

Getting More Flexible Scheduling in the RTSJ

Alexandros Zerzelidis and A.J. Wellings, *University of York, U.K., {alex,andy}@cs.york.ac.uk*

Abstract

This paper illustrates how the Real-Time Specification for Java (RTSJ) can be modified to allow applications to implement more flexible scheduling. The proposed approach is a two-level scheduling mechanism where the first level is the RTSJ priority scheduler and the second level is under application control. Minimum, backward-compatible changes to the RTSJ specification are discussed to motivate the required interface. The only assumptions made about the underlying real-time operating system is that it supports pre-emptive priority-based dispatching and that changes to priorities have immediate effect.

1. Introduction

Scheduling is the ordering of thread/process executions so that the underlying hardware resources (processors, networks, etc.) and software resources (shared data objects) are efficiently and predictably used. The Real-Time Specification for Java (RTSJ) [8] provides a framework from within which real-time scheduling can be performed for single-processor systems. The intention is that a range of schedulers should be supportable, with all schedulers conforming to the abstract `Scheduler` class. However, the current specification defines only a base scheduler, the `PriorityScheduler`. The scheduling framework can be summarized as follows:

Scheduling Policy: The RTSJ uses the notion of the “execution eligibility” of “schedulable objects” to determine the 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 implements priority inheritance algorithms on resource accesses (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, 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.

Whilst it is clear that the RTSJ’s intention is to support 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.

Section 2 defines a framework within which multiple application-defined schedulers can be implemented. Section 3 then uses this framework to show how an application-defined EDF scheduler can be constructed. Related work is briefly considered in Section 4. Section 5 summarises the changes needed in the RTSJ to support our approach, and the impact on feasibility analysis. Finally, the conclusions are given.

The remainder of this paper assumes that priority changes that require OS intervention occur immediately and are not deferred. The terms “thread” and “schedulable object” are used interchangeably.

2. Dynamic Priorities and Flexibility

The general **scheduling mechanism** supported by the RTSJ is undefined. In part, this is due to the variety of execution environments in which an application may execute. Whatever the execution environment, the “write-once carefully, run-anywhere conditionally” goal dictates that the RTSJ should define its scheduling mechanism. Most real-time operating systems support pre-emptive priority-based dispatching. Consequently, this paper argues that the RTSJ should define this as the base scheduling mechanism. However, many modern applications require more flexible scheduling [3], [14]. 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 OSs nowadays support hierarchical scheduling [10].

In this paper we propose that the RTSJ should support hierarchical scheduling within a fixed-priority framework. More specifically, we present a two-level scheduling scheme, with the RTSJ’s priority scheduler at the top level. Under this scheme an application can implement its own scheduler, which may have its own notion of execution eligibility, and request a band of

priorities. The application-defined scheduler can direct the execution of threads within the requested band by manipulating their priorities. Adopting this approach also allows multiple schedulers to 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.

2.1. The Proposed Model

Currently the `Scheduler` class is defined as follows:

```
package javax.realtime;
public abstract class Scheduler {
    protected Scheduler();
    protected abstract boolean addToFeasibility(
        Schedulable sched);
    protected abstract boolean removeFromFeasibility(
        Schedulable sched);
    public abstract boolean isFeasible();
    public abstract boolean setIfFeasible(
        Schedulable sched, ReleaseParameters rel,
        MemoryParameters mem);
    public static Scheduler getDefaultScheduler();
    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. 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 are carried out under the hood. This was to allow greater flexibility to an RTSJ implementation that would want to support other schedulers¹. However, this now means that, in order to expose the underlying mechanisms, a radical overhaul of the RTSJ scheduling API would be required.

The approach taken here is different. In order to keep changes to the API as small as possible, we keep the scheduling mechanism invisible to applications, relying instead on the priority-based dispatching to carry out application-defined scheduling policy decisions. An application-defined scheduler is assigned *four* priority queues of the `PriorityScheduler`. We name these H(high), M(medium), ML(medium-lock), and L(low). This set of priority queues is called a *scheduling band*. These priorities are to be used in the following manner:

- When schedulable objects are released (or become unblocked), they are to be released at the *high* priority level. This priority is where all scheduling decisions need to be carried out.
- The application-defined scheduler keeps track of the thread with the highest execution eligibility. This object has its priority set to the *medium* level.
- Queue L is where all the application scheduler's threads usually reside when they are not running.
- Finally, priority ML is associated with object locking and will be discussed later in the paper.

In the next subsection we will see how the `PriorityScheduler` uses these priorities to enforce application-defined scheduling policy decisions.

2.1.1. The new `PriorityScheduler` class. The basic idea behind this paper is that any scheduling policy can be supported by simply manipulating priorities, assuming we know when the RTSJ library/virtual machine is about to call an OS routine that might potentially block the calling schedulable object and cause a context switch. To facilitate this, four new methods are introduced into the `PriorityScheduler` class. The four methods are:

