M. Teresa Higuera-Toledano
Andy J. Wellings *Editors*

Distributed, Embedded and Real-time Java Systems

Springer

Distributed, Embedded and Real-time Java Systems

M. Teresa Higuera-Toledano • Andy J. Wellings
Editors

Distributed, Embedded and Real-time Java Systems

Springer

Editors
M. Teresa Higuera-Toledano
Universidad Complutense de Madrid
Facultad de Informática, DACYA
Calle del Profesor Gaecía
Santesmas
28040 Madrid
Spain

Andy J. Wellings
Department of Computer Science
University of York
Heslington
York
United Kingdom

ISBN 978-1-4419-8157-8 e-ISBN 978-1-4419-8158-5
DOI 10.1007/978-1-4419-8158-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011945437

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The Java language has had an enormous impact since its introduction in the last decade of the twentieth century. Its success has been particularly strong in enterprise applications, where Java is one of the preeminent technology in use. A domain where Java is still trying to gain a foothold is that of real-time and embedded systems, be they multicore or distributed. Over the last 5 years, the supporting Java-related technologies, in this increasingly important area, have matured. Some have matured more than others. The overall goal of this book is to provide insights into some of these technologies.

Audience

This book is aimed at researchers in real-time embedded systems, particularly those who wish to understand the current state of the art in using Java in this domain. Masters students will also find useful material that founds background reading for higher degrees in embedded and real-time systems.

Structure and Content

Much of the work in real-time distributed, embedded and real-time Java has focused on the Real-time Specification for Java (RTSJ) as the underlying base technology, and consequently many of the Chapters in this book address issues with, or solve problems using, this framework.

Real-time embedded system are also themselves evolving. Distribution has always been an important consideration, but single processor nodes are rapidly being replaced by multicore platforms. This has increased the emphasis on parallel programming and has had a major impact on embedded and real-time software development. The next three chapters, therefore, explore aspects of this

issue. In Chap. 1, Wellings, Dibble and Holmes discuss the impact of multi-core on the RTSJ itself and motivate why Version 1.1 of the specification will have more support for multiprocessors. Central to this support is the notion of processor affinity.

RTSJ has always remained silent on issues of distributed system as other work in the Java Community has been addressing this problem. Unfortunately, this work has been moribund for several years and is now considered inactive. It is, therefore, an appropriate time to re-evaluate the progress in this area. In Chap. 2, Basanta-Val and Anderson reviews the state of the art in distributed real-time Java technology considering both the problems and the solutions.

As embedded real-time system grow in size, they will become more complex and have to cope with a mixture of hard and soft real-time systems. Scheduling these systems, within the imposed time constraints, is a challenging problem. Of particular concern is how to integrate aperiodic events into the system. The accepted theoretical work in this area revolves around the usage of execution-time servers. In Chap. 3, Masson and Midonnet consider how these servers can be programmed in the RTSJ at the application level.

Irrespective of size, embedded systems have imposed constraints, be they power consumption, heat dissipation or weight. This results in the resources in the platform being kept to a minimum. Consequently, these resources must be carefully managed. Processors and networks are the two of the main resource types, and have been considered in Chaps. 1–3. Memory is the other main resource, and it is this that is the topic of Chap. 4 through Chap. 6. Real-time garbage collection is crucial to the success of real-time Java and the advent of multicore has added new impetus to that technology to produce parallel collectors. In Chap. 4, Siebert explains the basic concepts behind parallel collectors and reviews the approaches taken by the major RTSJ implementations.

RTSJ introduced a form of region-based memory management as an alternative to the use of the heap memory. This has been one of the most controversial features of the specification. For this reason, Higuera-Toledano, Yovine, and Garbervetsky (in Chap. 5) evaluate the role of region-based memory management in the RTSJ and look at the strengths and weaknesses of the associated RTSJ scoped memory model. The third aspect of memory management for embedded systems is how to access the underlying platform's physical memory – in order to maximize performance and interact with external devices. This has always been a difficult area and one that has been substantially revised in RTSJ Version 1.1. In Chap. 6, Dibble, Hunt and Wellings considers these low-level programming activities.

Part and parcel of developing embedded systems is deciding on the demarcation between what is in software and what is in hardware. From a Java perspective, there are two important topics: hardware support for the JVM and how applications call code that has been implemented in hardware. These two issues are addressed in Chaps. 7 and 8. In Chap. 7, Schoeberl reviews the approaches that have been taken to support the execution for Java programs. This includes important areas like support for the execution of Java byte code and garbage collection. In contrast, Whitham and

Audsley in Chap. 8, focus on the interaction between Java programs and functional accelerators implemented as coprocessors or in FPGAs.

