting from SporadicParameters, which could be used both as an AperiodicParameters and as a SporadicParameters class.

6.2.7 Impact on feasibility analysis

As well as providing a framework for schedulers, the RTSJ includes a framework for the supporting on-line feasibility analysis. However, the default feasibility analysis for the priority scheduler is very crude (it simply assumes an adequately fast machine to handle the periodic and sporadic load). The proposal here allocates all application-defined schedulers a CPU budget and replenishment period using a constrained version of the RTSJ processing group parameters mechanism. A ProcessingGroupParameters object defines a group period, cost, start time, deadline, and relevant overrun and miss handlers. For threads associated with a ProcessingGroupParameters object the system guarantees that they will not be collectively given more time per group period than indicated by the group cost. Of course, this mechanism requires support from the operating system. As explained in [Belliardi et al. 2006], an instance of ProcessingGroupParameters is logically associated with a virtual server. The server can only logically execute when i) it has not consumed more execution time in its current release than the cost (budget) parameter, ii) one of its associated schedulable objects (e.g. threads) is executable and is the most eligible of the executable schedulable objects. If the server is logically executable, the associated schedulable object is executed. When the cost has been consumed, any overrun handler is released, and the server is not eligible for logical execution until its next period is due. At this point, its allocated cost (budget) is replenished. If the server is logically executing when its deadline expires, any associated miss handler is released. The deadline and cost parameters of all the associated schedulable objects have the same impact as they would if the objects were not bound to a processing group.

This way threads within a scheduling band can be treated as if they are being served by a deferrable server [Strosnider et al. 1995]. Hence, if the priority scheduler is supporting true feasibility analysis, then this is not undermined by the proposed approach. Within a band, the application-defined scheduler can only assume that it gets its full budget each period. Hence, it can only give independent partial guarantees. To give full guarantees needs a global

server-based analysis (see [Davis and Burns 2005]). To give full independent guarantees would require the priority scheduler to guarantee the capacity specified in the processing group parameters, which would be a change to the RTSJ processing group semantics.

6.3 A simulated implementation of FMSF

The previous sections explained how a full-fledged implementation of the framework can be accomplished. However, such an implementation would be outside the scope of our work. The purpose of an implementation, within the scope of this thesis, is not to prove that the framework can be implemented. This was shown by modelling the system and showing that no race conditions exist. Nor is it about providing the most efficient or optimised implementation. Rather, the aim is to acquire an estimate on the order of magnitude of the overhead that the framework will introduce to the middleware. Therefore, since acquiring execution time estimates, and not determining the exact cost of the API calls, is our concern, we have opted for a simulated implementation of the framework, which avoids cumbersome implementation details, while still providing us with timing results equivalent to those of a full implementation. This has been done by coding all framework functionality as a set of user/application level classes. The scheduling band operations' base scheduler calls are "manually" called from within each thread. This can be seen in Figure 6.10.

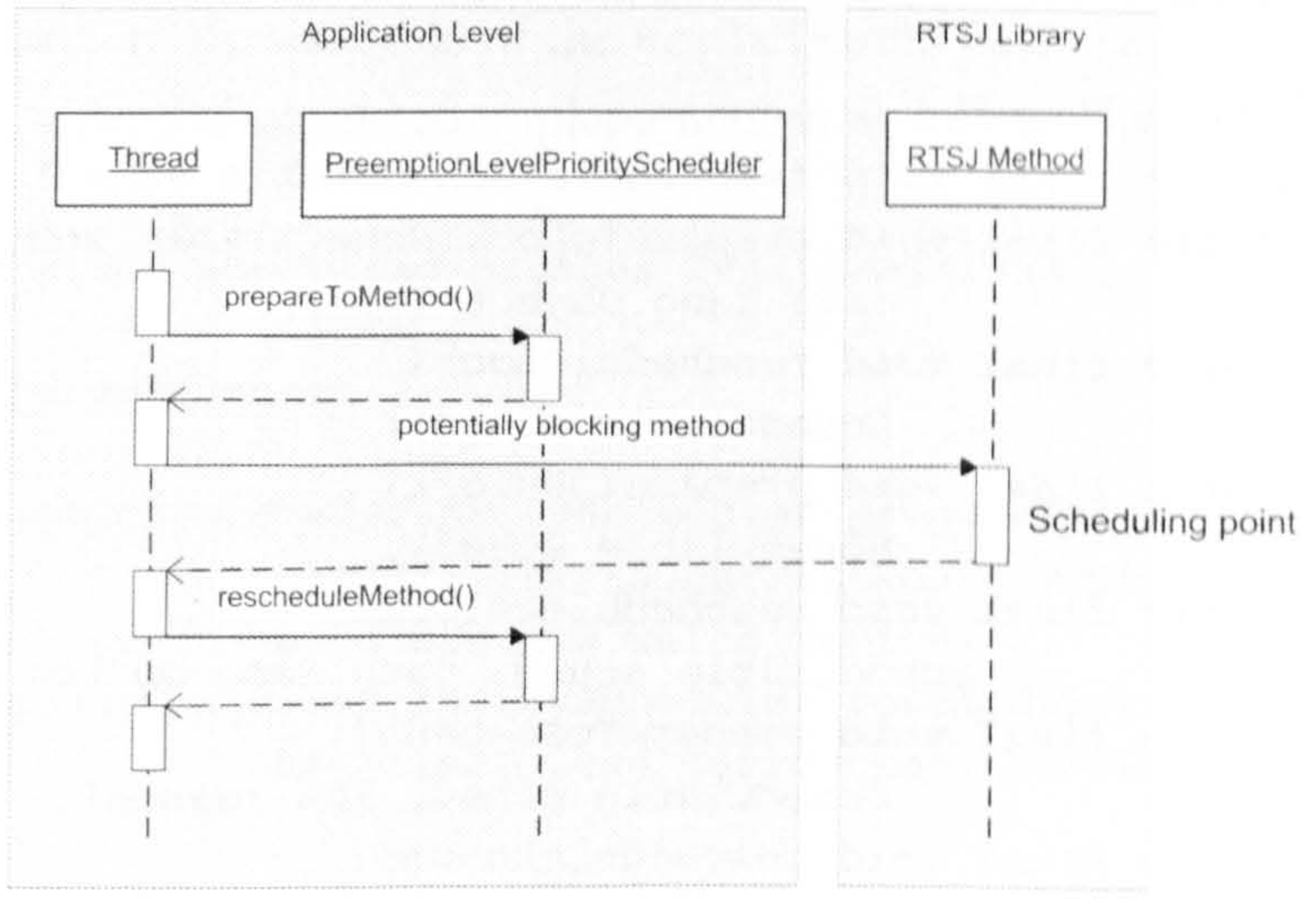


Figure 6.10: Execution of a scheduling band operation under a simulated framework implementation

By comparing this with Figures 6.2 and 6.3 in Section 6.2.2 we can see that the only difference between the two approaches, as far as the base scheduler calls are concerned, is that in a full implementation the calls are executed as part of the RTSJ library, whereas in the simulated approach they are executed as application code. This works the same for potentially blocking methods as well as bytecodes. However, in both cases the virtual machine needs to just-in-time compile this code, which means that the execution cost of the base scheduler calls in both cases is of the same order of magnitude. From that we draw the conclusion that our measurements of the simulated framework will give a faithful picture of the execution costs under a full implementation.

6.3.1 The `PreemptionLevelPriorityScheduler` class

The main difference between a full implementation and the simulated version is the implementation of the base scheduler calls. The purpose of the simulated version is to avoid reprogramming the RTSJ itself. Therefore, with this approach the base scheduler calls cannot be added to `PriorityScheduler`. Instead, a subclass of `PriorityScheduler` is introduced, `PreemptionLevelPriorityScheduler`. This class simply contains all the methods added to `PriorityScheduler` in Section 6.2.1. Its description is given below.

```
public class PreemptionLevelPriorityScheduler extends Scheduler
{
    ...
    // constants for "reason" argument same as in Section 6.2.1
    ...
    // new methods[19]
    public static final void reschedule(Schedulable sched);
    public static final void prepareToLock(Schedulable sched,
                           java.lang.Object lock);
    public static final void rescheduleLock(
                           Schedulable sched);
    public static final void prepareToUnlock(
                           Schedulable sched);
    public static final void rescheduleUnlock(
                      Schedulable sched, java.lang.Object lock);
    public static final void prepareToSuspend(
                           Schedulable sched, int reason);
    public static final void rescheduleResume(
                           Schedulable sched);
    public static final int getPreLevelsPerBand();
```

[19] These methods must be implemented as thread-safe.

```
  public static final void setPreLevelsPerBand(int levels);
  public static final void setScheduler(
          ApplicationDefinedScheduler appSched, int band);
}
```

The only notable difference between the two classes is that the base scheduler calls in `PreemptionLevelPriorityScheduler` are `public` instead of `protected`. The reason is that in the simulated version these methods are meant to be called from within application code, i.e. from different packages, while in a full implementation they are called by other RTSJ library methods. With the addition of this class the `Scheduler` hierarchy has now become the following:

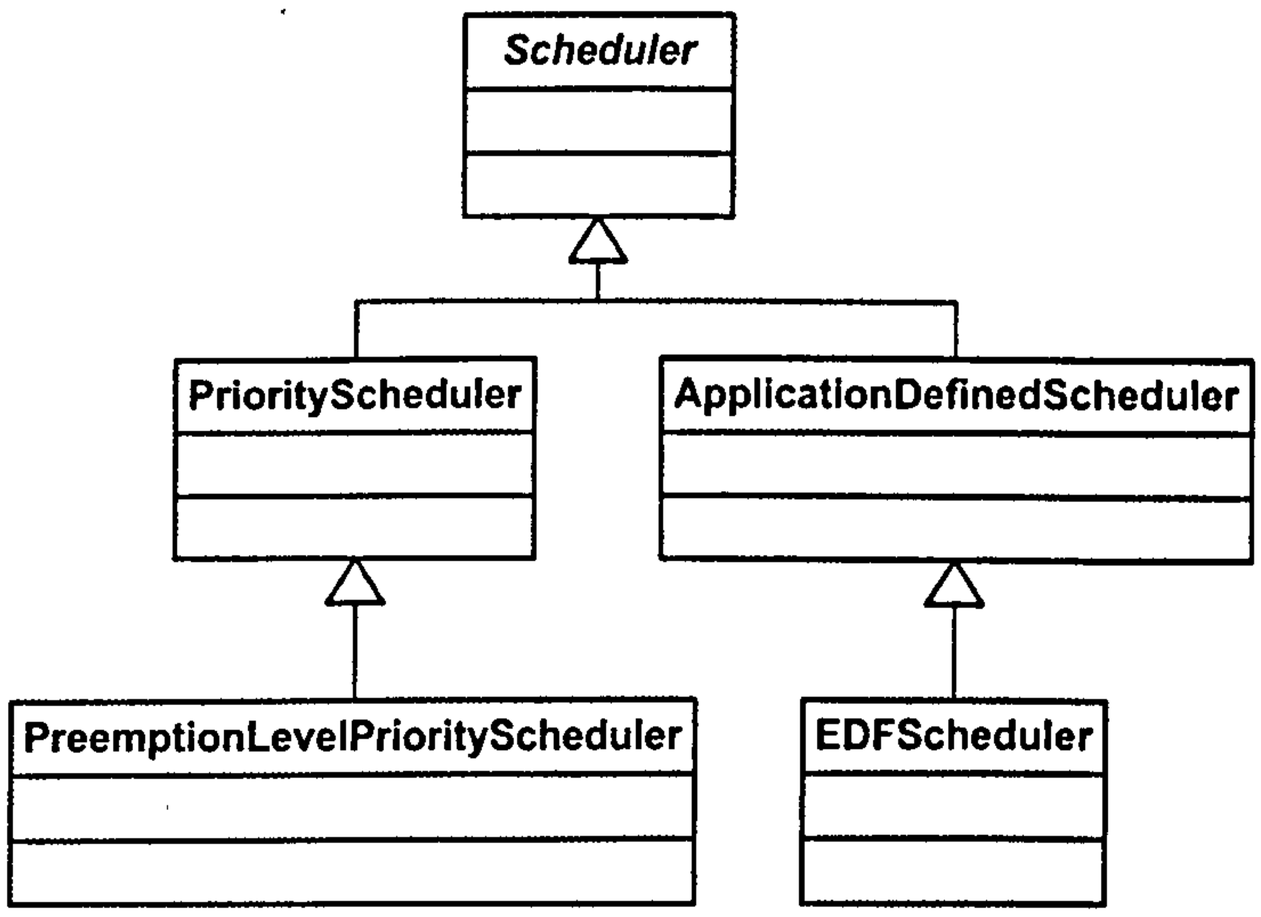


Figure 6.11: `PreemptionLevelPriorityScheduler` in the `Scheduler` class hierarchy

Going back to the example of the `RealtimeThread.sleep()` method we can rewrite it for the simulated implementation as follows:

```
public class AppThread extends RealtimeThread {
  ...
  public void run() {
    ...
    PreemptionLevelPriorityScheduler.prepareToSuspend(
                RealtimeThread.currentRealtimeThread(), SLEEP);
    RealtimeThread.sleep(new RelativeTime());
    PreemptionLevelPriorityScheduler.rescheduleResume(
                RealtimeThread.currentRealtimeThread());
    ...
  }
}
```

6.3.2 Changes to the `BPreCPResourcePolicy` class

A second modification found in the simulated version is the addition of two methods in the `BPreCPResourcePolicy` class. The new class is shown below.

```
package javax.realtime;
public class BPreCPResourcePolicy extends MonitorControl {
  private BPreCPResourcePolicy(int band, int ceiling);
  ...
  // additional methods
  public static BPreCPResourcePolicy getBPreCP(
                                          java.lang.Object obj);
  public static BPreCPResourcePolicy setBPreCP(
          java.lang.Object obj, BPreCPResourcePolicy bprecp);
}
```

The new methods, `getBPreCP()` and `setBPreCP()`, play the role of the `getMonitorControl(java.lang.Object obj)` and `setMonitorControl (java.lang.Object obj, MonitorControl policy)` in the `MonitorControl` class. The need to add them again stems from the simulated nature of the implementation. Even though the `setMonitorControl()` method could be called to bind a `BPreCPResourcePolicy` object to a shared object, this would not achieve the desired result. The RTSJ virtual machine applies only fixed-priority dispatching. Therefore, without modifications to the RTSJ, it would be impossible for the virtual machine to know how to handle a `monitorenter` bytecode instruction on the shared object in order to enforce the FMSF resource sharing protocol.

In the simulated FMSF implementation the application calls `setBPreCP(obj,bprecp)`, which saves the pair (`obj`,`bprecp`) and returns the previous `BPreCPResourcePolicy` for the object, if any. Whenever `obj` is used as the monitor in a `synchronized` statement, the `PreemptionLevelPriorityScheduler` calls `getBPreCP(obj)` on the object to retrieve its `BPreCPResourcePolicy` and, thus, be able to calculate its ceiling. This can be seen in the following code fragment.

```
public class AppThread extends RealtimeThread {
  ...
  public void run() {
    ...
    Object obj = new Object();
    BPreCPResourcePolicy bprecp= BPreCPResourcePolicy.instance(
                                                        3,5);
```

```
BPreCPResourcePolicy.setBPreCP(obj, bprecp);
...
PreemptionLevelPriorityScheduler.prepareToLock(
            RealtimeThread.currentRealtimeThread(), obj);
synchronized(obj) {
  PreemptionLevelPriorityScheduler.rescheduleLock(
            RealtimeThread.currentRealtimeThread());
  ...  // critical region
  PreemptionLevelPriorityScheduler.prepareToUnlock(
            RealtimeThread.currentRealtimeThread());
}
    PreemptionLevelPriorityScheduler.rescheduleUnlock(
                RealtimeThread.currentRealtimeThread(), obj);
    ...
  }
}
```

`BPreCPResourcePolicy.getBPreCP(obj)` is called from within `prepareToLock()` in order to calculate the object's ceiling.

6.4 Measuring the FMSF overhead

Based on the simulated implementation of the framework, this section presents tests that aim at providing an estimate of the execution time of each base scheduler call. The base scheduler calls are the only part of a scheduling operation that is introduced by the framework, therefore their execution time is essentially the overhead that the framework is introducing to the existing RTSJ scheduling framework. The tests where carried out on an AMD Athlon 700MHz machine, with 128MB RAM memory, running Red Hat Linux 8.0 with the `pthreadrt` library and using RTSJ 1.0.2-3 (RI version 1.3). No other activity was executing on the machine, therefore it is assumed that no preemption of the test applications took place. The following sections present the results as raw data, while Section 6.4.5 provides the overall assessment of these results. The test code is not included in this chapter in order to make reading easier. Instead, it is included as Appendix A.

6.4.1 Caveat

In the tests that follow, all measurements have been made using the `Clock.getRealtimeClock().getTime(AbsoluteTime dest)` RTSJ method to get the current time before and after the code to be measured. By subtracting the time before the code from the time after we get our

measurement. However, this way we also include in the measurement the execution time of `getTime()`. This can be seen in Figure 6.12.

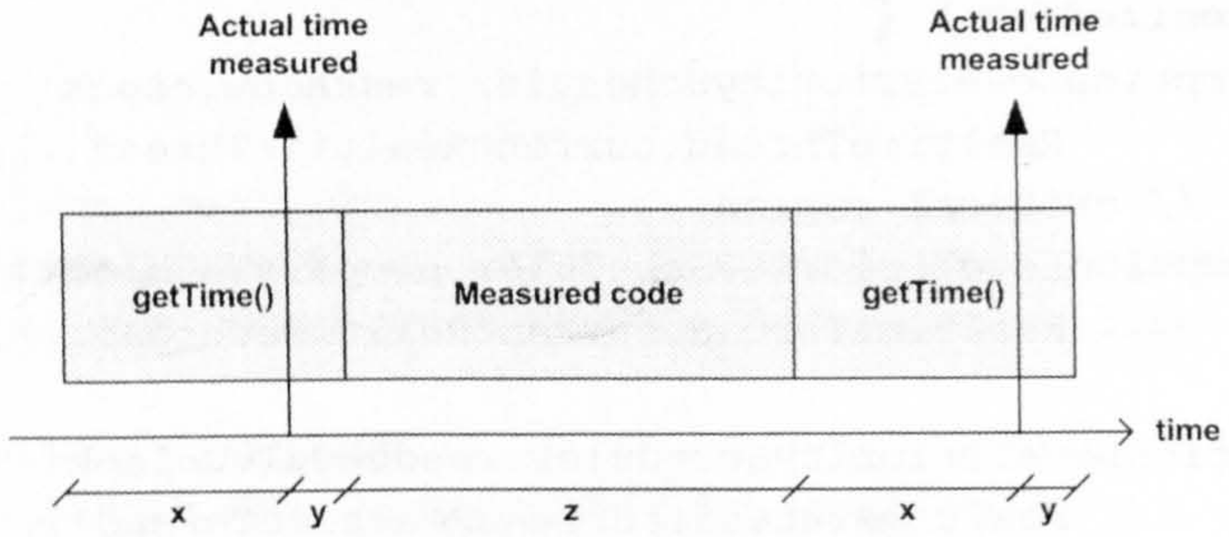


Figure 6.12: Measuring execution time with `getTime()`

The actual time measured is $y+z+x$. However, $x+y$ is the execution time of `getTime()`. Therefore, for every measurement the execution time of `getTime()` is added once. Taking advantage of this we can now measure the execution time of `getTime()` using `getTime()`. In this case z becomes $x+y$, therefore the whole measurement is $2(x+y)$. Dividing it by 2 should give us the cost of `getTime()`. This yields a cost $C_{gT}=2\mu s$.

To get a more precise estimate one needs to subtract this overhead from any measurement. However, this subtraction is not performed in the following tests, since the amount is too small to make a significant difference, and also because it is easy for the reader to calculate.