```
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;
    ...
    // constants for "state" argument
    public static final int LOCKED;
    public static final int UNLOCKED;
    // new methods
    protected static final void prepareToSuspend(
        Schedulable sched, int reason);
    protected static final void prepareToSuspend(
        Schedulable sched, Object lock,
        MonitorControl monitor);
    protected static final void reschedule(
        Schedulable sched, Object lock,
        MonitorControl monitor, int state);
    protected static final void reschedule(
        Schedulable sched);
}
```

The goal is to give control to the base scheduler (`PriorityScheduler`) *just before* the schedulable object calls a potentially suspending OS call and *right after* it returns from such a call². `prepareToSuspend()` precedes the OS call and `reschedule()` comes immediately after that. The RTSJ virtual machine and libraries are modified accordingly. As we can see, there are two variations for each method; the version with the `lock` argument is for the special case of locking an object through a synchronized statement or

¹ Since the semantics for methods like `waitForNextPeriod()` are only defined for the `PriorityScheduler`, other schedulers can support different semantics.

² This assumes that the RTSJ adopts a native thread model and that the OS performs all context switches. If the RTSJ performs its own scheduling, it calls the methods just before and after the context switch code.

method. Locking will be discussed more thoroughly in later sections. The other version of the two methods is for all other potentially suspending situations, as specified by the reason argument, e.g. `WAIT_FOR_NEXT_RELEASE`, `SLEEP`, `IO_WAIT`, etc. `prepareToSuspend()` is called before a thread executes a potentially blocking operation (e.g. after the end of each release). It sets the caller's priority to *high* (ready for the next release), and then asks the application-defined scheduler for the thread with the next highest execution eligibility. It sets the priority of this thread to *medium* and the method returns. The result is that, if the thread blocks, the next eligible thread will automatically execute. If the thread doesn't block, it will immediately call `reschedule()`. `reschedule()` takes care of a thread when it becomes available to run. It compares the execution eligibility of the calling thread with that of its current most eligible thread. If the caller has higher execution eligibility, the previous most eligible thread has its priority set to *low* and the caller has its priority set to *medium*. For example, consider some code in the RTSJ implementation that is about to put a thread to sleep through a POSIX call:

```
sleep(seconds);
```

This would be rewritten as:

```
PriorityScheduler.prepareToSuspend(
    RealtimeThread.currentRealtimeThread(), SLEEP);
sleep(seconds);
PriorityScheduler.reschedule(
    RealtimeThread.currentRealtimeThread());
```

With this code, an initial thread T_1 runs at medium priority M . The `prepareToSuspend()` method raises the priority of the thread to high (H). It then asks the thread's application-defined scheduler for the next most eligible thread (say T_2) in its band. It sets T_2 's priority to M and returns. Following, the POSIX call to `sleep()` is executed and T_1 suspends (while at priority H). Priority-based dispatching will now select the next thread, which is T_2 at priority M . When `seconds` have elapsed, T_1 awakens, preempting T_2 since its priority is still $H > M$. It immediately calls `reschedule()`, which checks which thread has the highest eligibility, sets its priority to M and the other thread's priority to low (L).

The API between the base scheduler and the user-defined schedulers is given in the next subsection.

2.1.2. Application-defined schedulers. To allow application-defined schedulers, a new subclass of `Scheduler` is introduced:

```
package javax.realtime;
public abstract class ApplicationDefinedScheduler
    extends Scheduler {
    public ApplicationDefinedScheduler(int low, int
        medium_low, int medium, int high,
        ProcessingGroupParameters cap,
        int preemptLevels);
    protected abstract void released(
        Schedulable sched, boolean running);
    protected abstract void preempted(
```

```
    Schedulable current, Schedulable newcomer);
    protected abstract void lockedObject(
        Schedulable sched, int objectCeiling);
    protected abstract void unlockedObject(
        Schedulable sched, int objectCeiling);
    protected abstract void suspended(
        Schedulable sched, int reason);
    protected abstract Schedulable getMostEligible();
    protected abstract Schedulable
        compareEligibility(Schedulable sched1,
            Schedulable sched2);
    protected abstract boolean setScheduler(
        Schedulable sched);
    protected static final ApplicationDefinedScheduler
        getScheduler(int priority);
    protected static final ApplicationDefinedScheduler
        getScheduler(Schedulable sched);
    protected static final int getSchedulingBand(
        int priority);
    public static final int getLevelsPerBand();
    public static final void setLevelsPerBand(
        int levels);
    protected static final int calculateAbsolute(
        int band, int level);
    protected static final void setSchedulable(
        Schedulable sched,
        ApplicationDefinedScheduler appScheduler);
}
```

In order to create an application-defined scheduler we must inherit from this class and implement all abstract methods, which form the one-way API between the `PriorityScheduler` and every application scheduler (only the base scheduler can issue calls to other schedulers). Following, we give a description of each abstract method: `released()` is called when a new thread in the scheduler's band has been started; `preempted(Schedulable, Schedulable)` is called when preemption happens within the band; `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); `suspended()` informs the scheduler that a thread has been suspended (in reality, this method is called right before the thread is actually suspended); the `getMostEligible()` method asks from the application scheduler to return its next most eligible thread; `compareEligibility()` asks the scheduler to specify which of the two given threads has greater eligibility according to the scheduler's scheduling policy; finally, `setScheduler()` notifies an application scheduler that a thread has been assigned to it. There are also six static methods, which will be discussed in the next subsection.

As an example of threads running under an application-defined scheduler let us consider the execution of three real-time threads (T_1 , T_2 and T_3) shown in Figure 1. They are released at times t_1 , t_2 , and t_3 respectively (where $t_1 < t_2 < t_3$). T_2 has the highest execution eligibility, followed by T_3 and T_1 . The upper part of Figure 1 shows what priority each thread has at any given point in time, while the lower part shows the resulting thread execution schedule. Note how all

threads execute at high when they are released and during a `prepareToSuspend()` call.

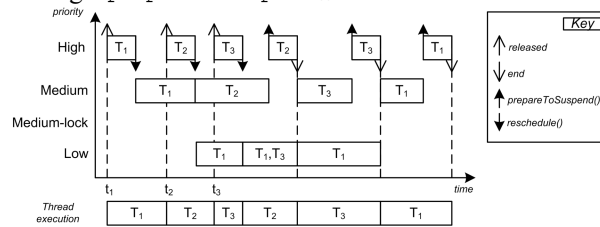


Figure 1: Manipulating Operating System Priorities

With this approach, a thread with lower execution eligibility will execute in preference to a higher execution eligibility thread but only for a limited time when it is released. This is similar to an OS that must allow a thread to be released before deciding what priority it should run at.

2.1.3. Multiple Schedulers. With the above approach multiple user-defined schedulers can coexist in the system, if they are allocated non-overlapping bands in the RTSJ priority range. Hence, the proposal supports two-level scheduling. The first level scheduler is priority-based, the second level is user-defined within a scheduling band. The `ApplicationDefinedScheduler` class, apart from specifying the API for application schedulers, also manages the system's scheduling bands, and for this reason it specifies a static API for the `PriorityScheduler` to use, as seen in the previous subsection. Each application-defined scheduler's constructor must invoke the `ApplicationDefinedScheduler` constructor through `super()`. The arguments passed are: the four priority levels the scheduler wants reserved (low, medium_lock, medium, high); a `ProcessingGroupParameters` object (`cap`) through which each application-defined scheduler is given a processor capacity for the schedulable objects it manages; and the numbers of preemption levels the band is going to need (`preemptLevels`) – this last argument will be discussed in the next subsection. The `ApplicationDefinedScheduler` class keeps all band related information (band-scheduler pairings). If the requested priority levels overlap with previous reservations, the `ApplicationDefinedScheduler` constructor throws an unchecked exception. The static methods are: `getScheduler(int)` returns the `ApplicationDefinedScheduler` assigned to the given priority level; `getScheduler(Schedulable)` returns the `ApplicationDefinedScheduler` which schedules the given thread; `getSchedulingBand(int)` returns the low priority of the band to which the given priority belongs to; `getLevelsPerBand()` and `setLevelsPerBand(int)` manipulate the default number of preemption levels per band; `setSchedulable()` is called by the `setScheduler()` method of each user-

defined scheduler to inform the `ApplicationDefinedScheduler` that a particular thread will be scheduled by the calling scheduler; finally, `calculateAbsolute()` will be discussed later in this paper.

2.2. Execution Eligibility Inversions

Execution eligibility inversion can occur whenever a schedulable object is blocked waiting for a resource. In order to limit the length 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 lock 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 also 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 (for example, an object lock). The programmer can change the default priority inversion control algorithm for individual objects (or for all objects) via the `MonitorControl` class hierarchy. At the root of this hierarchy is the following abstract class:

```
package javax.realtime;
public abstract class MonitorControl {
    protected MonitorControl();
    public static MonitorControl getMonitorControl();
    public static MonitorControl getMonitorControl(
        Object monitor);
    public static MonitorControl setMonitorControl(
        MonitorControl policy);
    public static MonitorControl setMonitorControl(
        Object monitor, 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`.

2.2.1. Adding preemption levels. The role of priority inversion control algorithms is to bind the time during which a higher priority schedulable object can be blocked (by a lower priority schedulable object) when trying to access a shared resource. Baker [2] used the concept of **preemption levels** to introduce *execution eligibility inversion control* to scheduling algorithms with different notions of execution eligibility, other than a fixed priority. With preemption levels, a schedulable object can only preempt another object if it has a higher preemption level. Each schedulable object

is assigned a preemption level according to the following rule: if a schedulable object has higher execution eligibility than another, but arrives later than the other, then it must have a higher preemption level than the other. This is to say that, in situations where no locking takes place, preemption levels concur with priorities as to which schedulable should run next. Each shared resource is given a ceiling preemption level, which, for single unit resources, is the highest preemption level of all the schedulable objects accessing the resource. Based on these definitions, Baker defined an execution eligibility inversion avoidance protocol, known as Stack Resource Policy (SRP), which states that a schedulable object can start execution only if it has the highest execution eligibility and its preemption level is higher than the ceiling of each locked resource. To make this check easier, the notion of the system ceiling is introduced, which is the highest ceiling amongst the locked resources, so that a thread's preemption level need only be higher than the system ceiling. This check ensures that once a thread starts its execution it cannot block on a lock³.

Preemption levels are ideal in helping us control resource sharing between scheduling bands. They can be applied to any scheduling policy, so effectively we can assign them across all bands and have a uniform way of controlling priority inversion throughout the whole range of priorities (for fixed-priority scheduling the preemption level equals the priority). As we have seen, when constructing an application scheduler we assign it a number of preemption levels (this must be at least equal to the number of threads the scheduler will manage). From this range, we assign each thread a *relative preemption level*, which is the level it has within its scheduler. So, for example, if we assign 5 preemption levels to a band then for any thread in that band its relative preemption level must be between 1 and 5. Based on this we can calculate the thread's absolute preemption level using the next formula:

$$apl = \sum_{i=1}^n levels_i + rpl$$

where *apl* is the thread's absolute preemption level, *rpl* is the relative preemption level, *n* is the number of bands below the thread's band, and *levels_i* is the number of preemption levels used by band *i*. In essence, the absolute preemption level is known if we know the pair (*low*, *rpl*), where *low* is the low priority of the thread's band, because we can calculate *n* when we know *low*. This is the calculation that the `calculateAbsolute()` static method in

`ApplicationDefinedScheduler` does, returning the absolute preemption level for a given pair of (*band*, *preemption level*).