Embedded systems are ubiquitous and increasingly control vital operations. Failure of these systems have an immediate, and sometimes catastrophic, impact on society's ability to function. Proponents of Java technology envisage a growing role for Java in the development of these high-integrity (often safety-critical) systems. A subset of Java augmented with the RTSJ, called Safety Critical Java (SCJ) is currently being developed by the Java Community Process. This work is approaching fruition and two of the most important aspects are discussed in Chaps. 9 and 10. Most embedded and real-time system directly or indirectly support the notion of a mission. In the RTSJ, this notion can be implemented but there is not direct representation. SCJ provides direct support for this concept. In Chap. 9, Nielsen and Hunt discuss the mission concept and show how it can be used. Paramount to high-integrity applications is reliability. Although Java is a strongly typed language run-time exceptions can still be generated. Recent versions of the language have added support for annotations, which essentially allow added information to be passed to the compiler, development tools supporting static analysis, and the JVM itself. In Chap. 10, Tang, Plsek and Jan Vitek explore the SCJ use of annotations to support memory safety, that is the absence of run-time exceptions resulting from violations of the SCJ memory model.

Up until now, the topics that have been addressed have been concerned with the underlying technology for real-time and embedded Java. The remainder of the book focuses on higher level issues. Real-time and embedded system designers have to be more resource-aware than the average Java programmer. Inevitably, this leads to complexities in the programming task. The RTSJ has introduced abstractions that help the programmer manage resources; however, some of these abstraction can be difficult to use (for example, scoped memory). In Chap. 11, Plsek, Loiret and Malohlava argue that this can make the use of the RTSJ error prone. They consider the role of RTSJ-aware component-oriented frameworks that can help provide a clearer separation between RTSJ-related issues and application-related issues.

One well-known Java framework for component-based service-oriented architectures is the Open Services Gateway initiative (OSGi). Although this initiative has wide support from industry, it lacks any support for real-time applications. In Chap. 12, Richardson and Wellings propose the integration of OSGi with the RTSJ to provide a real-time OSGi framework.

In the final chapter of this book, Holgado-Terriza and Videz-Aivar address the issue of building a complete embedded application that includes both software and hardware components from a reusable base.

Acknowledgements

The editors are grateful to Springer who gave us the opportunity to produce this book, and to all the authors for agreeing to contribute (and for reviewing each others chapters).

We are also indebted to Sitsofe Wheeler for his help with latex.

Andy J. Wellings
M. Teresa Higuera-Toledano

Contents

Chapter 1
Supporting Multiprocessors in the Real-Time Specification for Java Version 1.1

Andy J. Wellings, Peter Dibble, and David Holmes

Abstract Version 1.1 of the Real-Time Specification for Java (RTSJ) has significantly enhanced its support for multiprocessor platforms. This chapter discusses the rationale for the approach adopted and presents details of the impact on the specification. In particular, it considers the new dispatching and processor affinity models, issues of cost enforcement and the affinity of external events.

1.1 Introduction

Multiprocessor platforms (be they multicore or multichip systems)[1] are becoming more prevalent and their role in the provision of a wide variety of real-time and embedded systems is increasing. In particular symmetric multiprocessor (SMP) systems are now often the default platform for large real-time systems rather than a single processor system.

[1]The term *processor* is used in this chapter to indicate a logical processing element that is capable of physically executing a single thread of control at any point in time. Hence, multicore platforms have multiple processors, platforms that support hyperthreading also have more than one processor. It is assumed that all processors are capable of executing the same instruction sets.

A.J. Wellings (✉)
Department of Computer Science, University of York, York, UK
e-mail: andy@cs.york.ac.uk

P. Dibble
TimeSys, Pittsburgh, PA, USA
e-mail: peter.dibble@timesys.com

D. Holmes
17 Windward Place, Jacobs Well, QLD 4208, Australia
e-mail: dholmes@ieee.org

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_1,

Since its inception, the Real-Time Specification (RTSJ) has preferred to remain silent on multiprocessor issues. It allows vendors to support multiprocessor implementations, but provides no guidelines on how the specification should be interpreted in this area. Furthermore, it provides no programming abstractions to help developers configure their systems. The latest version of the RTSJ recognizes that this position is no longer tenable. However, multiprocessor, and multicore architectures in particular, are evolving at a bewildering rate, and Operating System Standards (in particular the suite of POSIX Standards) have yet to catch up. Consequently, it is difficult to determine the level of support that the specification should target. This is compounded by the requirement to stay faithful to the guiding principles that have delimited the scope of the RTSJ. These include [65]:

- Predictable execution as the first priority in all tradeoffs.
- Write once, run anywhere but not at the expense of predictability (later rephrased to "write once carefully, run anywhere conditionally").
- Address current practice but allow future implementations to include enhanced features.
- Allow variations in implementation decisions.