6.4.2 Single-thread test

The first test is comprised of a single EDF thread, i.e. a thread running under an EDF application-defined scheduler with a *low* priority set at `PriorityScheduler.getMinPriority()`. The thread executes all the `PreemptionLevelPriorityScheduler` base scheduler calls a thousand times each. Basically, the thread's `run()` method contains a `for()` loop within which it acquires timing results for each base scheduler call and performs some calculations. After exiting the loop the thread performs some final calculations, prints the results and terminates.

This test measures the execution times of `reschedule()`, `prepareToLock()`, `rescheduleLock()`, `prepareToUnlock()`, `rescheduleUnlock()`, `prepareToSuspend(SLEEP)`, `RealtimeThread.sleep()`, `rescheduleResume()` and `prepareToSuspend(WFNP)`. The `prepareToSuspend(SLEEP)` call was selected as representative of all other

prepareToSuspend calls, except prepareToSuspend(WFNP). This latter one is also representative of prepareToSuspend(THREAD_END), as the two execute the same code. The test calculates the actual amount of time spent in each call. The final printed results are the mean execution times for one thousand iterations. The results of this test can be seen in Figure 6.13 below.

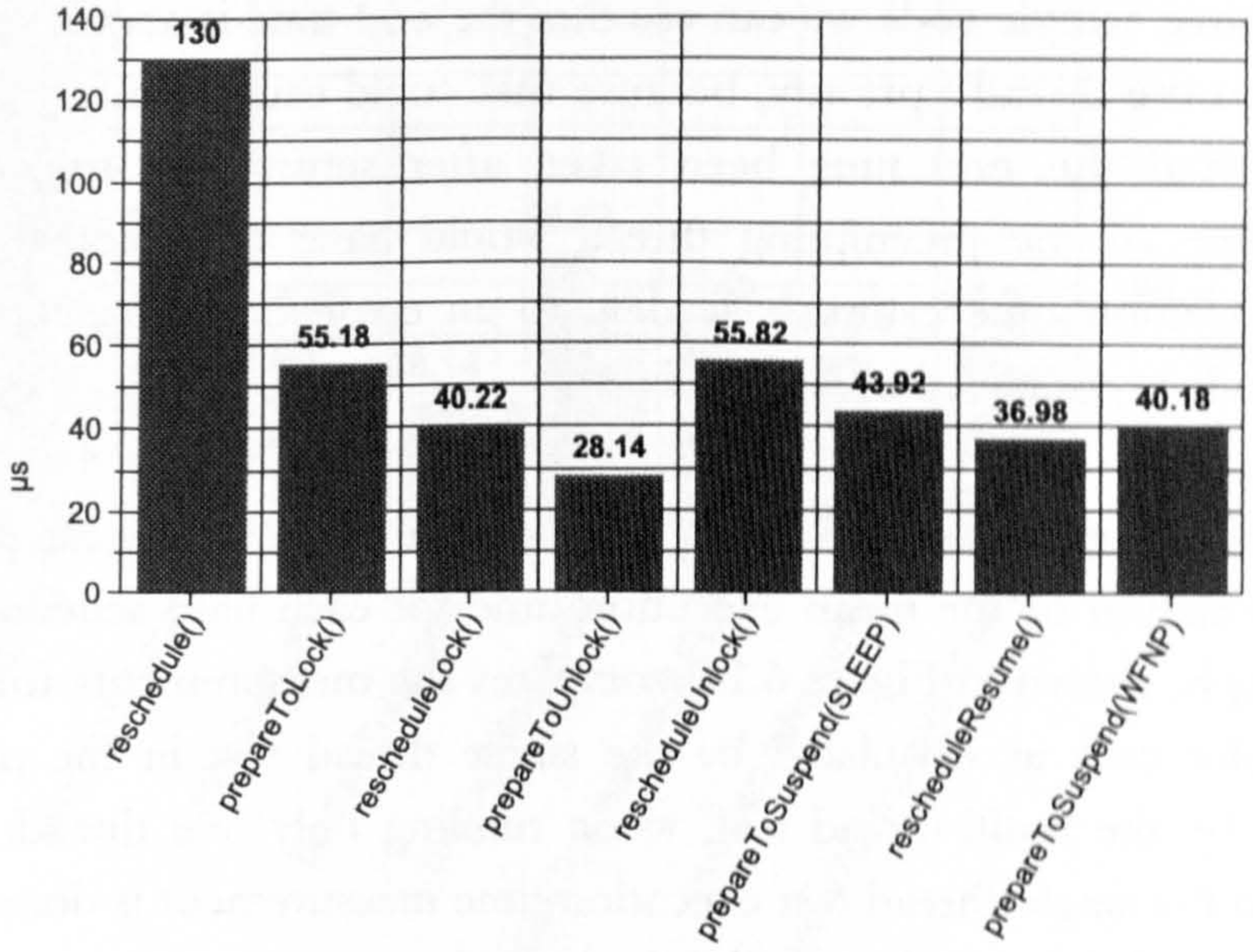


Figure 6.13: Base scheduler call execution times for one test thread

6.4.3 Multi-thread tests

The second test is actually a series of tests in which a number of EDF threads run concurrently, under the same EDF scheduler, each going through all the steps described in the previous section. The objective is to see how the execution time of each base scheduler call is affected with the increase in the number of concurrent threads. However, due to the fact that multiple threads are now running, the execution time has to be calculated within the base scheduler calls themselves. Therefore, each of the methods in the PreemptionLevelPriorityScheduler has been changed to accept one more parameter of type RelativeTime, which is returned by the method to contain the method's execution time. A method calculates its execution time according to the following model, where start, end and plp are static variables of the PreemptionLevelPriorityScheduler class.

```
  public synchronized static RelativeTime someMethod(Schedulable sched,
... , RelativeTime rTime) {

    start = Clock.getRealtimeClock().getTime(start);
    plp = (PreemptionLevelParameters)sched.getSchedulingParameters();
```

```
    // the method body

    end = Clock.getRealtimeClock().getTime(end);
    plp.setPriority(somePriority);
    rTime = end.subtract(start,rTime);
    return rTime;
}
```

In the above sample code we can see that the end time is acquired before any change in the thread's priority, because that could cause the thread to be preempted. Had the end time been taken after setting the priority, the execution time of the preempting thread would have been added to the method's execution time estimate, leading to an erroneous calculation. The actual test code is contained in Appendix A.

In the multi-thread case the timing results taken from all threads produce the final estimation of the mean execution time for each base scheduler call. The first graph, shown in Figure 6.14, compares the measurements for all the base scheduler calls as calculated by the single thread test in the previous section and by the multi-thread test, when running only one thread. As we have seen, in the single thread test execution time measurement is done within the test thread, while in the multi-thread version measurements are taken within the base scheduler calls. This is shown in the graph as "timing by thread" and "timing by scheduler". The timing difference between the two tests is due to two reasons. Firstly, in the "timing by thread" test there is the extra overhead of calling the base scheduler method, which does not exist when performing the measurement from within the method. Secondly, for reasons explained in the previous section, when timing is done by the scheduler, any `setPriority()` method that takes place right before the method returns is not taken into account when calculating the execution time. For these two reasons the "timing by thread" measurements are greater. Now, because the overhead of calling the base scheduler method exists for all methods, while the overhead of `setPriority()` exists only for the `reschedule` methods (the `prepareTo` methods never set the priority before returning), the minimum overhead will be that of calling the method. Therefore, the smallest of the observed differences between the two tests will give us a worst-case estimate of the overhead of the method call. As we can see, the smallest difference is for the `prepareToSuspend(SLEEP)` method and is $d_{pt}=43.92-39.6=4.32\mu s$. Conversely, the largest of the observed differences will give us a worst-case estimate of the overhead for both the call and setting the priority. This is for the `reschedule()` method and is

d_{res}=*130.057-105.7=24,3μs*. These differences are used below to adjust the execution time estimates in the multi-thread tests.

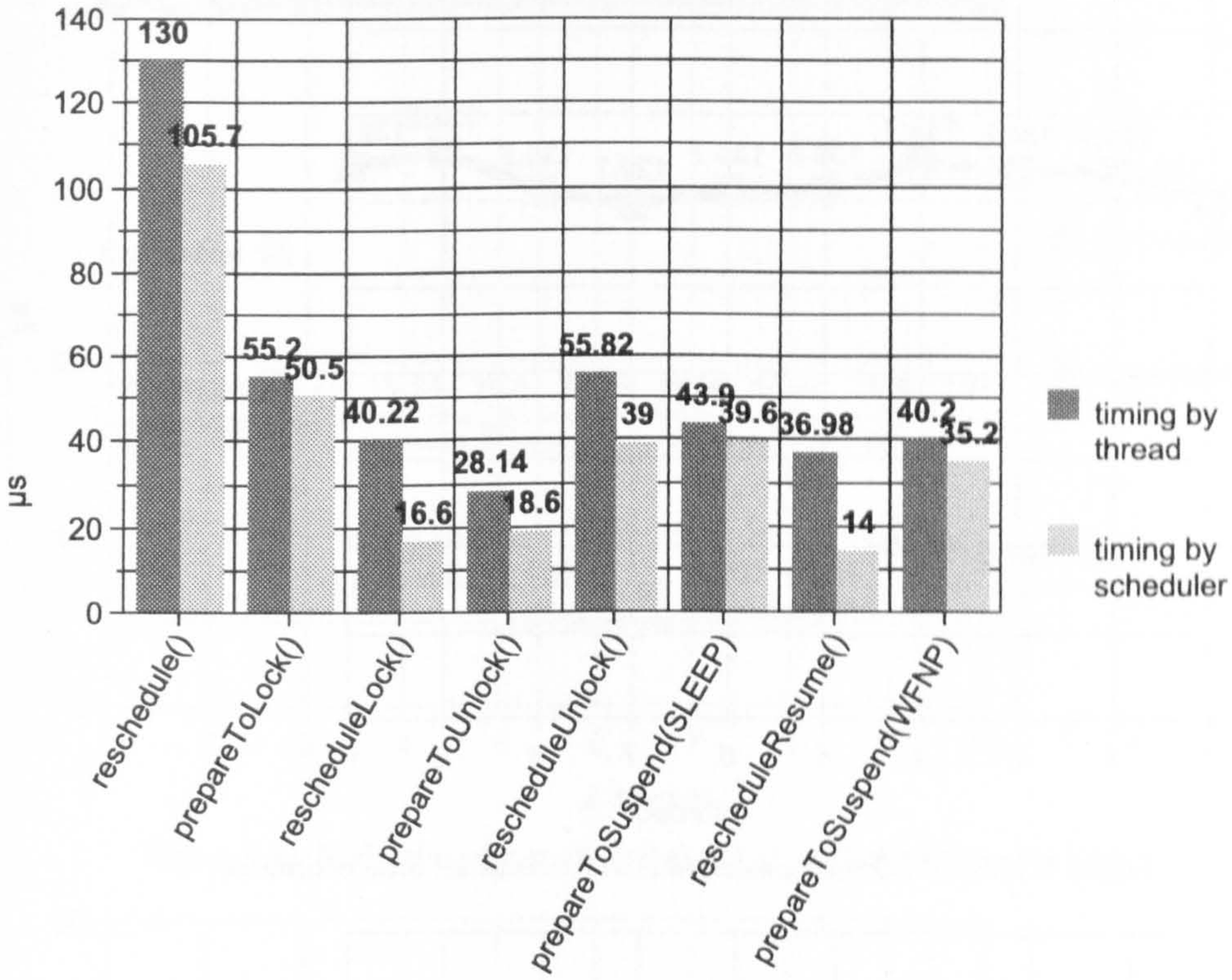


Figure 6.14: Base scheduler call execution times for one test thread

Next are presented the graphs of the execution time estimates for all the base scheduler calls. The graphs are made up of ten estimates for 1 to 10 threads running concurrently. Each graph is comprised of two lines. The lower line shows the original estimates as outputted by the test program, while the higher line shows the adjusted values after adding the d_{pt} and d_{res} values calculated above. d_{pt} is added to `prepareTo` method estimates, while d_{res} is added to `reschedule` method estimates.