We can now define a resource ceiling to be a pair (*low*, *rpl*) such that the absolute preemption level that it yields is the highest amongst the threads accessing the resource. To extend the RTSJ to support preemption levels, the following new classes are introduced⁴:

```
package javax.realtime;
public class PreemptionLevelParameters
    extends PriorityParameters {
    public PreemptionLevelParameters(
        int relativePreemptionLevel);
    public void setPreemptionLevel(
        int relativePreemptionLevel);
    public int getPreemptionLevel();
}

public class StackResourcePolicy
    extends MonitorControl {
    private StackResourcePolicy(int band,
        int ceiling);
    public int getSchedulingBand();
    public int getCeiling();
    public static int getMaxPreemptionLevel();
    public static StackResourcePolicy instance(
        int band, int ceiling);
}
```

Implementation of the priority inversion control algorithm is done at the middleware layer and is transparent to the OS. Every resource, which is accessed by threads running under an application-defined scheduler, should be governed by a `MonitorControl` object of type `StackResourcePolicy`. In light of locking, a thread's current band (the band it is currently executing in) can be either its own (original) band or a higher band. Before a thread enters a synchronized region, `prepareToSuspend(sched, lock, monitor)` is called, raising the thread to the *high* priority of its current band. Because of the SRP, the thread is guaranteed not to block, so no checking is needed. The method just sets the thread's priority to the appropriate level; if the `getSchedulingBand()` method of the `StackResourcePolicy` object, associated with the resource to be locked, returns a higher band than the thread's current band, locking takes place *outside the band* and the thread is moved to the *high* priority of the higher band. If the band returned is the same as the thread's current band, then the thread stays at the *high* priority of the current band. It is an erroneous condition for the method to return a lower band than the locking thread's own band, but it could return a lower band than the thread's current band (i.e. the thread has already locked another resource outside its own band). In this case the system ceiling doesn't

³ Baker's algorithm assumes that threads do not voluntarily suspend themselves whilst holding a lock.

⁴ Note, this is an extension of `PriorityParameters` as the `PriorityScheduler` ultimately schedules each schedulable object. Also, the base priority is not set by the application when creating `PreemptionLevelParameter` objects. It is set by the `PriorityScheduler`.

change when the locking takes place. Next the synchronized call takes place (notice that no matter which band the thread is locking at, its priority when making the synchronized call is *high* for the respective band). Immediately after, `reschedule(sched, lock, monitor, LOCK)` is called, which, depending on whether the thread is on its own or on a higher band, takes the thread to the *medium* or *medium_lock* priority of the band, respectively. When unlocking we call `reschedule(sched, lock, monitor, UNLOCK)`, which raises the thread to the *high* priority of the current band and checks to see if there is a thread available to run in the current band (it could be the case that a thread was released while the calling thread was holding the lock, but couldn't run because of the system ceiling). At this point there are two things to consider: i) "is the current band the thread's own band?", and if not, ii) "is the thread returning to its own band?" If the answer to the first question is yes, then `reschedule()` calls `getMostEligible()`, places the returned thread at the *middle* queue, and, if this thread is different from the calling thread, places the calling thread at the *low* queue. If the answer is no, then `reschedule()` calls `getMostEligible()` and places the returned thread (if any) at the *middle* queue of the current band. Depending, now, on whether the calling thread is returning to its own or to a higher-than-its-own band, it is placed on the *middle* queue of its own band or on the *middle_lock* queue of the band it returns to. Note that the scheme can cope with nested locking. However, there is one condition that needs to hold: **the locking thread must never suspend itself**.⁵

Consider an example of two EDF schedulers (EDF₁ and EDF₂), which have been allocated to priority bands (L₁=1, ML₁=2, M₁=3, H₁=4) and (L₂=7, ML₂=8, M₂=9, H₂=10) respectively, as illustrated in Figure 2 (priorities 5-6 and 11-28 have not been allocated to any application-defined scheduler, so they are scheduled directly by the default priority scheduler).