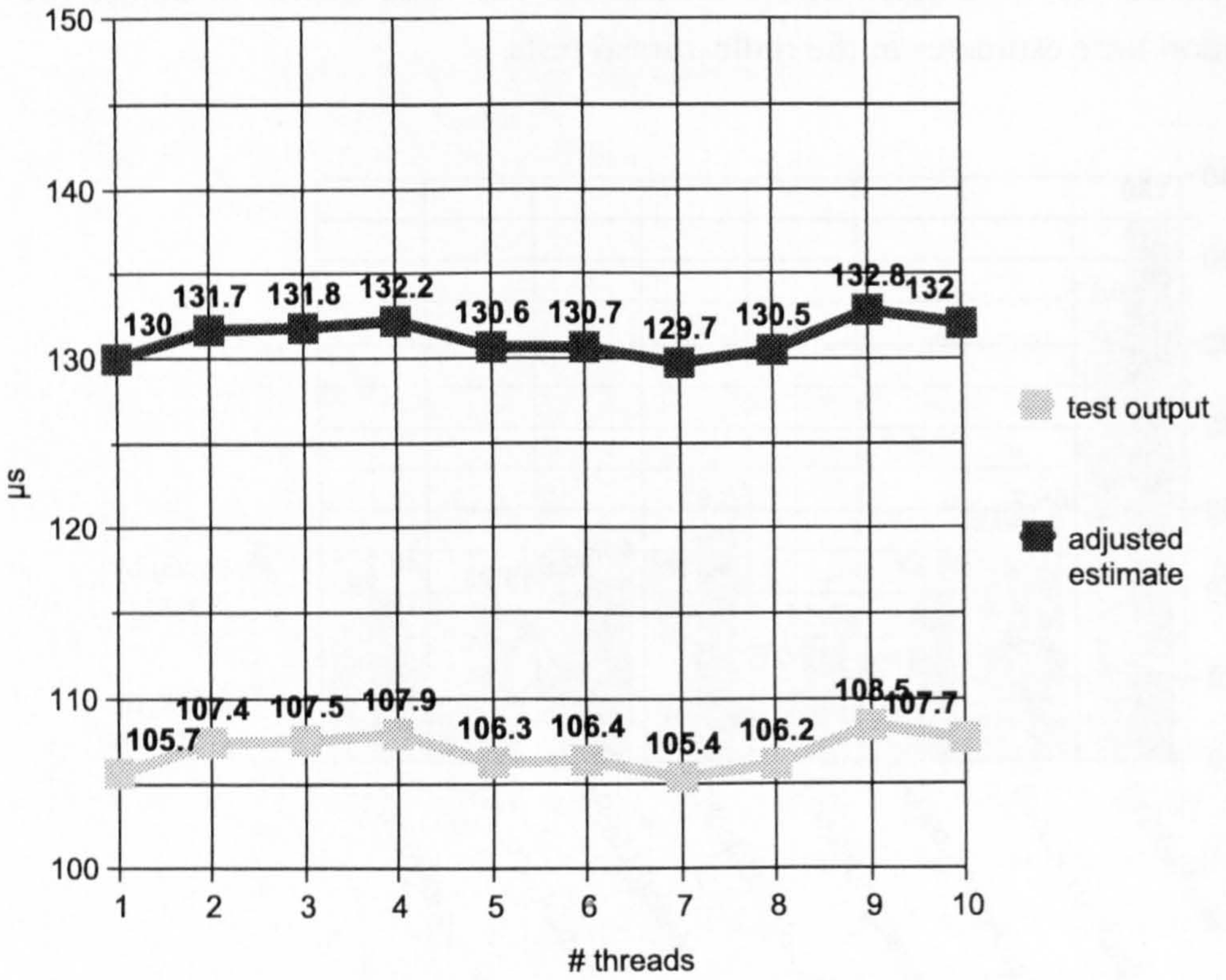


Figure 6.15: Multi-thread `reschedule()` execution time estimates

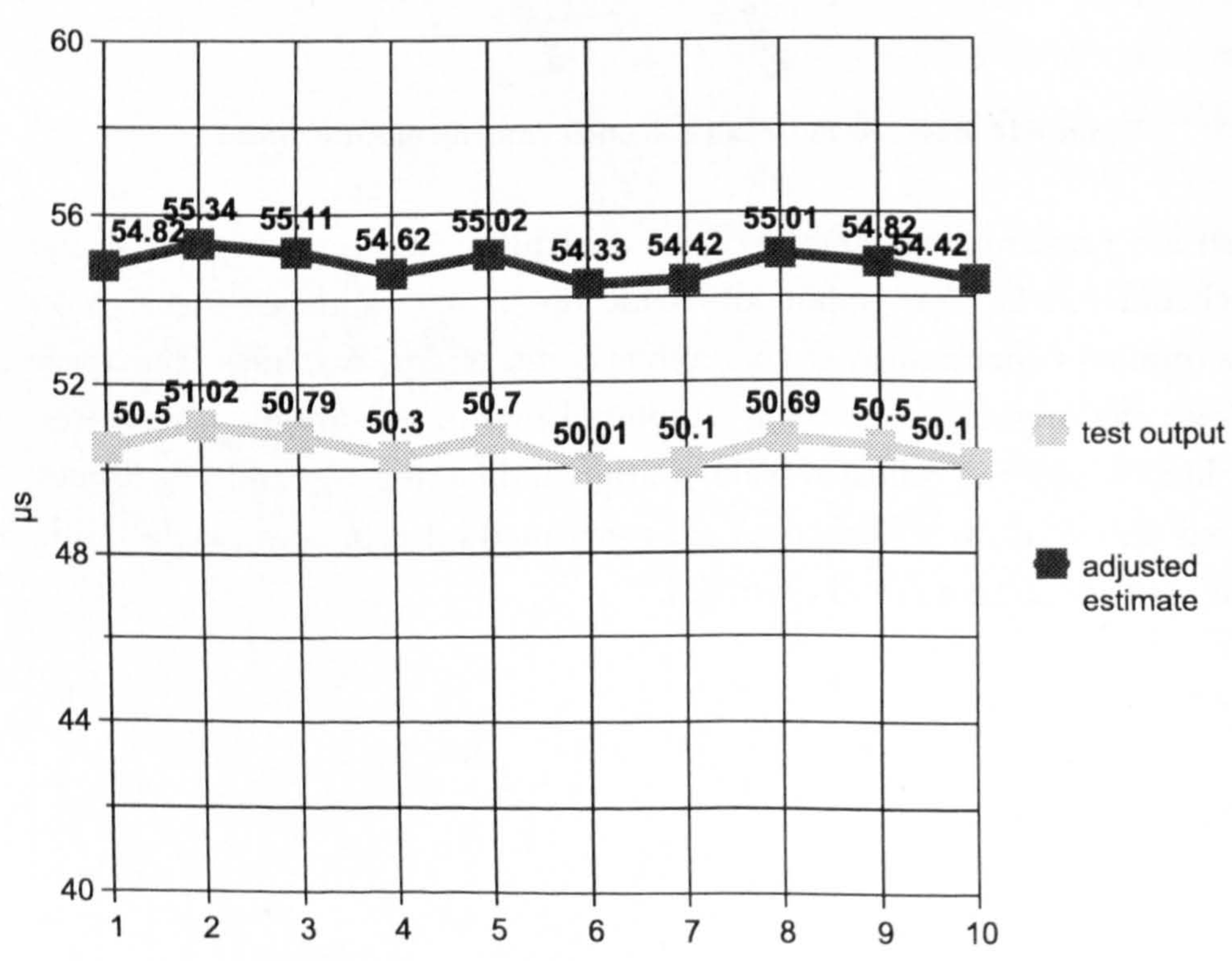


Figure 6.16: Multi-thread `prepareToLock()` execution time estimates

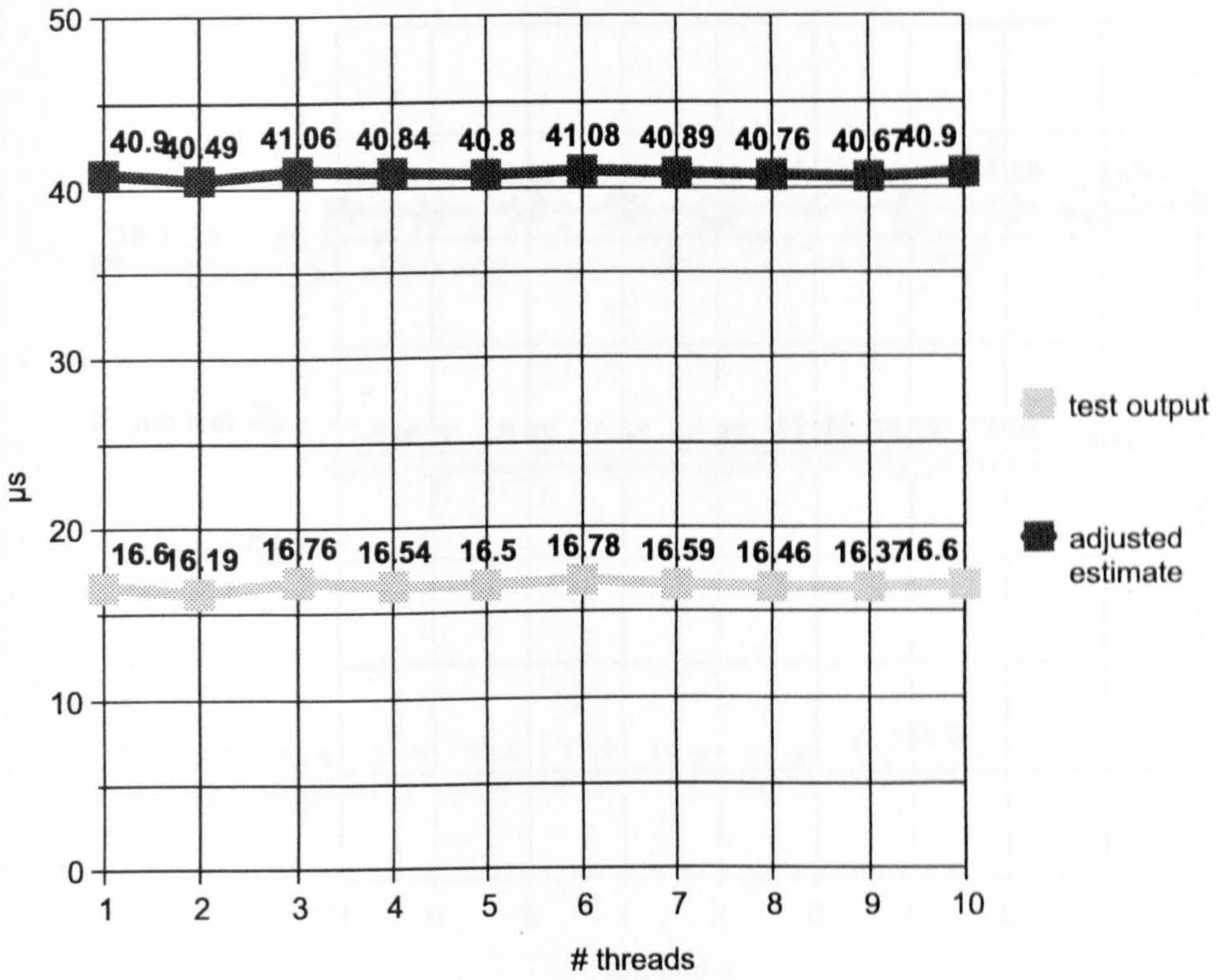


Figure 6.17: Multi-thread `rescheduleLock()` execution time estimates

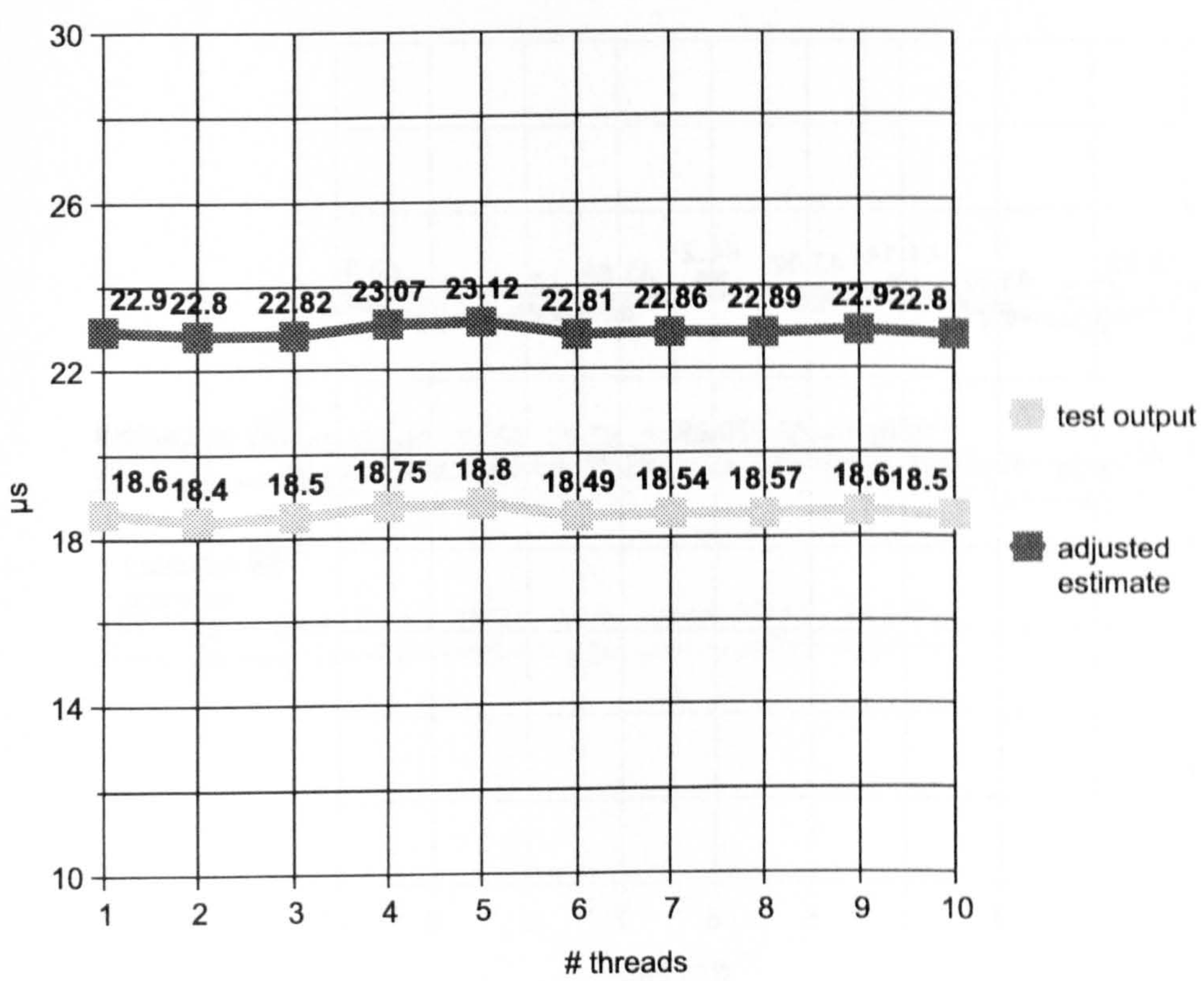


Figure 6.18: Multi-thread `prepareToUnlock()` execution time estimates

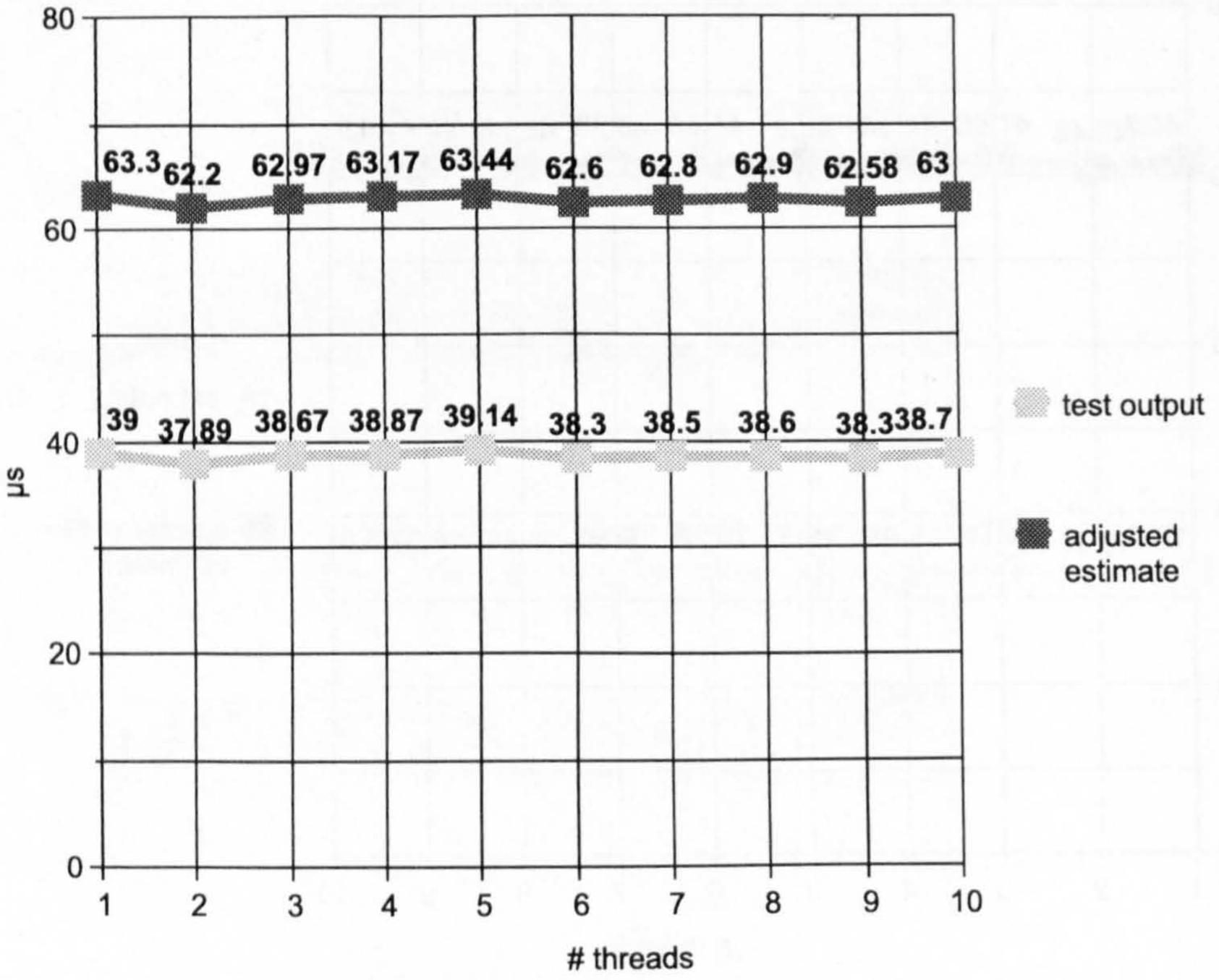


Figure 6.19: Multi-thread `rescheduleUnlock()` execution time estimates

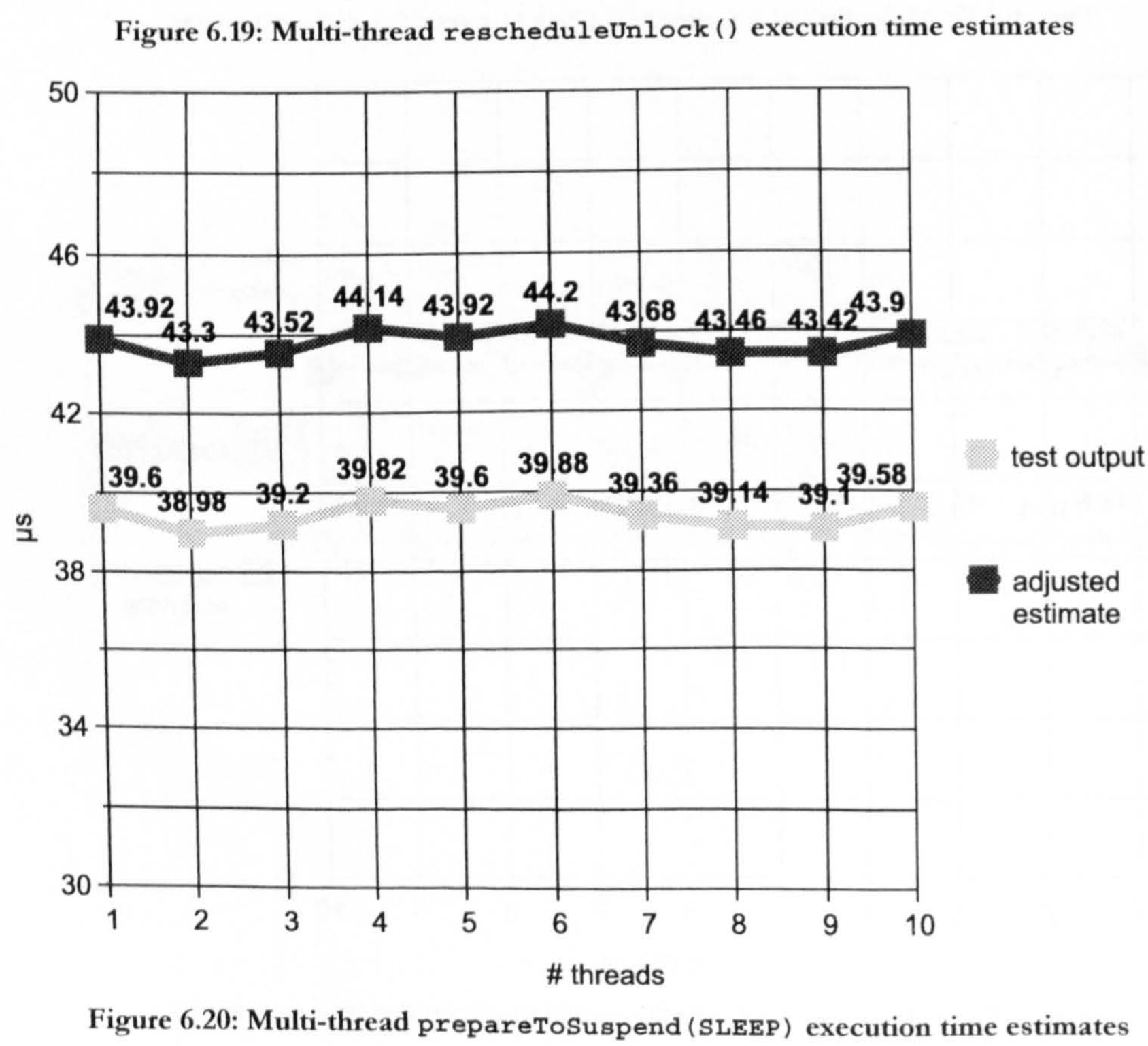


Figure 6.20: Multi-thread `prepareToSuspend(SLEEP)` execution time estimates

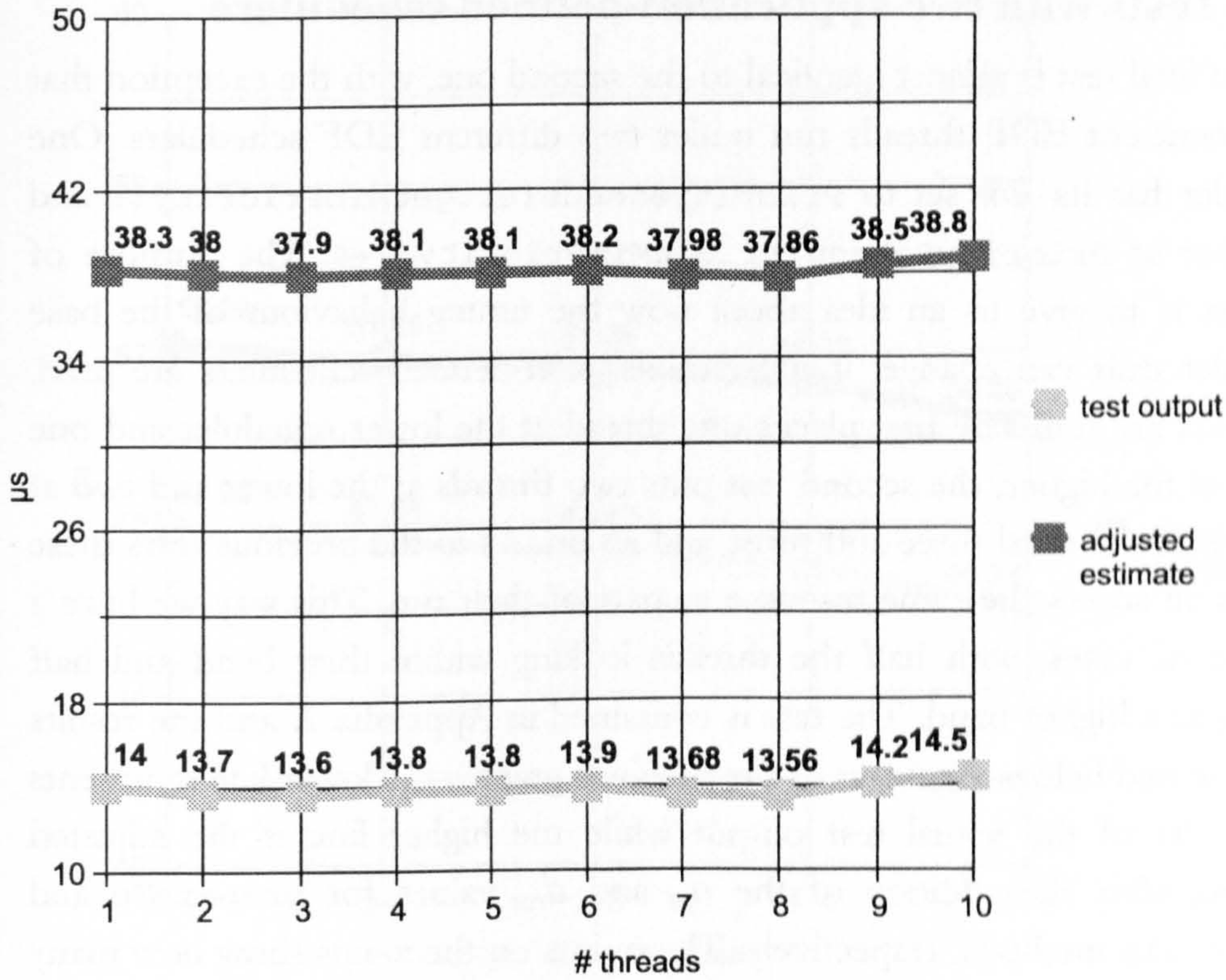


Figure 6.21: Multi-thread `rescheduleResume()` execution time estimates

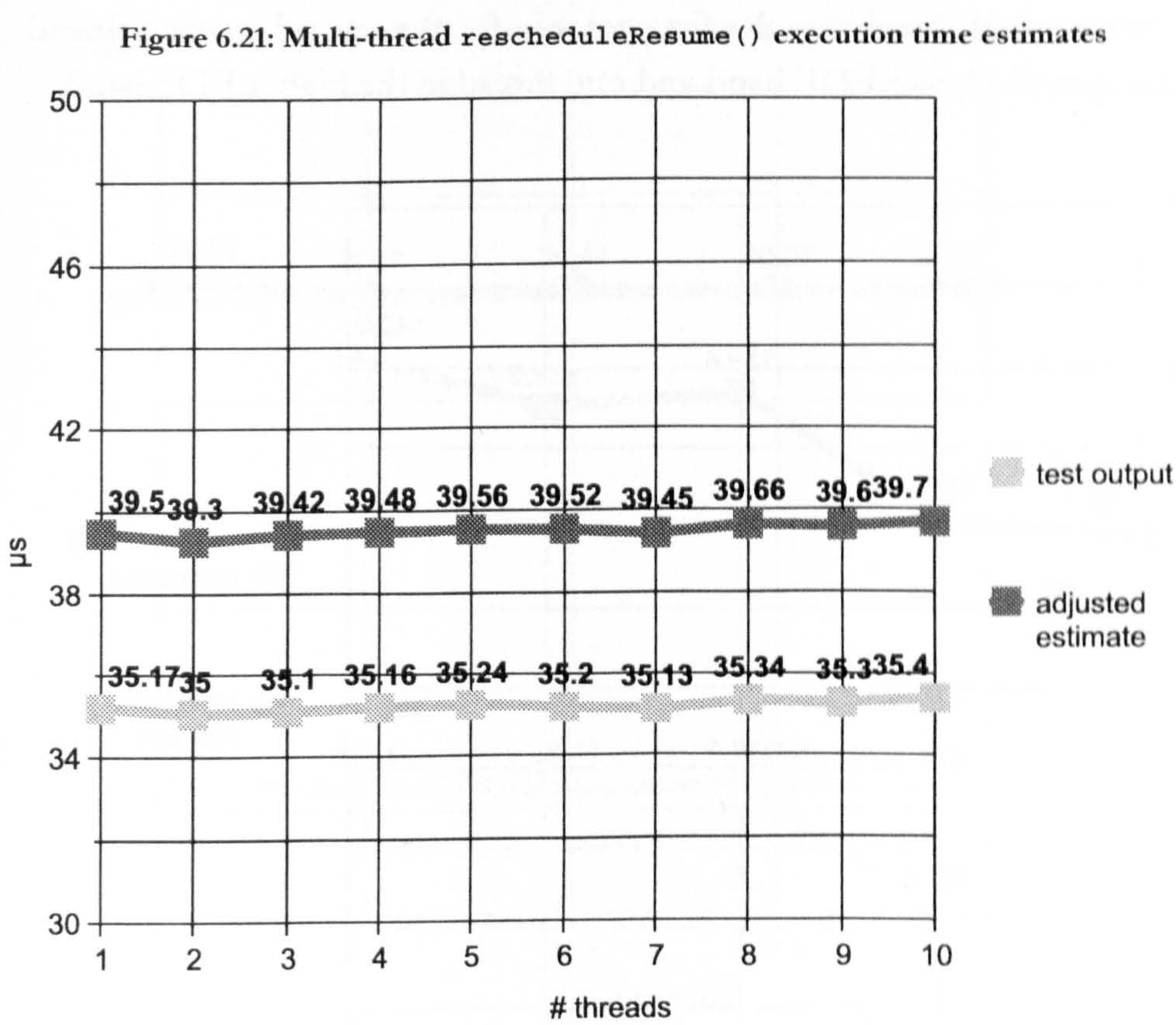


Figure 6.22: Multi-thread `prepareToSuspend(WFNP)` execution time estimates

6.4.4 Tests with two application-defined schedulers

The final test is almost identical to the second one, with the exception that the concurrent EDF threads run under two different EDF schedulers. One scheduler has its *low* set to `PriorityScheduler.getMinPriority()` and the other at `PriorityScheduler.getMinPriority()+4`. The purpose of this test is to give us an idea about how the timing behaviour of the base scheduler calls can change, if more application-defined schedulers are used. Five tests are run. The first places one thread at the lower scheduler and one thread at the higher, the second test puts two threads at the lower and two at the higher, the third three and three and so on. As in the previous tests these threads all access the same resource as part of their run. This way we have a mixture of cases, with half the threads locking within their band and half locking at a higher band. The test is contained in Appendix A and the results are presented below. Again, as in the previous graphs, the lower line represents the results of the actual test output while the higher line is the adjusted estimate, after the addition of the d_{pt} and d_{res} values for `prepareTo` and `reschedule` methods, respectively. The points on the x-axis show how many threads run on each band, e.g. the first point is for the test where one thread was running in the lower EDF band and one thread in the higher EDF band.

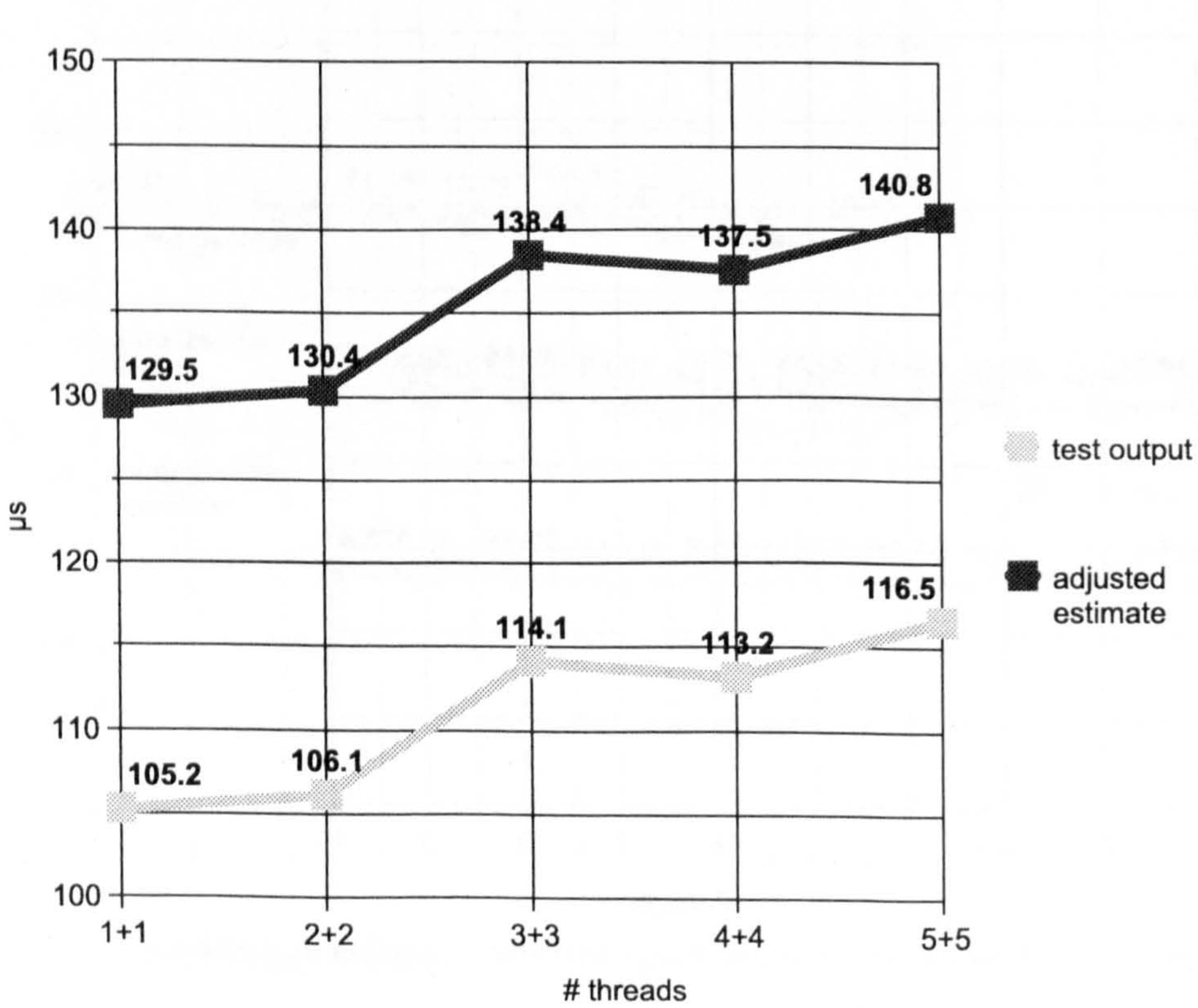


Figure 6.23: `reschedule()` execution time estimates for two EDF schedulers

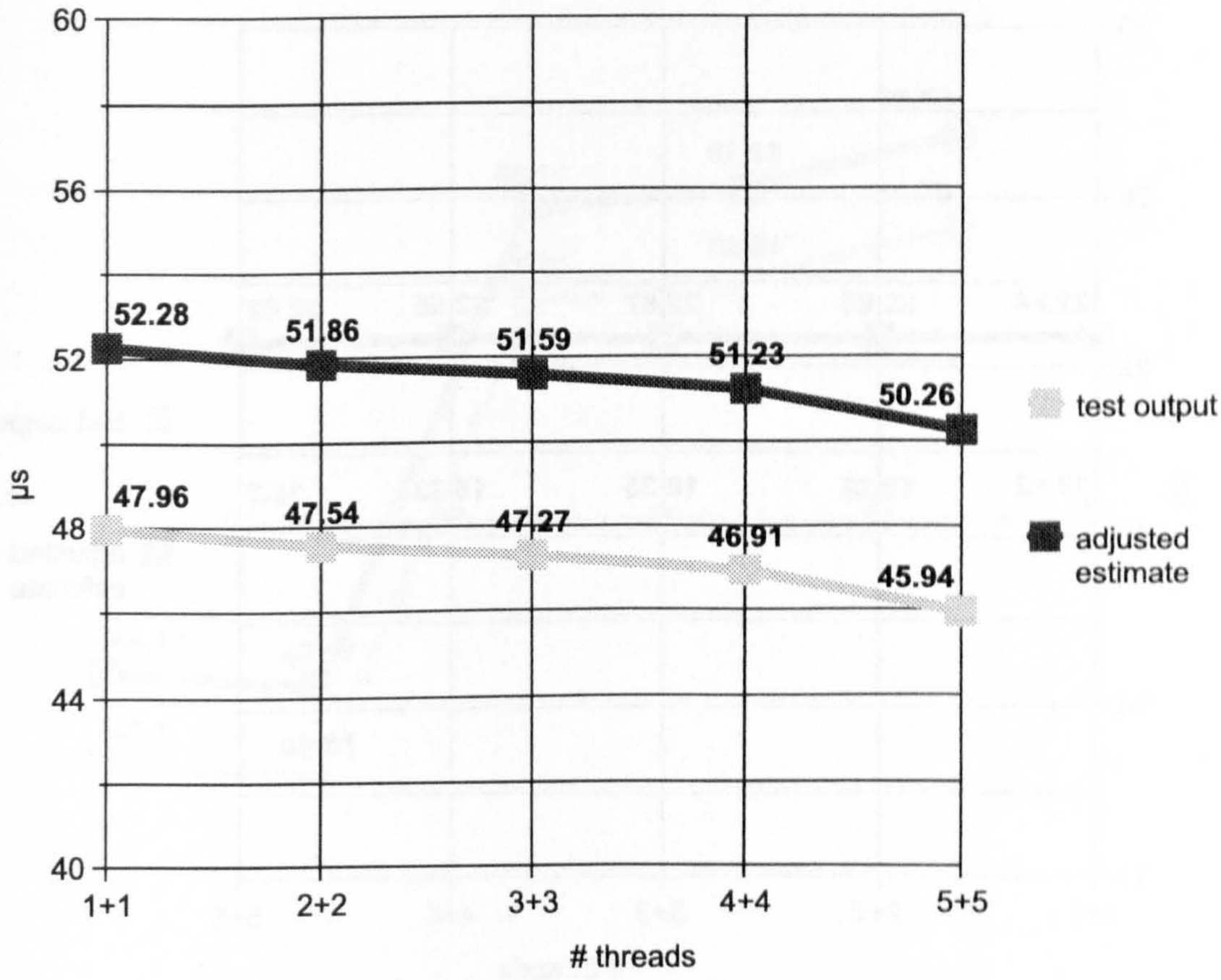


Figure 6.24: `prepareToLock()` execution time estimates for two EDF schedulers

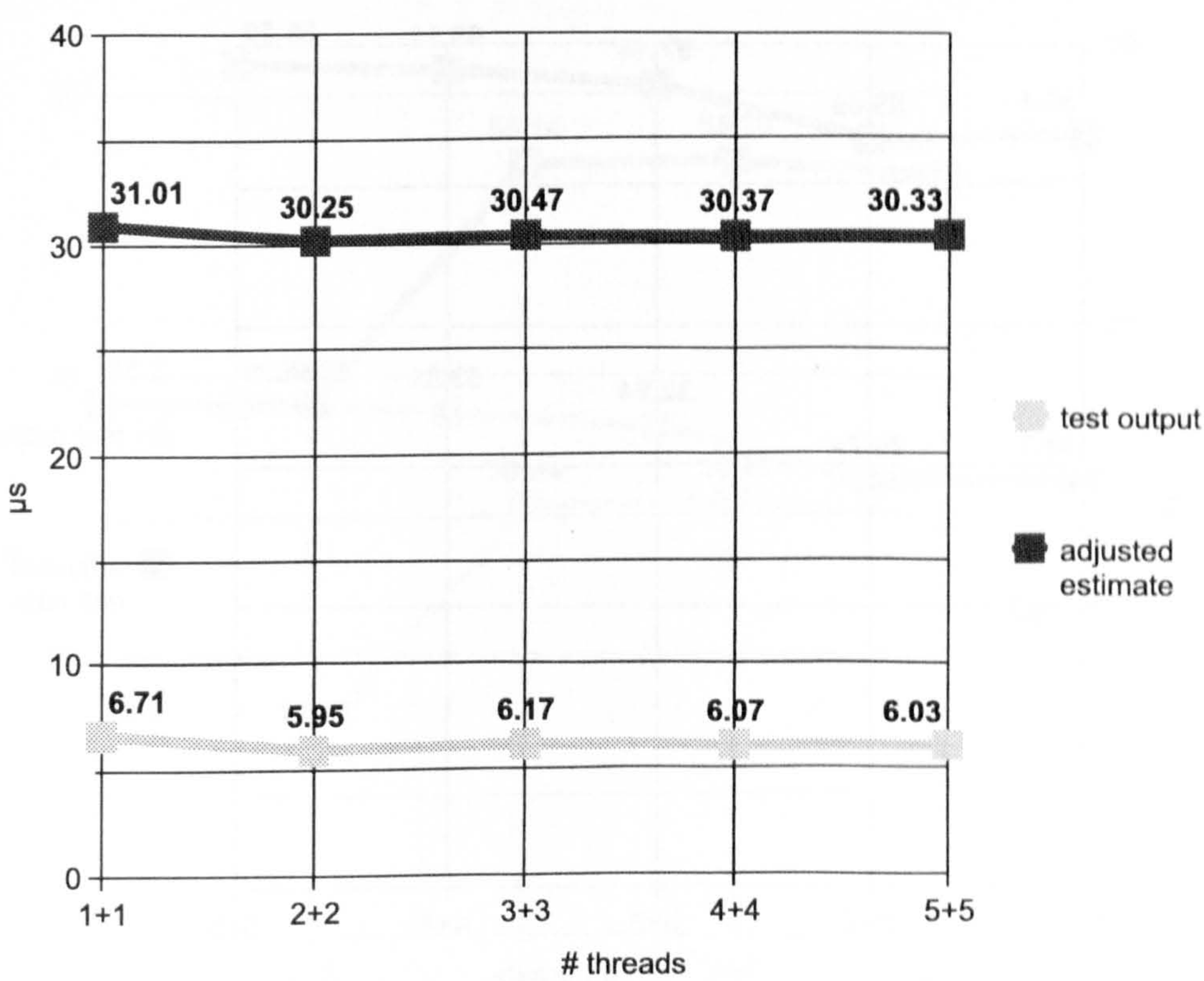


Figure 6.25: `rescheduleLock()` execution time estimates for two EDF schedulers

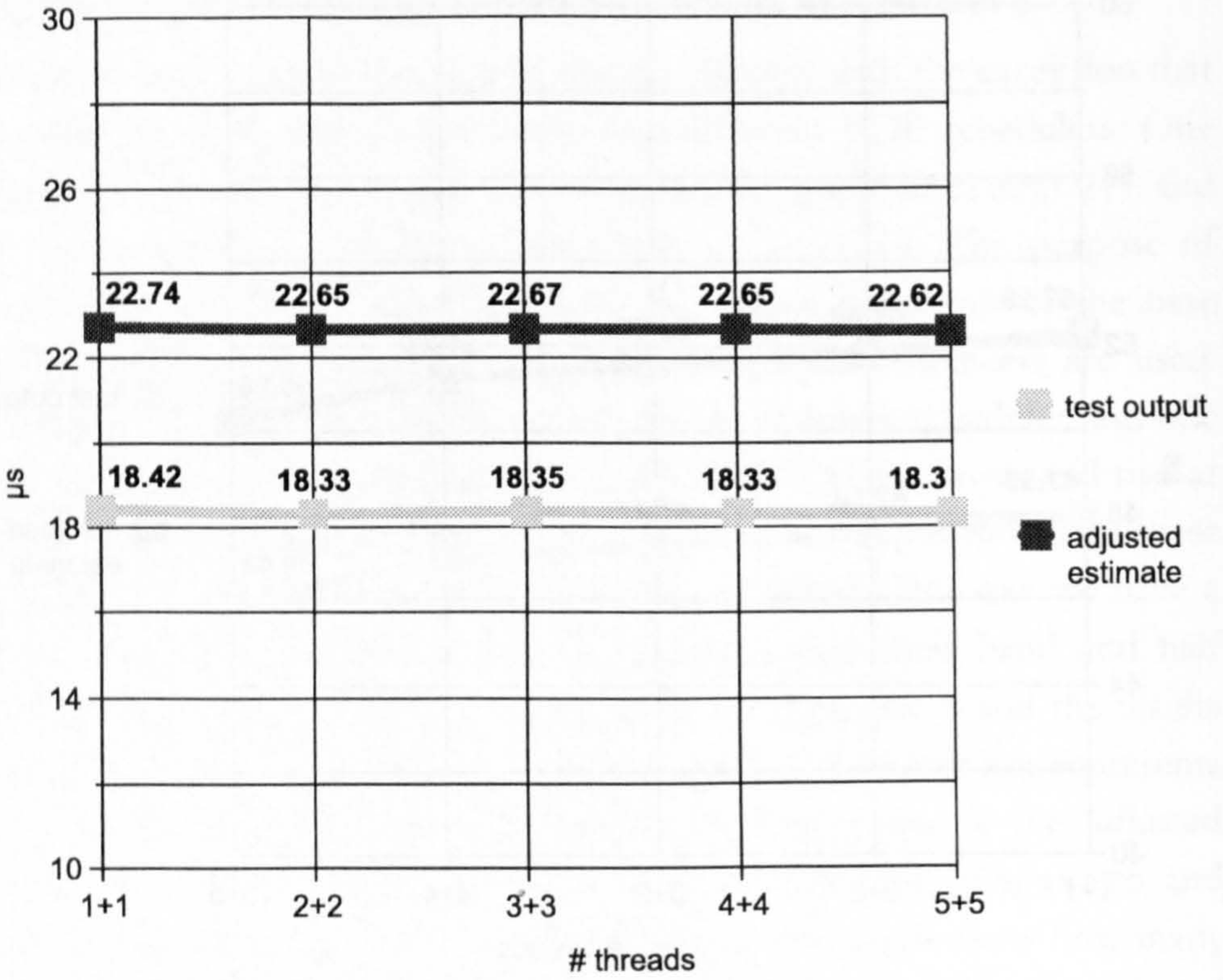


Figure 6.26: `prepareToUnlock()` execution time estimates for two EDF schedulers

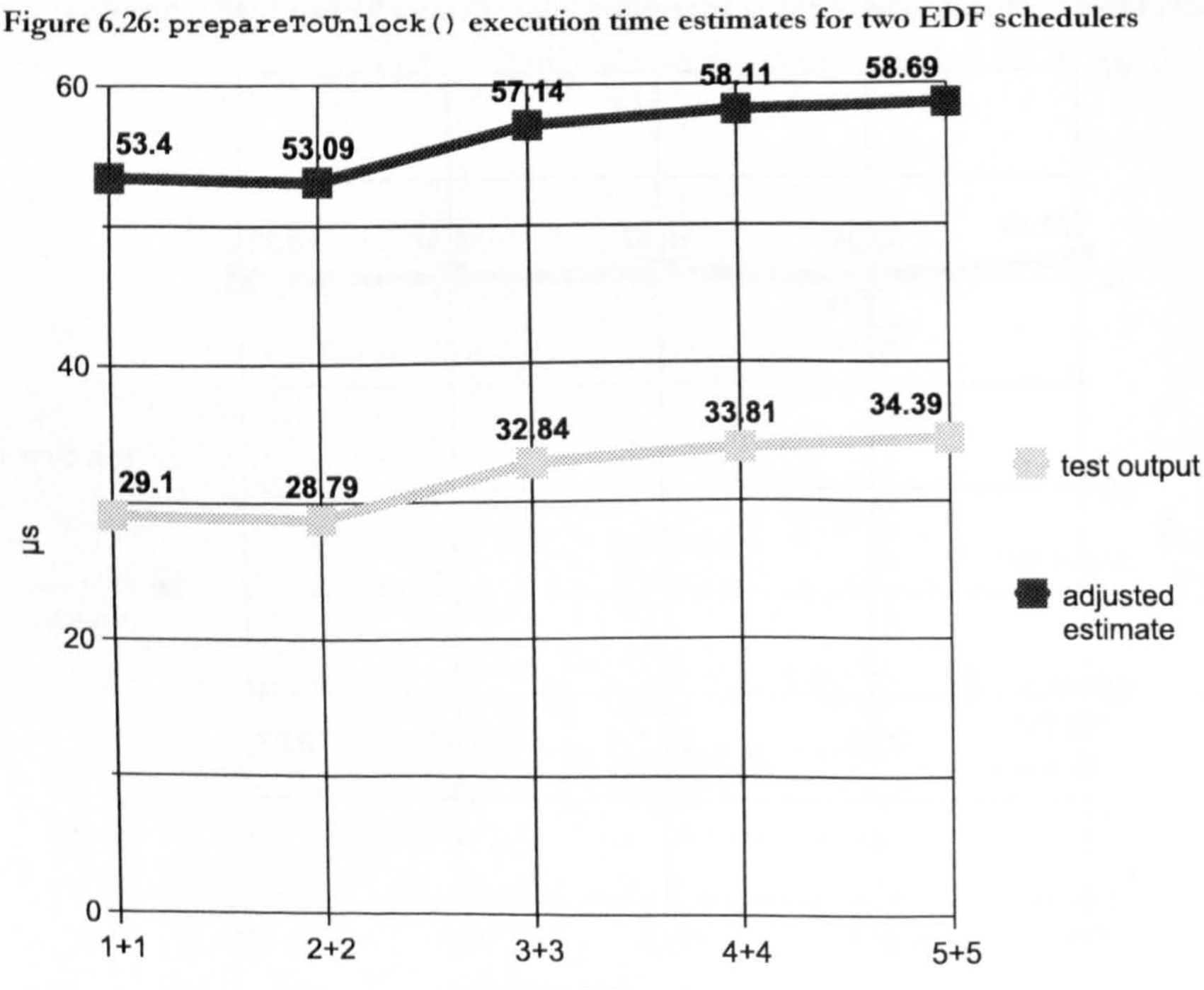


Figure 6.27: `rescheduleUnlock()` execution time estimates for two EDF schedulers