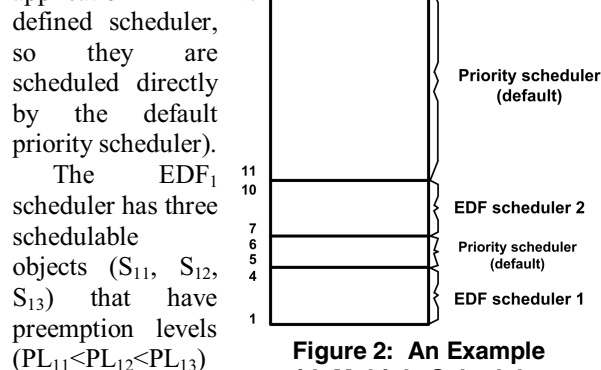


Figure 2: An Example with Multiple Schedulers

⁵ Note that the RTSJ *does* allow this. It is outside this paper's scope to discuss the modifications to our approach needed to cope with this behaviour. As it is, we maintain the system ceiling, when suspension occurs while locking.

respectively. The EDF₂ scheduler has two schedulable objects (S₂₁ and S₂₂) that have preemption levels (PL₂₁<PL₂₂) respectively. S₁₂, S₁₃, S₂₁, and S₂₂ all access a shared Object O. Since the highest absolute preemption level amongst the schedulables accessing O is that of S₂₂, the ceiling of the object will be the pair (L₂, PL₂₂). Now whenever S₁₂ (or S₁₃) accesses the object its priorities have to be raised to ML₂=10. This will ensure that S₁₂ (S₁₃) cannot be preempted by S₁₃ (S₁₂), or by anything running on priority queues 5-8. Now, as S₂₁ and S₂₂ also access the resource, they may become executable whilst S₁₂ (or S₁₃) is accessing the resource. Consider the case where S₂₁ becomes executable at H₂ priority level. It will preempt S₁₂ (S₁₃). The EDF₂ scheduler will move S₂₁ to the L₂ level, since its preemption level will not be higher than the system ceiling, which is (L₂, PL₂₂). L₂ is lower than ML₂, so S₁₂ (S₁₃) will continue to run. When S₁₂ (S₁₃) unlocks the object, it will be elevated to H₂ and query EDF₂ for its most eligible schedulable. This will return S₂₁, which will be put on M₂. S₁₂ (S₁₃) will next be moved to M₁, thus being preempted by S₂₁.

3. An EDF Scheduler

As an example and proof of concept for our user-defined scheduling scheme, we implemented an EDF scheduler.

```
public class EDFScheduler
    extends ApplicationDefinedScheduler {
    public EDFScheduler(
        int low, int medium_lock, int medium,
        int high, ProcessingGroupParameters cap,
        int preemptLevels);
    ...
}
```

The class implements all abstract `Scheduler` and `ApplicationDefinedScheduler` methods, and contains methods for manipulating all internal structures needed to implement the scheduler (EDF queues). The basis for our implementation is the *preemption level protocol* (PLP) by Burns et al. [5]. The preemption level protocol implements the EDF scheduling algorithm on priority queues and is based on the stack resource policy. Preemption levels are assigned according to the relative deadline of each schedulable object (the shorter the deadline, the higher the preemption level) [2]. In our implementation, the EDF scheduler has its own internal priority queues where threads are *logically* placed. The priorities of these queues equal the relative preemption levels used in the EDF band. The PLP protocol is well defined in [5] and we will not repeat it here.