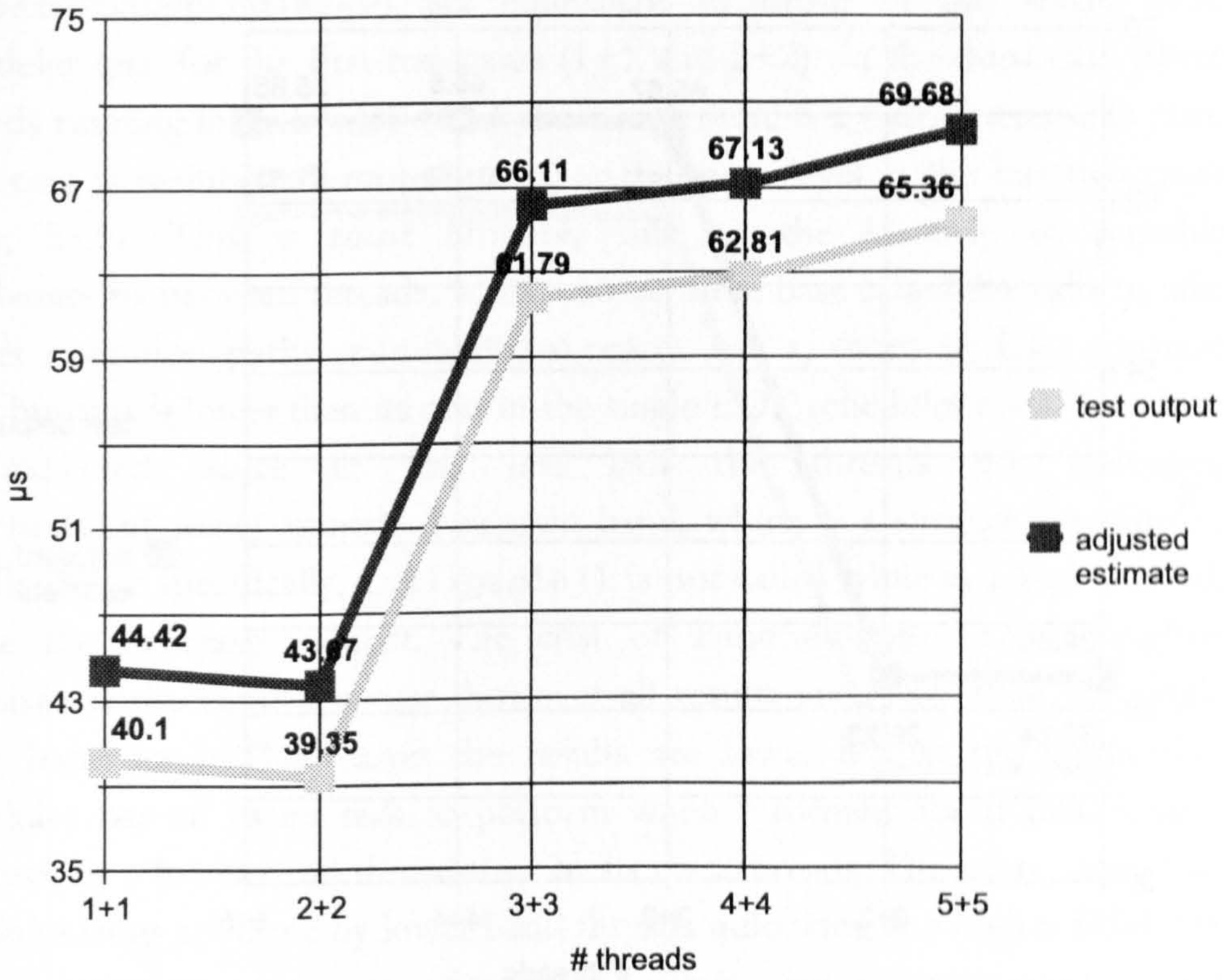


Figure 6.28: `prepareToSuspend(SLEEP)` execution time estimates for two EDF schedulers

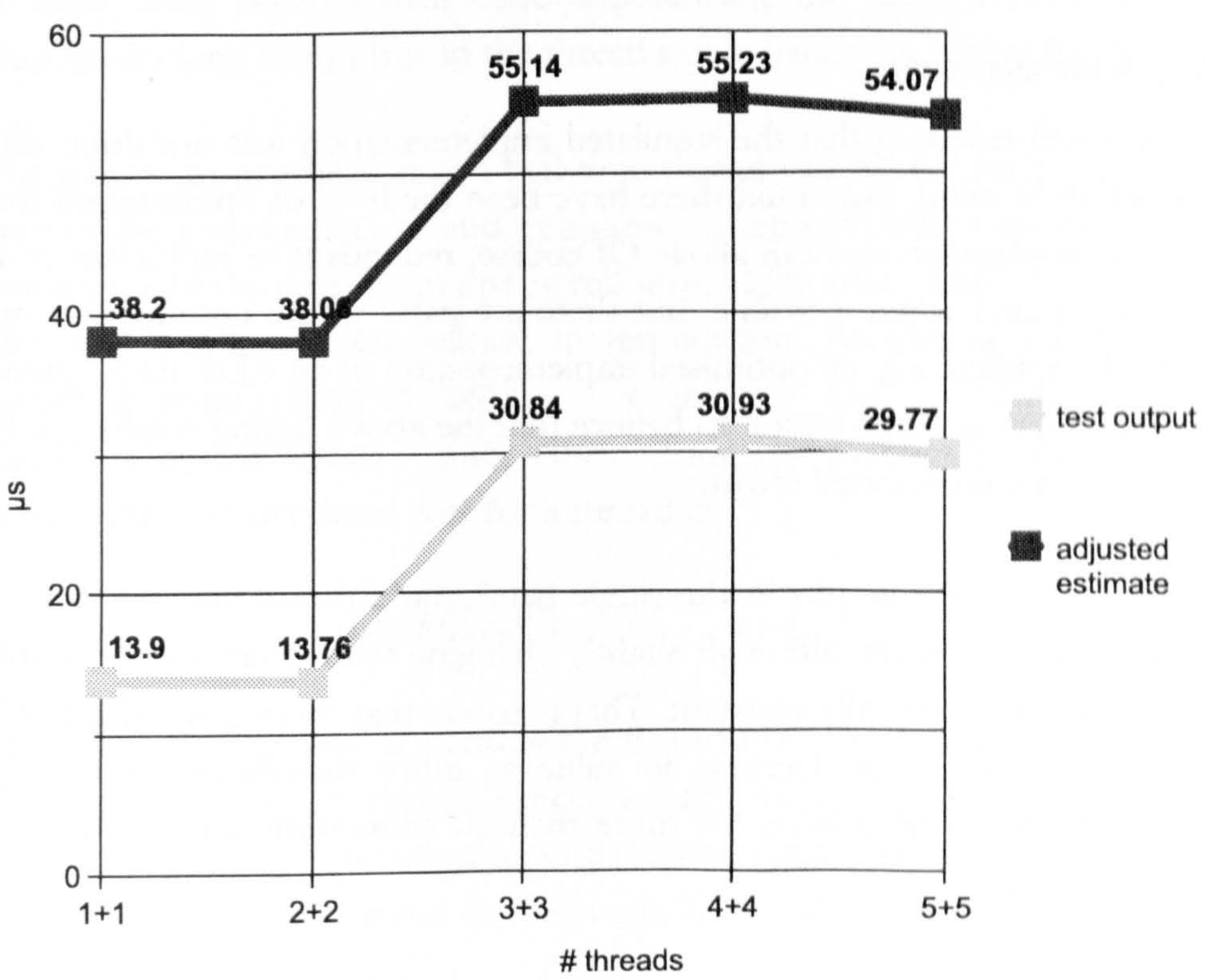


Figure 6.29: `rescheduleResume()` execution time estimates for two EDF schedulers

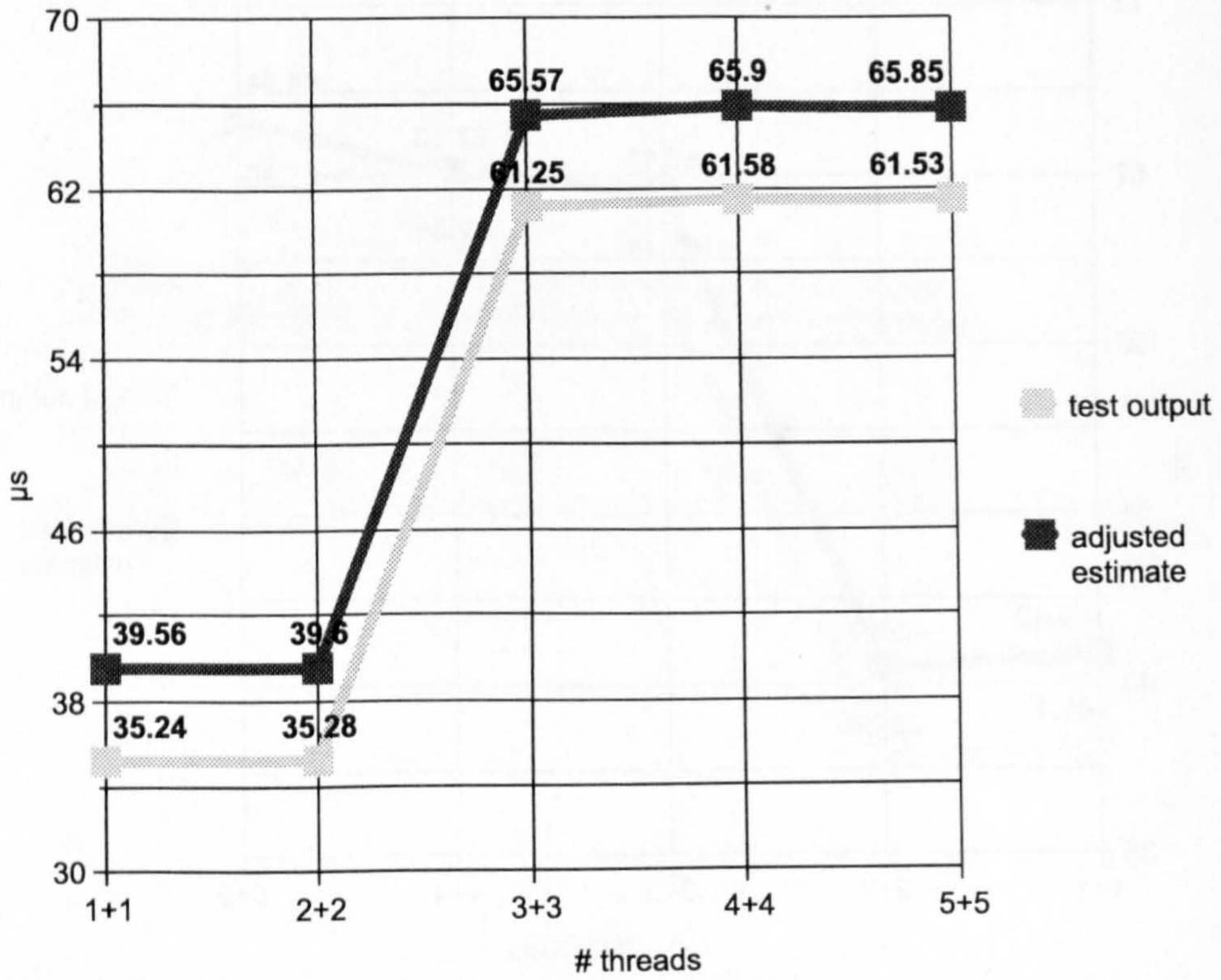


Figure 6.30: `prepareToSuspend(WFNP)` execution time estimates for two EDF schedulers

6.4.5 Assessment

It is worth restating that the simulated implementation was not done with optimisation in mind, nor could there have been the level of optimisation that only a full implementation can allow. Of course, redundant or inefficient code has been avoided where possible, but there are parts where optimisation can certainly be applied, e.g. an optimised implementation of an EDF ready queue. Therefore, there is every reason to believe that the above timing results can be improved in a concentrated effort.

By examining the graphs in the single band, multi-thread case we can see that the execution costs, although slightly changing with every test, as would be expected, are practically constant. That is to say that there is no observable tendency to increase or decrease in value as more threads are added. Of course, this cannot be proven for more than 10 concurrent threads, but is a good indication.

In contrast, some of the results of the last test with two EDF schedulers do not exhibit the same quality. The costs of `reschedule()`, `prepareToSuspend(SLEEP)`, `rescheduleResume()` and

`prepareToSuspend(WFNP)` are equivalent to those of the single EDF scheduler test, for the first two cases (1+1 and 2+2). In the third case (three threads running in each EDF band), however, there is a sharp increase in cost. This cost is maintained, more or less, at the same level in the last two cases (4+4, 5+5). This is most probably due to the increase in possible combinations between threads, which causes these base scheduler calls to take longer execution paths. `rescheduleLock()` has a, more or less, constant cost, but this is lower than its cost in the single EDF scheduler tests. This is to be expected, since in this test half the threads are executing `rescheduleLock()` outside their own band, which is a simpler situation to handle. More specifically, `isEligible()` is not called while at a higher band, hence the decrease in cost. The cost of `rescheduleUnlock()` slightly increases at the third test case, however all results are lower than the single EDF band tests. The reason the results are lower is that the application scheduler has an easier task to perform when informed about unlocking a resource by a lower band thread than by its own threads. Therefore, since half the unlockings are done by lower band threads unlocking at a higher band, the final execution cost is decreased. Only `prepareToLock()` and `prepareToUnlock()` yield values which are equivalent with those in the single band tests, because their code is practically the same irrespective of whether the locking takes place in the thread's own band or a higher band.

The minimum overhead that a real-time thread can expect each period is the sum of the `reschedule()` and `prepareToSuspend(WFNP)` methods. To that we must add the cost for `prepareToSuspend(THREAD_END)`, in order to get the total overhead form release to termination. As already mentioned, `prepareToSuspend(THREAD_END)` costs the same as `prepareToSuspend(WFNP)`, since their code is practically the same. Therefore, the base overhead cost for a thread is:

$$(C_{res}+C_{wfnp})*n + C_{end}\ ,\ n>1$$

where n is the number of periods the thread executes for. The minimum possible cost is when a thread executes only once, in which case only `reschedule()` and `prepareToSuspend(THREAD_END)` are executed. Based on the highest results measured in the single EDF scheduler tests, the worst measured case is:

$$C_{res} + C_{end} = 132.8 + 39.7 = 172.5\mu s$$

The cost of accessing a resource is the sum of costs for `prepareToLock()`, `rescheduleLock()`, `prepareToUnlock()` and `rescheduleUnlock()`. Again based on the highest results in the single EDF scheduler tests, the worst measured case is:

$$C_{pTL}+C_{resL}+C_{pTU}+C_{resU} = 55.34+41.08+23.12+63.44 = 182.98\mu s$$

The cost of suspension is the sum of `prepareToSuspend()` and `rescheduleResume()`. In our estimates we use the `prepareToSuspend(SLEEP)` as representative of all `prepareToSuspend()` calls. Based on the highest results in the single EDF scheduler tests, the cost for suspension is:

$$C_{pTS} + C_{resR} = 44.2 + 38.78 = 82.98\mu s$$

Finally, the total cost overhead for a band thread in a single period is given by the following formula:

$$C_{res}+(C_{pTL}+C_{resL}+C_{pTU}+C_{resU})*l+(C_{pTS}+C_{resR})*k+C_{wfnp}$$

where l is the number of resources a thread locks during its period, and k the number of self-suspensions a thread performs. Usually, k will be zero. If we assume that a typical thread locks three resources during its period, then a typical cost overhead for a single period is:

$$C_{res}+3*(C_{pTL}+C_{resL}+C_{pTU}+C_{resU})+C_{wfnp} = 132.8+3*182.98+39.7 = 721.44\mu s$$

If we assume a more complex system with threads locking outside their bands, then we have to replace the measurements from the single EDF scheduler tests with those of the two EDF scheduler tests. Thus, the above typical cost overhead for a single period becomes:

$$C_{res}+3*(C_{pTL}+C_{resL}+C_{pTU}+C_{resU})+C_{wfnp} =$$
$$140.8+3*(52.28+31.01+22.74+58.69)+65.9 = 700.86\mu s$$

To put these numbers into perspective a simple test has been set up to measure the execution cost of starting a thread, changing priority, locking a resource, unlocking, calling `sleep()` and calling `waitForNextPeriod()` in a

normal RTSJ `RealtimeThread`. The measurements are made, as usual, by using a `getTime()` before and after the code in question. `sleep()` is called with a zero argument, so it practically does not suspend and the time measured is the actual time of the `sleep()` method. It is worth noting that the cost of changing a thread's priority is perhaps the most crucial element in the framework's performance, since the framework protocol's basic mechanism is the manipulation of thread priorities. Therefore, it is important to have an idea about how expensive this operation is. This test is also contained in Appendix A. The times returned from this test were:

Table 6.1: Execution times of typical RTSJ operations

Initial start of a thread	*122ms*
Changing priority	*15μs*
Synchronizing on an object	*7μs*
Releasing an object	*6μs*
`sleep()`	*45μs*
`waitForNextPeriod()`	*1.1ms*

With these measurements at hand we can calculate the overhead percentage that the framework imposes on these operations. The overhead for the initial start of a thread is the cost of `reschedule()`. For locking and unlocking an object the overhead is the sum of `prepareToLock()`, `rescheduleLock()`, `prepareToUnlock()` and `rescheduleUnlock()`. For `sleep()` it is the sum of `prepareToSuspend(SLEEP)` and `rescheduleResume()`. Finally, for `waitForNextPeriod()` the overhead is produced by `reschedule()` and `prepareToSuspend(WFNP)`. It is important to note here that on the test platform used, the RTSJ reference implementation is not able to properly perform priority inheritance [TimeSys 2007]; that is, no priority change takes place when a thread has synchronized on a resource and a higher priority thread requests the same resource. Therefore, the above execution cost estimate for a synchronization operation is not correct. For a basic application of the priority inheritance protocol this time should be increased by the cost of two priority changes: one increasing the active priority of the lower thread upon request of the resource by the higher priority thread, and one decreasing the active priority of the lower thread upon releasing the resource. Thus, the total cost for synchronising on an object is *7+6+15+15=43μs*. The following table lists the costs of the operations with and without the framework and gives the increase in percentage. The values used for the base scheduler calls are the highest

measured by the previous tests. The final row is the increase in one period of a typical thread case, as presented earlier.

Table 6.2: Increase percentages due to the framework

	RTSJ	FMSF	% increase
Initial start of a thread	*122ms*	*+140.8µs*	*0.115%*
Synchronization	*43µs*	*+182.98µs*	*425.53%*
`sleep()`	*45µs*	*+124.91µs*	*277.56%*
`waitForNextPeriod()`	*1.1ms*	*+206.7µs*	*18.8%*
Typical thread period	*1.229ms*	*+721.44µs*	*58.7%*

Based on percentages the increase might seem significant, for example in the case of synchronization. However, percentages can be misleading. For example, *1ns* increased to *5ns* is an increase of *400%*, but *5ns* is still a miniscule amount of time. Moreover, an increase of this scale is to be expected, since the framework introduces a non-negligible amount of new logic into these operations. This is the price that has to be paid for the increased flexibility offered. Especially with regard to synchronizations it has to be said that even with multiple threads waiting on an object the framework protocol would still incur the same amount of overhead, whereas with priority inheritance we would have multiple priority changes, which would increase the total cost of the operation. All in all, the crucial factor is the application domain that the framework is going to be used in and whether the applications that might use it can bear the overhead. It is viewed that the increase of under a millisecond in a typical thread period is satisfactory for the types of applications that would need to make use of flexible scheduling.

6.5 Summary

This chapter aimed at demonstrating the applicability of the Flexible Middleware Scheduling Framework to existing real-time middleware systems and applications. To this effect an implementation of the framework in the Real-Time Specification for Java has been described. Such an implementation accommodates an initial goal of the RTSJ, which was to support the state-of-practice in real-time systems development and mechanisms to allow advances in state-of-the-art. The work in this chapter provides an extension to the RTSJ's scheduling framework, thus offering flexible scheduling in a portable way. The approach is backward compatible with the current version of the

RTSJ in that programs that do not define their own schedulers will be able to execute unchanged even on a version of RTSJ that implements the framework. Furthermore, these programs will not need to experience any overhead, apart from the negligible execution time of a single `if` statement, which will check if a particular thread is scheduled by an application scheduler or not.

Secondly, this chapter provided a simulated implementation of the framework on RTSJ, both as a proof of concept and as a test bed for measuring the introduced overhead. The simulated implementation is done at the application level, with the application programs explicitly calling the framework's base scheduler calls at each scheduling point. Based on this implementation a series of tests was conducted in order to provide insight on the execution time overhead. The tests were of three types: a single thread test, a single band, multithread test, and a two-band multithread test. These tests, in conjunction with execution time measurements of basic RTSJ operations (synchronization, thread release etc.), showed that the overhead is reasonable. With a typical execution time increase of under a millisecond per period the framework is deemed appropriate for the types of real-time applications that need the services of flexible scheduling.

Chapter 7

Conclusions and Future Work

This chapter summarises the contributions of this thesis, thus concluding the research effort. Furthermore, it provides suggestions for further development of the proposed framework. The principal aim is to enable real-time program execution middleware to support arbitrary scheduling policies chosen by the application. In other words, this thesis facilitates application-defined flexible scheduling in real-time middleware systems.

Chapter 2 of the thesis is concerned with placing this work within the framework of a broader effort to introduce flexible scheduling to real-time systems. Through this survey comparative similarities have been highlighted and also differences in approach. Three major areas of real-time computing are examined; real-time middleware, including distributed middleware, real-time operating systems, and real-time programming languages. Although other approaches are useful within their corresponding domains, none can satisfactorily accommodate flexible scheduling in real-time program execution middleware.

The purpose of Chapter 3 is to provide insight in deciding the resource sharing protocol used in the proposed framework. The chapter conducts a survey of the best known resource sharing protocols, in order to adjudge which protocol forms the best basis for efficient resource sharing between arbitrary scheduling policies. In the course of the chapter the relationship between the different protocols is explained and their differences are summarised in a table. The Basic Preemption Ceiling Protocol is chosen for two main reasons. Firstly, it allows resource access eligibility to be expressed in a scheduling policy neutral way, and secondly, it provides deadlock avoidance and bounded eligibility inversion even when tasks self-suspend while holding resources.

Chapter 4 is the main locus of this thesis, presenting the Flexible Middleware Scheduling Framework. It first sets out the rationale behind the framework's protocol by explaining the basic mechanism of enforcing arbitrary scheduling policies through the manipulation of task priorities. The concept of a scheduling band is introduced, which defines the operating limits of each application scheduler. The chapter proceeds with explaining the mechanism of scheduling band operations and their role in enforcing the protocol. Then the framework's resource sharing protocol is presented, which is an amalgam of the Basic Preemption Ceiling Protocol and the Priority Ceiling Emulation Protocol. Finally, the framework's protocol itself is described, covering all scheduling band operations. This chapter's contribution is the introduction of a framework for flexible scheduling in real-time middleware systems. This framework allows applications to define scheduling policies of their choice; it allows multiple arbitrary policies to co-exist in the system; and it allows tasks under different policies to safely and efficiently share resources. This way applications with diverse scheduling needs, across the soft/hard real-time spectrum, can not only co-exist in a system, but also cooperate with one another.

In Chapter 5 an initial evaluation of the framework is given which shows its viability. This is demonstrated in three ways. The first part is the description of an EDF application-defined scheduler. This is presented as a case study and identifies mechanisms that can be used by any application-defined scheduler. The second part presents a verification of the framework protocol's operation. This is done through the use of a model, constructed in the UPPAAL model checker, and provides strong indication that a correct implementation of the

protocol is fault-free. The third part of the chapter examines the types of different scheduling policies that can be supported through the use of the framework. The examination shows that a significant number of both hard and soft real-time scheduling policies can be supported. In general, the contribution of this chapter is to demonstrate the applicability and viability of the flexible middleware scheduling framework in addressing a wide range of scheduling needs.

Finally, Chapter 6 concludes the evaluation of the framework by examining the issue of implementing it. The Real-Time Specification for Java is chosen as the implementation platform, because of its current impetus. The chapter describes two types of implementation. The first description is that of a full implementation of the framework that contains all the necessary class descriptions. However, because undertaking a full implementation is outside the scope of this thesis, a second approach is presented, that of a simulated implementation. In this latter case the framework classes are implemented as an application level library, with certain alterations in the class descriptions being necessary. Following the description, timing measurements are presented based on the simulated implementation. The overhead imposed by the framework operation is shown to be adequately low. This chapter's contribution is three-fold. Firstly, it demonstrates that the FMSF framework can be feasibly implemented. Secondly, and more importantly, by describing how the framework can enhance the RTSJ with the addition of flexible scheduling this chapter highlights the framework's merits and ability to be easily introduced to an existing middleware system. Last but not least, it presents measurements on the execution time overhead that the framework introduces, proving that the framework can viably support real-time applications.

7.1 Contributions

The major contribution of this thesis is the introduction of a framework for flexible scheduling in real-time program execution middleware. The introductory chapter sets two design constraints: support for state-of-the-art practices and portability of the middleware implementation. In Chapter 2 a number of approaches to flexible real-time scheduling were reviewed and were found lacking in a number of aspects. The FMSF framework honours the constraints set and addresses all the limitations of the existing approaches.

More specifically, the framework allows applications to define and use their own arbitrary scheduling policies. It allows such different policies to co-exist in one system, with an application being able to select a policy at runtime. Moreover, it allows different applications running under different policies to share resources amongst them with minimum execution eligibility inversion, i.e. one instance. Applications can also define and use their own resource sharing protocol. Additionally, applications not using the framework can co-exist and share resources with applications that are using it. This combination of capabilities is a powerful tool for the specification of both state-of-practice and novel approaches in real-time scheduling. Furthermore, the framework promotes portability by being based on the most common of all real-time scheduling policies, the fixed-priority preemptive scheduler. This greatly facilitates the framework's implementation on different platforms.

With respect to the framework, the thesis contributes to its analysis and proof of correctness. In Chapter. 4, proofs are provided for such basic properties of the FMSF protocol as deadlock-free operation, bounded eligibility inversion and highest eligibility dispatching. In addition, Chapter 5 contains a verification of the framework protocol's operation through model checking. A further contribution is the high-level description, in Chapter 5, of an implementation of an EDF scheduler to be used in conjunction with the framework. The same chapter also provides an analysis on the scheduling policy categories that can be used in conjunction with the framework.

This thesis also provides contributions in the area of resource sharing protocols. Chapter 3 gives a thorough analysis of the most widely used resource sharing protocols. One contribution is the exposure of the interconnection between these protocols, and in particular the roles of eligibility inheritance and the preemption level test. This analysis has lead to the exposure of a flaw in the Preemption Level Protocol, which is a significant contribution in its own right. Another contribution is the survey amongst resource sharing protocols on their suitability for heterogeneous scheduling environments, i.e. environments where multiple scheduling policies co-exist. One last contribution in this category is the analysis of the types of scheduling policies that can be used with preemption levels.

A key contribution for real-time Java application developers and researchers is the proposed implementation of the framework in the RTSJ. This realises the initial intentions of the RTSJ of supporting a range of

schedulers. To this effect a number of class descriptions are given in Chapter 6. A final contribution is a simulated implementation of the framework on the RTSJ, based on which execution time overheads are measured, showing that the framework can be feasibly implemented.

7.2 Future work

Although the framework presented in this thesis forms an integrated solution and has been shown to be viable, a number of areas exist in which further work could be performed. These are the following:

1. Carry out a full implementation of the framework, as laid out in Chapter 6. Based on this implementation better timing measurements could be taken, giving us a better view of the capabilities and also the limitations of the framework. This effort would focus on the implementation issues and difficulties rather than the theoretical aspects of the approach.
2. Enable the cooperation of different schedulers, thus allowing the development of more complex scheduling schemes. The framework could allow feedback to be passed from one scheduler to another in order to achieve coordination of scheduling efforts. This could be useful in, for example, soft real-time schedulers of multimedia applications, where one scheduler would decrease its CPU reservation in order for another to increase its own.
3. Add direct support for round-robin scheduling. As part of the framework specification round-robin scheduling could be added as an optional requirement for the priority base scheduler. The problem of supporting this scheduling scheme on a priority scheduler has already been tackled in [Burns et al. 2003] and this can form the basis for adding it to the framework. The round-robin scheme is particularly helpful for multimedia applications and direct support for it would be ideal, since, as we have seen in Chapter 5, its implementation as an application scheduler can be problematic.
4. Investigate how the framework can be extended to support scheduling of distributed systems. Already the framework has been reviewed by members of the Distributed Real-Time Java effort [Anderson and Jensen 2006]. The framework specification could be extended to support distributed real-time scheduling in any distributed technology,

e.g. by requiring that the framework be present in each node of a distributed system. Furthermore, in conjunction with the first suggestion, schedulers in different nodes could be passing feedback between them, guiding the execution of distributed threads on the different nodes.

7.3 Final comment

In Chapter 1 the proposition of this thesis has been given, which is the following:

A two-level scheduling framework, having a fixed-priority base scheduler and support for other schedulers through dynamic manipulation of task priorities, provides an appropriate framework for flexible scheduling in real-time middleware software.

Subsequent chapters lay the groundwork for and present an exact protocol for such a framework, termed the Flexible Middleware Scheduling Framework (FMSF). Its correctness is checked and a measure of its capabilities is demonstrated, first by reviewing the range of possible supported schedulers and secondly by implementing a simulated version and getting positive timing results. As a final note it can be said that the FMSF demonstrates the proposed thesis.

Appendix A

Test Code

A.1 Single-thread test

As we can see, the thread acquires the current absolute time before and after each base scheduler call. The thread calls `reschedule()`, `prepareToLock()`, synchronizes on an object, calls `rescheduleLock()`, `prepareToUnlock()`, exits the `synchronized` region, calls `rescheduleUnlock()`, `prepareToSuspend(SLEEP)`, calls `RealtimeThread.sleep()`, then `rescheduleResume()`, `prepareToSuspend(WFNP)` and finally waits for its next period. The prepareToSuspend(`SLEEP`) call was selected as representative of all the `prepareToSuspend` calls, except `prepareToSuspend(WFNP)`. This latter one is also representative of `prepareToSuspend(THREAD_END)`, as the two execute the same code. At the start of each period (except the first), right before the end of the loop, the thread calculates the actual amount of time spent in each call and stores this in the `RelativeTime diffx` variables. Then it separately adds the milliseconds and microseconds of each measurement to two ranges of variables, `millisx` and `uicrosx`, $1 \leq x \leq 10$, which keep the

total sum for all the iterations of the `for` loop, for each of the base scheduler calls. To mark the end of each iteration the thread prints a dot. At the end, the thread prints the results, which are the `millisx` and `uicrosx` values divided by the number of iterations, in this case 1000.

```
public class TestThreadLogic implements Runnable {
  final static int numOfIter = 1000;
  ...
  public void run() {
    AbsoluteTime start = new AbsoluteTime(),
                 afterRes = new AbsoluteTime(),
                 afterPTL = new AbsoluteTime(),
                 beforeResL = new AbsoluteTime(),
                 afterResL = new AbsoluteTime(),
                 afterPTU = new AbsoluteTime(),
                 beforeResU = new AbsoluteTime(),
                 afterResU = new AbsoluteTime(),
                 afterPTSSleep = new AbsoluteTime(),
                 beforResResume = new AbsoluteTime(),
                 afterResResume = new AbsoluteTime(),
                 afterPTSWFNP = new AbsoluteTime();
RelativeTime diff1 = new RelativeTime(),
             diff2 = new RelativeTime(),
             diff3 = new RelativeTime(),
             diff4 = new RelativeTime(),
             diff5 = new RelativeTime(),
             diff6 = new RelativeTime(),
             diff7 = new RelativeTime(),
             diff8 = new RelativeTime(),
             zero = new RelativeTime();
int i,uicros1 = 0,
      uicros2 = 0,
      uicros3 = 0,
      uicros4 = 0,
      uicros5 = 0,
      uicros6 = 0,
      uicros7 = 0,
      uicros8 = 0;
long millis1 = 0,
     millis2 = 0,
     millis3 = 0,
     millis4 = 0,
     millis5 = 0,
     millis6 = 0,
     millis7 = 0,
     millis8 = 0;

for (i=0; i<numOfIter; i++) {
  start = Clock.getRealtimeClock().getTime(start);
  PreemptionLevelPriorityScheduler.reschedule(RealtimeThread.
                                    currentRealtimeThread());
  afterRes = Clock.getRealtimeClock().getTime(afterRes);

  PreemptionLevelPriorityScheduler.prepareToLock(
            RealtimeThread.currentRealtimeThread(),obj);
```

```
afterPTL = Clock.getRealtimeClock().getTime(afterPTL);
synchronized(obj) {
  beforeResL = Clock.getRealtimeClock().getTime(beforeResL);
  PreemptionLevelPrioritySchedulerLock(
                  RealtimeThread.currentRealtimeThread());
  afterResL = Clock.getRealtimeClock().getTime(afterResL);

  PreemptionLevelPrioritySchedulerUnlock(
                RealtimeThread.currentRealtimeThread());
  afterPTU = Clock.getRealtimeClock().getTime(afterPTU);
}
beforeResU = Clock.getRealtimeClock().getTime(beforeResU);
PreemptionLevelPrioritySchedulerUnlock(
              RealtimeThread.currentRealtimeThread(),obj);
afterResU = Clock.getRealtimeClock().getTime(afterResU);

PreemptionLevelPrioritySchedulerSuspend(
                   RealtimeThread.currentRealtimeThread(),
                   PreemptionLevelPrioritySchedulerSLEEP);
afterPTSSleep = Clock.getRealtimeClock().getTime(afterPTSSleep);
try {
  RealtimeThread.sleep(zero);
} catch(InterruptedException ie) {
  System.out.println("InterruptedException thrown!");
}
beforResResume = Clock.getRealtimeClock().getTime(beforResResume);
PreemptionLevelPrioritySchedulerRescheduleResume(RealtimeThread.
                                        currentRealtimeThread());
afterResResume = Clock.getRealtimeClock().getTime(afterResResume);

PreemptionLevelPrioritySchedulerSuspend(
          RealtimeThread.currentRealtimeThread(),
          PreemptionLevelPrioritySchedulerWAIT_FOR_NEXT_RELEASE);
afterPTSWFNP = Clock.getRealtimeClock().getTime(afterPTSWFNP);
if (!RealtimeThread.waitForNextPeriod())
  System.out.println("fmsfThread: Deadline overrun at "+Clock.
                                getRealtimeClock().getTime());

diff1 = afterRes.subtract(start, diff1);
diff2 = afterPTL.subtract(afterRes, diff2);
diff3 = afterResL.subtract(beforeResL, diff3);
diff4 = afterPTU.subtract(afterResL, diff4);
diff5 = afterResU.subtract(beforeResU, diff5);
diff6 = afterPTSSleep.subtract(afterResU, diff6);
diff7 = afterResResume.subtract(beforResResume, diff7);
diff8 = afterPTSWFNP.subtract(afterResResume, diff8);
millis1 += diff1.getMilliseconds();
millis2 += diff2.getMilliseconds();
millis3 += diff3.getMilliseconds();
millis4 += diff4.getMilliseconds();
millis5 += diff5.getMilliseconds();
millis6 += diff6.getMilliseconds();
millis7 += diff7.getMilliseconds();
millis8 += diff8.getMilliseconds();
uicros1 += diff1.getNanoseconds()/1000;
uicros2 += diff2.getNanoseconds()/1000;
uicros3 += diff3.getNanoseconds()/1000;
```

```
        uicros4 += diff4.getNanoseconds()/1000;
        uicros5 += diff5.getNanoseconds()/1000;
        uicros6 += diff6.getNanoseconds()/1000;
        uicros7 += diff7.getNanoseconds()/1000;
        uicros8 += diff8.getNanoseconds()/1000;
        System.out.print('.');
    }

    System.out.println("\nResults for "+numOfIter+" iterations");
    System.out.println("_____");
    System.out.println("Mean execution cost for reschedule():
                "+(millis1/numOfIter)+"ms
                "+((float)uicros1/numOfIter)+"us");
    System.out.println("Mean execution cost for prepareToLock():
                "+(millis2/numOfIter)+"ms
                "+((float)uicros2/numOfIter)+"us");
    System.out.println("Mean execution cost for rescheduleLock():
                "+(millis3/numOfIter)+"ms
                "+((float)uicros3/numOfIter)+"us");
    System.out.println("Mean execution cost for prepareToUnlock():
                "+(millis4/numOfIter)+"ms
                "+((float)uicros4/numOfIter)+"us");
    System.out.println("Mean execution cost for rescheduleUnlock():
                "+(millis5/numOfIter)+"ms
                "+((float)uicros5/numOfIter)+"us");
    System.out.println("Mean execution cost for prepareToSuspend(SLEEP):
                "+(millis6/numOfIter)+"ms
                "+((float)uicros6/numOfIter)+"us");
    System.out.println("Mean execution cost for rescheduleResume():
                "+(millis7/numOfIter)+"ms
                "+((float)uicros7/numOfIter)+"us");
    System.out.println("Mean execution cost for prepareToSuspend(WFNP):
                "+(millis8/numOfIter)+"ms
                "+((float)uicros8/numOfIter)+"us");

  }
  ...
}
```

A.2 Multi-thread test

Due to the fact that in this test multiple threads are running, the execution time has to be calculated within the base scheduler calls themselves. Therefore, each of the methods in the `PreemptionLevelPriorityScheduler` has been changed to accept one more parameter of type `RelativeTime`, which is returned by the method to contain the method's execution time. This has been explained in Section 6.4.3.