Figure 3 shows the mapping of `EDFScheduler` queues to `PriorityScheduler` queues. Box (a) shows the application code instantiating an `EDFScheduler`. Box (b) shows the created internal EDF queues (there

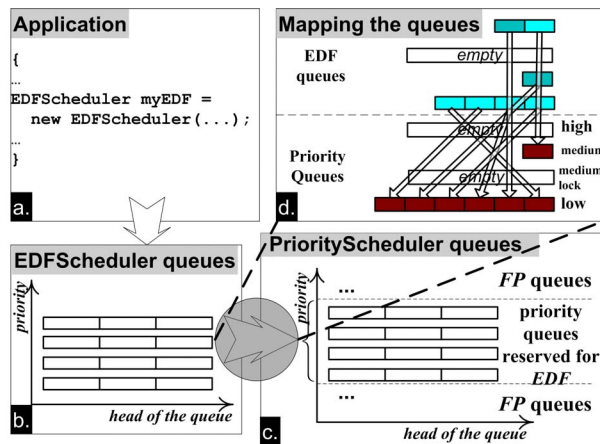


Figure 3: Mapping between EDFScheduler and PriorityScheduler

can be any number of these queues). Box (c) shows the four queues reserved for the `EDFScheduler` in the `PriorityScheduler`. Finally, box (d) shows the mapping between the threads on the priority queues and their *logical* counterparts on the EDF queues. Here we are not interested on how the threads are placed on the EDF queues. The aim is to show that while threads are placed on priority queues in FIFO order, their logical counterparts are placed in EDF order, and hence the overlapping arrows that show the mapping.

4. Related Work

There are three approaches to achieve flexible scheduling:

- **Pluggable schedulers** – in this approach the system provides a framework into which different schedulers can be plugged. The CORBA Dynamic Scheduling [13] specification is an example of this approach. Kernel loadable schedulers also fall into this category, such as that used within the SHaRK kernel [11].
- **Application-defined schedulers** – in this approach, the system notifies the application every time an event occurs that requires a scheduling decision to be taken. The application then informs the system which thread should execute next. The proposed extensions to real-time POSIX support this approach [1].
- **Implementation-defined schedulers** – in this approach, an implementation is allowed to define alternative schedulers. Typically this would require the underlying operating system (virtual machine, in the case of Java) to be modified. The Ada 95 language allows this approach.

Currently, 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 paper only assumes the presence of the priority scheduler and that priority changes have an immediate effect. An attempt has been made [9] to support a utility accrual scheduler in the RTSJ but this required a non standard interface and was not generalized. Similarly, although JTime supports multiple schedulers, this has been achieved in an ad hoc manner [7].

The use of dynamic priority changes to support alternative scheduling policies is well established. The approach adopted here is based on [4]. Li et al [12] have recently taken this approach and provided a formalized POSIX framework, although they do not support resource sharing between different schedulers.

5. Discussion and Conclusions

5.1. Specification Changes

The current RTSJ scheduler framework is under-specified and alternative schedulers cannot be implemented in a standard way. Furthermore, applications that use such implementation-defined schedulers will not be portable (by definition) between different implementations. Consequently, it is inevitable that the specification will have to evolve if it is to meet the demands of future real-time applications. In addition to the changes made to the `PriorityScheduler` class and the introduction of the new `ApplicationDefinedScheduler` class, a new `MonitorControl` policy is required to provide support for execution-eligibility inheritance. Baker's pre-emption level control policy is already well established and provides a sound theoretical basis for this. The implementation strategy given in [5] also allowed this to be applied effectively within a priority framework.

Moreover, the following additions to the RTSJ infrastructure are needed to fully support the approach:

- It is an anomaly that RTSJ does not provide the CPU time consumed by a schedulable object (or remaining) for the current release. These are essential for some scheduling algorithms (e.g. value-density scheduling).
- Several of the semantics governing the behaviour of schedulable objects are defined to be scheduler specific, with only those for the priority scheduler given. With the approach proposed in this paper, these semantics will be applied to all SO irrespective of their controlling application-defined schedulers.