In this case the timing results taken from all threads produce the final estimation of the mean execution time for each base scheduler call. This

means that the final calculations cannot be made inside a test thread but are done within the `main()` method. As we can see, `main()` is also contained in the `MultiTestThreadLogic` class, therefore it can access the same class variables as the `run()` method of each test thread. In order to ensure that `main()` will calculate the results only after all test threads have terminated, it waits on the `MultiTestThreadLogic` class monitor. Before terminating, each test thread synchronizes on the same monitor and increases the counter `threadsFinished`. When the last test thread terminates the number of finished test threads is the same as the total number of test threads, and the thread notifies the main thread, which then produces the final estimation.

```
public class MultiTestThreadLogic implements Runnable {
  final static int numOfThreads = 10;
  final static int numOfIter = 1000;
  static int threadsFinished = 0;
  static float resUicros = 0,
               pTLUicros = 0,
               resLUicros = 0,
               pTUUicros = 0,
               resUUicros = 0,
               pTSSleepUicros = 0,
               resResSleepUicros = 0,
               pTSWFNPUicros = 0;
  static long resMillis = 0,
              pTLMillis = 0,
              resLMillis = 0,
              pTUMillis = 0,
              resUMillis = 0,
              pTSSleepMillis = 0,
              resResSleepMillis = 0,
              pTSWFNPMillis = 0;

public void run() {
  RelativeTime resch = new RelativeTime(),
               pTL = new RelativeTime(),
               resL = new RelativeTime(),
               pTU = new RelativeTime(),
               resU = new RelativeTime(),
               pTSSleep = new RelativeTime(),
               resResSleep = new RelativeTime(),
               pTSWFNP = new RelativeTime(),
               zero = new RelativeTime();
int i,uicros1 = 0,
      uicros2 = 0,
      uicros3 = 0,
      uicros4 = 0,
      uicros5 = 0,
      uicros6 = 0,
      uicros7 = 0,
      uicros8 = 0;
long millis1 = 0,
     millis2 = 0,
```

```
        millis3 = 0,
        millis4 = 0,
        millis5 = 0,
        millis6 = 0,
        millis7 = 0,
        millis8 = 0;

for (i=0; i<numOfIter; i++) {
  resch = PreemptionLevelPriorityScheduler.reschedule(RealtimeThread.
                                                      currentRealtimeThread(),resch)
                                                      ;
pTL         =         PreemptionLevelPriorityScheduler.prepareToLock(
                                                      RealtimeThread.currentRealtimeT
                                                      hread(),obj,pTL);
synchronized(obj) {
  resL   =   PreemptionLevelPriorityScheduler.rescheduleLock(Realtime
                        Thread.currentRealtimeThread(),resL);

  pTU   =   PreemptionLevelPriorityScheduler.prepareToUnlock(Realtime
                        Thread.currentRealtimeThread(),pTU);
}
resU   =   PreemptionLevelPriorityScheduler.rescheduleUnlock(Realtime
                        Thread.currentRealtimeThread(),obj,resU);

pTSSleep      =      PreemptionLevelPriorityScheduler.prepareToSuspend(
                        RealtimeThread.currentRealtimeThread(),
                        PreemptionLevelPriorityScheduler.SLEEP,pTSSlee
                        p);
try {
  RealtimeThread.sleep(zero);
} catch(InterruptedException ie) {
  System.out.println("InterruptedException thrown!");
}
resResSleep    =    PreemptionLevelPriorityScheduler.rescheduleResume(
      RealtimeThread.currentRealtimeThread(),resResSleep);

pTSWFNP      =      PreemptionLevelPriorityScheduler.prepareToSuspend(
      RealtimeThread.currentRealtimeThread(),
      PreemptionLevelPriorityScheduler.WAIT_FOR_NEXT_RELEASE,pTSWFN
      P);
if (!RealtimeThread.waitForNextPeriod())

        System.out.println(RealtimeThread.currentRealtimeThread().to
        String()         +"         Deadline         overrun         at
        "+Clock.getRealtimeClock().getTime());

        millis1 += resch.getMilliseconds();
        millis2 += pTL.getMilliseconds();
        millis3 += resL.getMilliseconds();
        millis4 += pTU.getMilliseconds();
        millis5 += resU.getMilliseconds();
        millis6 += pTSSleep.getMilliseconds();
        millis7 += resResSleep.getMilliseconds();
        millis8 += pTSWFNP.getMilliseconds();
        uicros1 += resch.getNanoseconds()/1000;
        uicros2 += pTL.getNanoseconds()/1000;
```

```
    uicros3 += resL.getNanoseconds()/1000;
    uicros4 += pTU.getNanoseconds()/1000;
    uicros5 += resU.getNanoseconds()/1000;
    uicros6 += pTSSleep.getNanoseconds()/1000;
    uicros7 += resResSleep.getNanoseconds()/1000;
    uicros8 += pTSWFNP.getNanoseconds()/1000;
    System.out.print('.');
  }

  synchronized(this.getClass()) {
    resMillis += millis1/numOfIter;
    pTLMillis += millis2/numOfIter;
    resLMillis += millis3/numOfIter;
    pTUMillis += millis4/numOfIter;
    resUMillis += millis5/numOfIter;
    pTSSleepMillis += millis6/numOfIter;
    resResSleepMillis += millis7/numOfIter;
    pTSWFNPMillis += millis8/numOfIter;

    resUicros += (float)uicros1/numOfIter;
    pTLUicros += (float)uicros2/numOfIter;
    resLUicros += (float)uicros3/numOfIter;
    pTUUicros += (float)uicros4/numOfIter;
    resUUicros += (float)uicros5/numOfIter;
    pTSSleepUicros += (float)uicros6/numOfIter;
    resResSleepUicros += (float)uicros7/numOfIter;
    pTSWFNPUicros += (float)uicros8/numOfIter;

    threadsFinished++;
      if (threadsFinished==numOfThreads)
        this.getClass().notifyAll();
    }
  }

  public static void main(String[] args) {
    ...
    Thread1.start();
    ...

    synchronized(threadLogic1.getClass()) {
      try {
        threadLogic1.getClass().wait();
      } catch(InterruptedException ie) {
      System.out.println("InterruptedException thrown!");}
    }

    System.out.println("\nResults for "+numOfThreads+" threads,
                       "+numOfIter+" iterations each");
    System.out.println("_____");
    System.out.println("Mean execution cost for reschedule():
                       "+(resMillis/threadsFinished)+"ms
                       "+(resUicros/threadsFinished)+"us");
    System.out.println("Mean execution cost for prepareToLock():
                       "+(pTLMillis/threadsFinished)+"ms
                       "+(pTLUicros/threadsFinished)+"us");
```

```
System.out.println("Mean execution cost for rescheduleLock():
                    "+(resLMillis/threadsFinished)+"ms
                    "+(resLUicros/threadsFinished)+"us");
System.out.println("Mean execution cost for prepareToUnlock():
                    "+(pTUMillis/threadsFinished)+"ms
                    "+(pTUUicros/threadsFinished)+"us");
System.out.println("Mean execution cost for rescheduleUnlock():
                    "+(resUMillis/threadsFinished)+"ms
                    "+(resUUicros/threadsFinished)+"us");
System.out.println("Mean execution cost for prepareToSuspend(SLEEP):
                    "+(pTSSleepMillis/threadsFinished)+"ms
                    "+(pTSSleepUicros/threadsFinished)+"us");
   System.out.println("Mean execution cost for rescheduleResume():
                       "+(resResSleepMillis/threadsFinished)+"ms
                       "+(resResSleepUicros/threadsFinished)+"us");
   System.out.println("Mean execution cost for prepareToSuspend(WFNP):
                       "+(pTSWFNPMillis/threadsFinished)+"ms
                       "+(pTSWFNPUicros/threadsFinished)+"us");

  }
}
```

A.3 Tests on RTSJ methods and operations

This test measures the execution cost of starting a thread, locking a resource, unlocking, calling `sleep()` and calling `waitForNextPeriod()` in a normal RTSJ `RealtimeThread`. As with the previous tests, this test is comprised of a single `Runnable` class, whose `main()` method constructs a thread with this class as its logic. Measuring the cost of starting the thread is done by acquiring the current time in `main()`, right before calling `start()`, and then again as the first thing within the `run()` method. The `run()` method has access to the same variables as `main()`, since they belong to the same class. Therefore, by calculating the difference between the two times we get an estimate of the cost of starting a thread. The other measurements are made, as usual, by using a `getTime()` before and after the code in question. `waitForNextPeriod()` in particular is measured in the second period, so that the thread is already started when the measurement is made. The current time is taken when `waitForNextPeriod()` unblocks to start the second period and taken again when it unblocks at the start of the third period. The cost of the method is approximated by subtracting the thread's period from the difference of the two times. The test code is shown below.

```
public class RTLogic implements Runnable {
  private Object res = new Object();
  static AbsoluteTime start = new AbsoluteTime();
  AbsoluteTime released = new AbsoluteTime(),
```

```
                beforeSync = new AbsoluteTime(),
                afterSync = new AbsoluteTime(),
                beforeUnSync = new AbsoluteTime(),
                afterUnSync = new AbsoluteTime();

RelativeTime diff1 = new RelativeTime(),
             diff2 = new RelativeTime(),
             diff3 = new RelativeTime();
int uicros1 = 0,
    uicros2 = 0,
    uicros3 = 0;
long millis1 = 0,
     millis2 = 0,
     millis3 = 0;

public void run() {
  released = Clock.getRealtimeClock().getTime(released);

  beforeGetTime = Clock.getRealtimeClock().getTime(beforeGetTime);
  foo = Clock.getRealtimeClock().getTime(foo);
  afterGetTime = Clock.getRealtimeClock().getTime(afterGetTime);

  beforeSync = Clock.getRealtimeClock().getTime(beforeSync);
  synchronized(res) {
    afterSync = Clock.getRealtimeClock().getTime(afterSync);
    beforeUnSync = Clock.getRealtimeClock().getTime(beforeUnSync);
  }
  afterUnSync = Clock.getRealtimeClock().getTime(afterUnSync);

  try {
    beforeSleep = Clock.getRealtimeClock().getTime(beforeSleep);
    RealtimeThread.sleep(new RelativeTime());
    afterSleep = Clock.getRealtimeClock().getTime(afterSleep);
  } catch(InterruptedException ie) {
      System.out.println("InterruptedException thrown!");
  }

  if (!RealtimeThread.waitForNextPeriod())
    System.out.println("rtThread: Deadline overrun!");

  beforeWFNP = Clock.getRealtimeClock().getTime(beforeWFNP);
  if (!RealtimeThread.waitForNextPeriod())
    System.out.println("rtThread: Deadline overrun!");
  afterWFNP = Clock.getRealtimeClock().getTime(afterWFNP);

  diff1 = released.subtract(start, diff1);
  diff2 = afterSync.subtract(beforeSync, diff2);
  diff3 = afterUnSync.subtract(beforeUnSync, diff3);
  diff4 = afterSleep.subtract(beforeSleep, diff4);
  diff5 = (afterWFNP.subtract(beforeWFNP)).subtract(period, diff5);

  millis1 += diff1.getMilliseconds();
  millis2 += diff2.getMilliseconds();
  millis3 += diff3.getMilliseconds();
  millis4 += diff4.getMilliseconds();
  millis5 += diff5.getMilliseconds();
```

```
uicros1 += diff1.getNanoseconds()/1000;
uicros2 += diff2.getNanoseconds()/1000;
uicros3 += diff3.getNanoseconds()/1000;
uicros4 += diff4.getNanoseconds()/1000;
uicros5 += diff5.getNanoseconds()/1000;

System.out.println("\nResults");
System.out.println("_____");
System.out.println("Execution cost for starting:  "+(millis1)+"ms
                    "+((float)uicros1)+"us");
System.out.println("Execution cost for synching:  "+(millis2)+"ms
                    "+((float)uicros2)+"us");
  System.out.println("Execution cost for unsynching:  "+(millis3)+"ms
                    "+((float)uicros3)+"us");
  System.out.println("Execution cost for sleeping:  "+(millis4)+"ms
                    "+((float)uicros4)+"us");
  System.out.println("Execution cost for WFNP:  "+(millis5)+"ms
                    "+((float)uicros5)+"us");
}

public static void main(String[] args) {
  ...
  RealtimeThread rtThread = new RealtimeThread(pp1, release1, mem,
                            ImmortalMemory.instance(),        null,
                            rtLogic);

  start = Clock.getRealtimeClock().getTime(start);
  rtThread.start();
}
```

A.3.1 Measuring `setPriority()`

```
public class MeasureSetPriority implements Runnable {
  static int minPrio = PriorityScheduler.instance().getMinPriority();
  AbsoluteTime beforeSetPrio = new AbsoluteTime(),
               afterSetPrio = new AbsoluteTime();

  static RelativeTime period = new RelativeTime((long)200,0);

  RelativeTime diff1 = new RelativeTime(),
               diff2 = new RelativeTime();
  int i,uicros1 = 0,
      uicros2 = 0;
 long millis1 = 0,
      millis2 = 0;

 public void run() {
   PriorityParameters  pp  =  (PriorityParameters)(RealtimeThread.current
                              RealtimeThread().getSchedulingParameters(
                              ));

   for (i=0;i<500;i++) {
     beforeSetPrio = Clock.getRealtimeClock().getTime(beforeSetPrio);
```

```
        pp.setPriority(minPrio+1);
        afterSetPrio = Clock.getRealtimeClock().getTime(afterSetPrio);

        diff1 = afterSetPrio.subtract(beforeSetPrio, diff1);

        beforeSetPrio = Clock.getRealtimeClock().getTime(beforeSetPrio);
        pp.setPriority(minPrio);
        afterSetPrio = Clock.getRealtimeClock().getTime(afterSetPrio);

        diff2 = afterSetPrio.subtract(beforeSetPrio, diff2);

        millis1 += diff1.getMilliseconds();
        millis1 += diff2.getMilliseconds();

        uicros1 += (diff1.getNanoseconds()/1000);
        uicros1 += (diff2.getNanoseconds()/1000);

      }

      System.out.println("\nResults");
      System.out.println("_____");
      System.out.println("Execution cost for changing priority:
                          "+(millis1/1000)+"ms
                          "+((float)uicros1/1000)+"us");
    }

    public static void main(String[] args) {
      PriorityParameters pp1 = new PriorityParameters(minPrio);
      PeriodicParameters release1 = new PeriodicParameters(
                          new RelativeTime((long)100,0),          //start now
                          period,                                 //period
                          null,                                   //cost
                          null,                                   //deadline
                          null, null);                            //handlers
      MemoryParameters mem = new MemoryParameters(MemoryParameters.NO_MAX,
                                                  MemoryParameters.NO_MAX);
      MeasureSetPriority logic = new MeasureSetPriority();

      RealtimeThread rtThread = new RealtimeThread(pp1,
                                                   release1,
                                                   mem,
                                                   ImmortalMemory.instance(),
                                                   null,
                                                   logic);

      rtThread.start();
    }
  }
```

Table of Symbols

τ *or* τ_i	a task or the i^{th} task
$\pi(\tau)$	relative preemption level of τ
$\pi(\tau,t)$	relative preemption level of τ at time t
$\overline{\pi}$	system ceiling
t *or* t_i	a time instance or the i^{th} time instance
$p(\tau)$	fixed priority of τ
r *or* r_i	a resource or the i^{th} resource
$\lceil r \rceil$	preemption ceiling of r
$e(\tau)$	eligibility of τ
$e_f(\tau)$	effective eligibility of τ
$e_a(\tau)$	active eligibility of τ
c_i *or* C_i	execution cost of τ_i
D_i	relative deadline of τ_i
d_i	absolute deadline of τ_i
$\overrightarrow{rel(\tau)}$	*release* operation for τ
$\overrightarrow{loc(\tau, r)}$	*lock* operation of τ on r
$\overrightarrow{unL(\tau, r)}$	*unlock* operation of τ on r
$\overrightarrow{wt(\tau, r, cv)}$	*wait* operation of τ on condition variable cv while locking r
$\overrightarrow{sus(\tau)}$	*suspension* operation for τ
$\overrightarrow{yld(\tau)}$	*yield* operation for τ
$\overrightarrow{chg(\tau)}$	*change* operation for τ
$\overrightarrow{end(\tau)}$	*end* operation of τ
pbsc	preceding base scheduler call
sbsc	succeeding base scheduler call
B_i	scheduling band with *low* priority i
$\lvert\pi\rvert(\tau)$	absolute preemption level of τ
$\Pi(B)$	range of absolute preemption levels assigned to band B
B_r^{loc}	band for which $\lceil r \rceil \in \Pi(B_r^{loc})$
B_τ	τ's own band
B_τ^{cur}	band in which τ currently executes
$S(B)$	application scheduler of band B

BS	base scheduler
$p_L(B)$	*low* priority of band *B*
$p_{ML}(B)$	*medium_lock* priority of band *B*
$p_M(B)$	*medium* priority of band *B*
$p_H(B)$	*high* priority of band *B*

References

[Ada-Europe 2007] Ada-Europe. 2007. Ada Reference Manual – ISO/IEC 8652:2007(E) with Technical Corrigendum 1 and Amendment 1. Available at: http://www.adaic.org/standards/05rm/RM-Final.pdf

[Aldea and Harbour 2001] Aldea Rivas M, Gonzalez Harbour M. 2001. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In International Conference on Reliable Software Technologies, Ada-Europe, May 2001.

[Aldea and Harbour 2002] Aldea Rivas M, Gonzalez Harbour M. 2002. POSIX-compatible application-defined scheduling in MaRTE OS. In Proceedings 14th Euromicro Conference on Real-Time Systems, pp.67 – 75.

[Aldea and Harbour 2003] Aldea Rivas M, Harbour M G. 2003. Application-defined scheduling in Ada. In Proceedings 12th International Workshop on Real-Time Ada (IRTAW '03), ACM Press, New York, NY, pp.42-51.

[Aldea and Harbour 2004] Aldea Rivas M, González Harbour M. 2004. Proposal for Application-Defined Scheduling in POSIX. Flexible Real-time Systems Technology (FIRST) European Project (IST-2001 34140), Deliverable EX.1v2, May 2004.

[Aldea and Harbour 2004b] Aldea Rivas M, González Harbour M. 2004. A New Generalized Approach to Application-Defined Scheduling. 16th Euromicro Conference on Real-Time Systems (Work in progress session), Catania, Sicily (Italy), July 2004.

[Aldea et al. 2004] Aldea M, Miranda J, González Harbour M. 2004. Implementing an Application-Defined Scheduling Framework for Ada Tasking. Lecture Notes in Computer Science, Volume 3063, Springer-Verlag, pp.283-296.

[Aldea et al. 2006] Aldea M, Bernat G, Broster I, Burns A, Dobrin R, Drake J M, Fohler G, Gai P, González Harbour M, Guidi G, Gutiérrez J J, Lennvall T, Lipari G, Martínez J M, Medina J L, Palencia J C, Trimarchi M. 2006. FSF: A Real-Time Scheduling Architecture Framework. In Proceedings 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp.113-124.

[Anderson and Jensen 2006] Anderson J S, Jensen E D. 2006. Distributed Real-Time Specification for Java: A Status Report (Digest). In Proceedings 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '06), ACM Press, pp.3-9.

[Aswathanarayana et al. 2005] Aswathanarayana T, Niehaus D, Subramonian V, Gill C. 2005. Design and Performance of Configurable Endsystem Scheduling Mechanisms. In Proceedings 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05), pp. 32-43.