5.2. 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 the RTSJ processing group parameters mechanism. This means that the threads within a scheduling band can be treated as if they are being served by a deferrable server [15]. 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 no more than its full budget each period. Hence, it can only give independent partial guarantees. To give full guarantees needs a global server-based analysis (see [6]). To give full independent guarantees requires the base scheduler to guarantee the capacity specified in the processing group parameters, which would be a change to the processing group semantics.

5.3. Conclusions

One of the initial goals of the RTSJ was to support the state-of-practice in real-time systems development and mechanisms to allow advances in state-of-the-art. As a result of this, the specification provides several frameworks that are only partially instantiated. The scheduler framework is one of them (the others being the feasibility analysis framework, the clock framework and the physical memory framework). Whilst this is laudable, more specification work is needed if these frameworks are to become usable in a standard and portable way. In this paper we have extended the scheduler framework to allow the hierarchical scheduling of real-time systems within priority bands. The approach is backward compatible with the current version of the RTSJ in that programs that do not define their own schedulers will execute unchanged on a version of RTSJ that supported the approach proposed in this paper.

A longer version of the paper can be found at: <http://www.cs.york.ac.uk/rts/publications.html>

Acknowledgements

The authors gratefully acknowledge the discussions they had with Alan Burns, Hao Cai, Rob Davis and Peter Dibble.

References

- [1] Aldea Rivas, M., and González Harbour, M. (2002), "POSIX-Compatible Application-Defined Scheduling in MaRTE OS", 14th Euromicro Conference on Real-Time Systems, IEEE Computer Society Press, pp. 67–75.
- [2] Baker, T.P (1991) , "Stack-Based Scheduling of Real-Time Processes", *Real-Time Systems Journal*, 3(1), pp. 57-99.
- [3] Brandt et al (2003), "Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes", pp.396, 24th IEEE RTSS.
- [4] Burns, A., and Wellings, A.J. (1995), "Concurrency in Ada, 2nd Edition", Addison Wesley.
- [5] Burns, A., Wellings, A.J., and Taft, T. S. (2004), "Supporting Deadlines and EDF Scheduling in Ada", *Lecture Notes in Computer Science*, Springer-Verlag, Volume 3063 / 2004, pp. 156-165.
- [6] Davis, R. I., and Burns, A. (2005), "Hierarchical Fixed Priority Pre-emptive Scheduling", p.389-398, RTSS 2005.
- [7] Dibble, P. and Wellings, A.J. (2004), "The Real-Time Specification for Java: Current Status and Future Direction", 7th International Conference on Object-Oriented Real-Time Distributed Computing, ISORC 2004, pp. 71-77.
- [8] Dibble (Ed) (2005), "The Real-Time Specification for Java", Version 1.0.1, www.rtsj.org
- [9] Feizabadi et al (2003), "Utility Accrual Scheduling with Real-Time Java", *JTRES 03*, pp. 550-563, *Lecture Notes in Computer Science*, Springer-Verlag Heidelberg, Vol. 2889/2003.
- [10] "FIRST" European Union IST Project (2005), "FIRST: Flexible Integrated Real-Time Systems Technology, Final Report", Deliverable D-FR, June 2005, <http://130.243.76.81:8080/salsart/first/>.
- [11] Gai et al (2001), "A New Kernel Approach for Modular Real-Time Systems Development", p. 199, *Proceedings of the 13th Euromicro Conference on Real-Time Systems*.
- [12] Li et al (2004), "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems", *IEEE Transactions on Software Engineering*, 30(9), pp. 613-629.
- [13] OMG (2003), "Real-time Corba Version 2.0", *OMG Document formal/03-11-01*, <http://www.omg.org/docs/formal/03-11-01.pdf>
- [14] Regehr, J., Jones, M. B., and Stankovic, J. A. (2000), "Operating System Support for Multimedia: The Programming Model Matters", *Technical Report MSR-TR-2000-89*, <http://research.microsoft.com/~mbj/papers/tr-2000-89.pdf>.
- [15] Strosnider, J. K., Lehoczky, J. P., and Sha, L. (1995), "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments", pp.73–91, *IEEE Transactions on Computers* 44, 1 (January 1995).