[Baker 1991] Baker T P. 1991. Stack-Based Scheduling of Real-Time Processes. Real-Time Systems Journal, Vol.3, Issue 1, pp.57-99.

[Behrmann et al. 2004] Behrmann G, David A, Larsen K G. 2004. A Tutorial on UPPAAL, 4th International School on Formal Methods (SFM-RT 2004), LNCS 3185, pp. 200--236.

[Belliardi et al. 2006] Belliardi R, Brosgol B, Dibble P, Holmes D, Wellings A J. 2006. The Real-Time Specification for Java, version 1.0.2, 2006-May-16. Dibble P, editor. Available at: www.rtsj.org

[Bengtsson and Yi 2004] Bengtsson J, Yi W. 2004. Timed Automata: Semantics, Algorithms and Tools, in Lecture Notes on Concurrency and Petri Nets, LNCS 3098, Springer-Verlag.

[Benveniste et al. 1994] Benveniste A, Berry G, Caspi P, Couronn'e P, Dupont F, Gauthier T, Halbwachs N, Le Guernic P, Le Maire C, Mignard F, Paris J P, Sorel Y. 1994. Synchronous technology for real-time systems. In Proceedings of Real-Time Systems, Paris, January 1994. North-Holland.

[Bollella et al. 2000] Bollella G, Brosgol B, Dibble P, Furr S, Gosling J, Hardin D, Turnbull M. 2000. The Real-Time Specification for Java. Addison-Wesley.

[Brandt et al. 1998] Brandt S., Nutt G, Berk T, Mankovich J. 1998. A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage. In Proceedings 19th IEEE Real-Time Systems Symposium (RTSS'98), p. 307.

[Brandt and Nutt 2002] Brandt S A, Nutt G J. 2002. Flexible Soft Real-Time Processing in Middleware. Real-Time Systems Journal, Volume 22, Issue 1, pp.77-118.

[Brandt et al. 2003] Brandt S A, Banachowski S, Lin C, Bisson T. 2003. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes. In Proceedings 24th IEEE Real-Time Systems Symposium (RTSS), pp.396.

[Burns and Wellings 1997] Burns A, Wellings A J. 1997. Concurrency in Ada, 2nd Edition, Cambridge University Press.

[Burns and Wellings 2001] Burns A, Wellings A J. 2001. Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX. 3rd Edition, Addison Wesley – Pearson Education.

[Burns et al. 2003] Burns A, González Harbour M, Wellings A J. 2003. A Round Robin Scheduling Policy for Ada. Lecture Notes in Computer Science, vol. 2655, pp. 334-343.

[Burns et al. 2004] Burns A, Wellings A J, Taft T S. 2004. Supporting Deadlines and EDF Scheduling in Ada. Lecture Notes in Computer Science, Springer-Verlag, Volume 3063, pp.156-165.

[Buttazzo 2005] Buttazzo G. 2005. Rate Monotonic vs. EDF: Judgment Day. Real-Time Systems Journal, Springer-Verlag, vol. 29, no 1, pp.5-26.

[Candea and Jones 1998] Candea G M, Jones M B. 1998. Vassal: Loadable Scheduler Support for multi-Policy Scheduling. In Proceedings 2nd USENIX Windows NT Symposium, Seattle, Washington, August 1998.

[CDE 2007a] The Computer Desktop Encyclopedia. 2007. Definition for: Middleware. [Internet], [cited 2007-Oct-30]. The Computer Language Company Inc. Available through ZDNet at: http://dictionary.zdnet.com/definition/Middleware.html

[CDE 2007b] The Computer Desktop Encyclopedia. 2007. Definition for: Architecture-neutral. [Internet], [cited 2007-Oct-30]. The Computer Language Company Inc. Available through ZDNet at: http://dictionary.zdnet.com/ definition/architecture+neutral.html

[Chen and Tripathi 1994] Chen C M, Tripathi S K. 1994. Multiprocessor priority ceiling based protocols. Technical Report CSTR-3253, UMICAS-TR-94-42, Dept. of Computer Science, University of Maryland at College Park.

[Chen and Lin 1990] Chen M, Lin K. 1990. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. Real-Time Systems Journal, Vol. 2, No. 4, Nov. 1990, pp.325-346.

[Chen et al. 2004] Chen Y S, Chang L P, Kuo T W. 2004. A Configurable Synchronization Protocol for Real-Time Self-Suspending Processes. In Proceedings Real-Time and Embedded Computer Systems and Applications (RTCSA), Gothenburg Sweden, August 25-27, 2004.

[Corsaro et al. 2001] Corsaro A, Schmidt D C, Cytron R, Gill C. 2001. Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems.

Proceedings of the 3rd International Symposium on Distributed Objects and Applications, pp.289-299, Rome, Italy. OMG.

[Davis and Wellings 1995] Davis R, Wellings A. 1995. Dual priority scheduling. In Proceedings 16th IEEE Real-Time Systems Symposium (RTSS'95), 5-7 Dec 1995, pp.100-109.

[Davis and Burns 2005] Davis R I, Burns A. 2005. Hierarchical Fixed Priority Pre-emptive Scheduling. In Proceedings 26th IEEE International Real-Time Systems Symposium (RTSS'05), pp.389-398.

[Deng et al. 1996] Deng Z, Liu J W, Sun S. 1996. Dynamic Scheduling of Hard Real-Time Applications in Open System Environment. University of Illinois at Urbana-Champaign Technical Report, UIUCDCS-R-96-1981.

[Deng and Liu 1997] Deng Z, Liu J W-S. 1997. Scheduling real-time applications in an open environment. In Proceedings 18th IEEE Real-Time Systems Symposium (RTSS '97), p. 308.

[Dibble and Wellings 2004] Dibble P, Wellings A J. 2004. The Real-Time Specification for Java: Current Status and Future Direction. In Proceedings 7th International Conference on Object-Oriented Real-Time Distributed Computing (ISORC'04), pp.71-77.

[DiPippo et al. 2001] DiPippo L C, Wolfe V F, Esibov L, Bethmangalkar G C, Bethmangalkar R, Johnston R, Thuraisingham B, Mauer J. 2001. Scheduling and Priority Mapping for Static Real-Time Middleware. Real-Time Systems Journal, Volume 20, Issue 2 (Mar. 2001), pp.155-182.

[Eide et al. 2004] Eide E, Stack T, Regehr J, Lepreau J. 2004. Dynamic CPU Management for Real-Time, Middleware-Based Systems. In Proceedings 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04) (May 25 - 28, 2004). IEEE Computer Society, Washington DC, p.286.

[Feizabadi et al. 2003] Feizabadi S, Beebee Jr. W, Ravindran B, Li P, Rinard M. 2003. Utilitiy Accrual Scheduling with Real-Time Java. Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), LNCS Volume 2889/2003, pp.550-563, Springer Verlag.

[FIRST 2005] European Union Project IST-2001 34140. FIRST: Flexible Integrated Real-Time Systems Technology. Final Report, Deliverable D-FR, Produced by SalsArt Research Group, Mälardalen University, Sweden, 2005-Jun-28.

[Ford and Susarla 1996] Ford B, Susarla S. 1996. CPU Inheritance Scheduling. In Proceedings 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96), ACM Press, pp.91-105.

[Frisbie et al. 2004] Frisbie M, Niehaus D, Subramonian V, Gill C. 2004. Group scheduling in systems software. In Proceedings 18th International Parallel and Distributed Processing Symposium, 26-30 April 2004, p.120.

[Gai et al. 2001] Gai P, Abeni L, Giorgi M, Buttazzo G. 2001. A New Kernel Approach for Modular Real-Time systems Development. In Proceedings 13th IEEE Euromicro Conference on Real-Time Systems, June 2001.

[Gai et al. 2001b] Gai P, Lipari G, Di Natale M. 2001. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip. In Proceedings IEEE International Real-Time Systems Symposium (RTSS).

[Gai 2005] Gai P. 2005. Real-time Systems for Embedded Microcontrollers. Presentation slides, 2005, found at http://feanor.sssup.it/~giorgio/slides/rts/stack.pdf

[Ganssle and Barr 2003] Ganssle J, Barr M. 2003. Embedded Systems Dictionary. CMP Books.

[Gill et al. 2002] Gill C, Cytron R, Schmidt D. 2002. Middleware scheduling optimization techniques for distributed real-time and embedded systems. Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), San Diego, CA, USA, pp. 311 – 318.

[Gill et al. 2003] Gill C D, Cytron R K, Schmidt D C. 2003. Multiparadigm scheduling for distributed real-time embedded computing. Proceedings of the IEEE, Volume 91, Issue 1, Jan. 2003, pp.183 – 197.

[Goyal et al. 1996] Goyal P, Guo X, Vin H M. 1996. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In Proceedings 2nd Symposium on Operating Systems Design and Implementation, pp. 107-121.

[Gwinn 2004] Gwinn J. 2004. Realtime POSIX Status. [posted 2004-2-20; cited 2007-8-30] Available at: http://www.opengroup.org/rtforum/oa_rtes/uploads/40/4626/Gwinn_POSIX_Status_February_4,_2004.pdf

[Hänninen et al. 2006] Hänninen K, Mäki-Turja J, Bohlin M, Carlson J, Nolin M. 2005. Analysing Stack Usage in Preemptive Shared Stack Systems. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, 2006, found at http://www.mrtc.mdh.se/publications/1136.pdf

[Huang et al. 1992] Huang J, Stankovic J A, Ramamritham K, Towsley D, Purimetla B. 1992. On Using Priority Inheritance in Real-Time Databases. Special Issue of Real-Time Systems Journal, Vol. 4. No. 3, September 1992. Available at: http://citeseer.ist.psu.edu/huang92using.html

[IEEE 2004] IEEE and The Open Group. 2004. Standard 1003.1 – The Single UNIX Specification Version 3, 2004 Edition. Available at: http://www.unix.org/single_unix_specification/

[Jemander 2005] Jemander T. 2005. The need for configurable and flexible scheduling in a RTOS aspiring to solve contemporary problems. In Proceedings 1st International Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT).

[Jiang and Cheng 2001] Jiang F, Cheng A M K. 2001. A Context Switch Reduction Technique for Real-Time Task Synchronization. In Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS'01), p.10031a.

[Jones 1997] Jones M B. 1997. What really happened on Mars Rover Pathfinder. The Risks Digest, Vol 19, Issue 49, 1997/12/9, Neumann P G, moderator. Available at http://catless.ncl.ac.uk/Risks/19.49.html

[Jones and Regehr 1999] Jones M B, Regehr J. 1999. CPU Reservations and Time Constraints: Implementation Experience on Windows NT. In Proceedings 3rd USENIX Windows NT Symposium, pp. 93-102.

[Kalogeraki et al. 2000] Kalogeraki V, Melliar-Smith P M, Moser L E. 2000. Dynamic Scheduling for Soft Real-Time Distributed Object Systems. In Proceedings of the 3rd IEEE international Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), March 15 - 17, 2000, p.114.

[Kaneko et al. 1996] Kaneko H, Stankovic J A, Sen S, Ramamritham K. 1996. Integrated scheduling of multimedia and hard real-time tasks. In Proceedings 17th IEEE Real-Time Systems Symposium (RTSS '96), p. 206.

[Kim and Koh 1995] Kim Y, Koh K. 1995. Pessimistic Deadline Ceiling Protocol: a concurrency control protocol under earliest deadline first scheduling. In Proceedings 7th Euromicro Workshop on Real-Time Systems (ECRTS), p.80.

[Kuo and Li 1999] Kuo T, Li C. 1999. A Fixed-Priority-Driven Open Environment for Real-Time Applications. In Proceedings 20th IEEE Real-Time Systems Symposium (RTSS'99), IEEE Computer Society, p.256.

[Lam and Ng 2000] Lam K, Ng J K. 2000. A conditional abortable priority ceiling protocol for scheduling mixed real-time tasks. Journal of Systems Architecture, vol.46, issue 7 (Apr. 2000), pp.573-585, Elsevier North-Holland.

[Lauer and Needham 1979] Lauer H C, Needham R M. 1979. On the Duality of Operating System Structures. SIGOPS Oper. Syst. Rev. 13, 2 (Apr. 1979), 3-19. DOI=http://doi.acm.org/10.1145/850657.850658

[Li et al. 2004] Li P, Ravindran B, Suhaib S, Feizabadi S. 2004. A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems. IEEE Transactions on Software Engineering, vol. 30, issue 9, pp.613-629.

[Lindholm and Yellin 1999] Lindholm T, Yellin F. 1999. The Java Virtual Machine Specification, 2nd Edition. Prentice Hall PTR. Available online as HTML at: https://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

[Liu 2000] Liu J W S. 2000. Real-Time Systems. Prentice-Hall.

[Lipari et al. 2004] Lipari G, Lamastra G, Abeni L. 2004. Task Synchronization in Reservation-Based Real-Time Systems. IEEE Trans. Comput. Vol. 53, Issue 12 (Dec. 2004), pp.1591-1601. Found at: http://dx.doi.org/10.1109/ TC.2004.120

[Locke 2002] Locke D. 2002. Priority inheritance: The Real Story. LinuxDevices.com, July 2002, found at http://www.linuxdevices.com/articles /AT5698775833.html

[Mok 1983] Mok A K-L. 1983. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. Ph.D. Thesis, MIT, Department of Electrical Engineering and Computer Science, MIT/LCS/TR-297, May 1983.

[deNiz et al. 2001] de Niz D, Saowanee Saewong, Rajkumar R, Abeni L. 2001. Resource sharing in reservation-based systems. In Proceedings 7th IEEE Real-Time Technology and Applications Symposium, pp.130 – 131.

[OMG 2005] Object Management Group. 2005. Real-time CORBA Specification. Version 1.2, January 2005. Available at: http://www.omg.org/technology/ documents/formal/real-time_CORBA.htm

[Rajkumar et al. 1988] Rajkumar R, Sha L, Lehoczky J. 1988. Real-Time Synchronization Protocols for Multiprocessors. In Proceedings IEEE Real-Time Systems Symposium (RTSS), pp. 259–269.

[Rajkumar 1991] Rajkumar R. 1991. Synchronization in multiple processor systems. Chapter in "Synchronization in Real-Time Systems: A Priority Inheritance Approach", Kluwer Academic Publishers.

[Real and Wellings 1999] Real J, Wellings A J. 1999. The Ceiling Protocol in Multi-moded Real-Time Systems. Lecture Notes in Computer Science, vol.1622, pp. 275-286, available at http://www.springerlink.com/content/blewf4dlevywk712/fulltext.pdf

[Real and Crespo 2004] Real J, Crespo A. 2004. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. Real-Time Systems

Journal, vol.26, pp.161-197, available at http://www.springerlink.com/content/h15ln8v8313028t3/

[Regehr et al. 2000] Regehr J, Jones M B, Stankovic J A. 2000. Operating System Support for Multimedia: The Programming Model Matters. Technical Report MSR-TR-2000-89. Available at: http://research.microsoft.com/~mbj/papers/tr-2000-89.pdf

[RTSJ 2007] The Real-Time Specification for Java Website. [updated 2007-Jul-15; cited 2007-Jul-26]. Found at: http://www.rtsj.org

[Sha et al. 1990] Sha L, Rajkumar R, Lehoczky J P. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, Vol.39, Iss.9, p.1175-1185, Sep. 1990. available at http://dx.doi.org/10.1109/12.57058

[Sprunt et al. 1989] Sprunt B, Sha L, Lehoczky J. 1989. Aperiodic Task Scheduling for Hard-Real-Time Systems. Real-Time Systems Journal, Vol. 1, Issue 1, pp.27-60.

[Spuri and Stankovic 1994] Spuri M, Stankovic J A. 1994. How to Integrate Precedence Constraints and Shared Resources in Real-Time Scheduling. IEEE Transactions on Computers, vol. 43, no. 12, December 1994, pp.1407-1412.

[Spuri and Buttazzo 1996] Spuri M, Buttazzo G. 1996. Scheduling aperiodic tasks in dynamic priority systems. Volume 10, Issue 2, pp.179-210.

[Squadrito et al. 1996] Squadrito M, DiPippo L C, Wolfe V F. 1996. The Affected Set Priority Ceiling Protocol For Real-time Object-Oriented Databases. Proceedings of the First International Workshop on Real-Time Databases, March 1996.

[Squadrito et al. 1999] Squadrito M, Esibov L, DiPippo L C, Wolfe V F, Cooper G, Thurasingham B, Krupp P, Milligan M, Johnston R, Bethmangalkar R. 1999. The affected set priority ceiling protocols for real-time object-oriented concurrency control. The International Journal of Computer Systems Science and Engineering, vol. 14, issue 4, pp.227-239, July 1999.

[Stankovic et al. 1998] Stankovic J A, Spuri M, Ramamritham K, Buttazzo G. 1998. Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms. Kluwer Academic Publishers.

[Strosnider et al. 1995] Strosnider J K, Lehoczky J P, Sha L. 1995. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. IEEE Transactions on Computers, Volume 44, Issue 1 (Jan. 1995), pp.73-91.

[Sun and Liu 1996] Sun J, Liu J. 1996. Synchronization protocols in distributed real-time systems. In Proceedings 16th International Conference on Distributed Computing Systems, May 1996. Available at: http://citeseer.ist.psu.edu/ sun96synchronization.html

[TimeSys 2007] TimeSys Corporation. 2007. RTSJ Reference Implementation (RI) and Technology Compatibility Kit (TCK) [Internet]. [cited 2007-Oct-22]. Found at: http://www.timesys.com/java/

[Tokuda and Kitayama 1993] Tokuda H, Kitayama T. 1993. Dynamic QoS Control based on Real-Time Threads. 3rd International Workshop on Network and Operating Systems Support for Digital Audio and Video, LNCS Vol. 846, pp.114-123, Springer-Verlag.

[Wang and Lin 1998] Wang Y C, Lin K J. 1998. Enhancing the real-time capability of the Linux kernel. In Proceedings 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98), Hiroshima, Japan, October 1998.

[Wang and Lin 1999] Wang Y C, Lin K J. 1999. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In Proceedings 20th IEEE Real-Time Systems Symposium, pp.246–255.

[Wolfe et al. 1999] Wolfe V F, DiPippo L C, Bethmagalkar R, Cooper G, Johnston R, Kortmann P, Watson B, Wohlever S. 1999. RapidSched: static scheduling and analysis for real-time CORBA. In Proceedings 4th International Workshop on Object-Oriented Real-Time Dependable Systems, pp.34 – 39.

[Yodaiken 1999] Yodaiken V. 1999. The RTLinux Manifesto. In Proceedings 5th Linux Expo, Raleigh, NC, March 1999.

[Yodaiken 2002] Yodaiken V. 2002. Against Priority Inheritance. FSMLabs Technical Report, June 2002, found at http://www.fsmlabs.com/resources/ white_papers/

[Zerzelidis and Wellings 2006] Zerzelidis A, Wellings A J. 2006. Getting More Flexible Scheduling in the RTSJ. In Proceedings 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06), IEEE Computer Society, pp.3-10.

[Zerzelidis and Wellings 2006b] Zerzelidis A, Wellings A J. 2006. Model-based verification of a framework for flexible scheduling in the real-time specification for Java. In Proceedings 4th International Workshop on Java Technologies For Real-Time and Embedded Systems (JTRES'06), vol. 177. ACM Press, pp.20-29.

[Zerzelidis et al. 2007] Zerzelidis A, Burns A, Wellings A J. 2007 Correcting the EDF protocol in Ada 2005. In Proceedings 13th International Real-Time Ada Workshop (IRTAW-13).

[Zhang and Cordes 2002] Zhang C, Cordes D. 2002. A resource synchronization protocol for dynamic scheduling real-time CORBA. In Proceedings IEEE SoutheastCon 2002, pp.15 – 20.

[Zhang et al. 2005] Zhang J, DiPippo L, Fay-Wolfe V, Bryan K, Murphy M. 2005. A real-time distributed scheduling service for middleware systems. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), 2-4 Feb. 2005, pp. 59 – 65.