This chapter is structured as follows. In Sect. 1.2, the motivations for providing more direct support for multiprocessors in the RTSJ is given along with the constraints imposed by the RTSJ guiding principles. In the majority of cases, the RTSJ run-time environment executes as middleware on top of an underlying operating system. The facilities provided by these operating systems constrain further the possible support the specification can provide. Section 1.3 reviews the support that is typical of current operating systems in widespread use. These motivations and constraints are used to generate, in Sect. 1.4, a list of multiprocessor requirements for the RTSJ Version 1.1. Then, in Sect. 1.5, the full support for multiprocessor systems in the RTSJ version 1.1 is discussed. Finally, conclusions are drawn. An appendix details the new API.

1.2 Motivations and Constraints

Whilst many applications do not need more control over the mapping of schedulable objects[2] to processors, there are occasions when such control is important. Here the focus is on symmetric multiprocessor (SMP) systems, which are defined to be identical multiprocessors that have access to shared main memory with a uniform access time. The impact of non uniform memory architectures is left for future versions of the specification, as are heterogeneous processor architectures or systems with homogeneous processor architectures with different performance characteristics.

[2]The RTSJ uses the generic term schedulable object where other languages and operating systems might use the term real-time thread or task.

1.2.1 Motivation: To Obtain Performance Benefits

The primary motivation for moving from a single processor to a multiprocessor implementation platform is to obtain an increase in performance. Dedicating one CPU to a particular schedulable object will ensure maximum execution speed for that schedulable object. Restricting a schedulable object to run on a single CPU also prevents the performance cost caused by the cache (TLB or other non-memory related context associated with the schedulable object) invalidation that occurs when a schedulable object ceases to execute on one CPU and then recommences execution on a different CPU [251]. Furthermore, configuring interrupts to be handled on a subset of the available processors allows performance-critical schedulable objects to run unhindered on other processors.

1.2.2 Motivation: To Support Temporal Isolation

Where an application consists of subsystems of mixed criticality levels (non-critical, mission-critical and safety-critical) running on the same execution platform, some form of protection between the different levels is required. The strict typing model of Java provides strong protection in the spatial domain. The Cost Enforcement Model of the RTSJ (including processing group parameters) provides a limited form of temporal protection but is often very coarse grained. More reliable temporal protection is obtainable by partitioning the set of processors and allowing schedulable objects in each criticality level to be executed in a separate partition. For example, placing all non-critical schedulable objects in a separate partition will ensure that any overruns of these less critical schedulable objects cannot impact a high-criticality subsystem.

1.2.3 Constraint: Predictability of Scheduling

The scheduling of threads on multiprocessor systems can be:

1. Global – all processors can execute all schedulable objects.
2. Fully partitioned – each schedulable object is executed only by a single processor; the set of schedulable objects is partitioned between the set of processors.
3. Clustered – each schedulable object can be executed by a subset of the processors; hence the schedulable objects set may be partitioned into groups and each group can be executed on a subset of the processors.

In addition to the above, there are hybrid schemes. For example, semi partitioned schemes fixed the majority of schedulable objects to individual processors but allow the remainder to migrate between processors (sometime at fixed points in their executions). The situation can become even more complex if the platform can vary the number of processors available for execution of the program.

Although the state of the art in schedulability analysis for multiprocessor systems continues to advance [26, 121], it is clear that for fixed priority scheduling, neither global nor fully-partitioned systems are optimal. There is no agreed approach and the well-known single processor scheduling algorithms are not optimal in a multiprocessor environment. Cluster-based scheduling seems essential for systems with a large number of processors. However, within cluster, hybrid approaches allow greater utilization to be achieved over global scheduling within cluster [121].

Hence, it is not possible for the RTSJ to provide a default priority-based multiprocessor scheduler that takes care of the allocation of schedulable objects to processors. And even if it was, it would require a very specialized real-time operating system to support it.

1.2.4 Constraint: Predictability of Resource Sharing

Given the discussion in Sect. 1.2.3, it is hardly surprising that the sharing of resources (shared objects) between schedulable objects, which potentially can be executing on different processors, is problematic. The essence of the problem is how to deal with contention for global resources; how to keep the data coherent and how to bound the time that a schedulable object waits for access to it.

Over the last 40 years, many different approaches have been explored. One approach that has maintained its popularity over the years (and which has provided the inspiration for the Java model) is the monitor. A monitor encapsulates a shared resource (usually some shared variables) and provides a procedural/functional interface to that resource.

In Java, a monitor is an object with the important property that the methods that are labeled as synchronized are executed atomically with respect to each other. This means that one synchronized method call cannot interfere with the execution of another synchronized method. The way this is achieved, in practice, is by ensuring that the calls are executed in mutual exclusion. There are several different ways of implementing this, for example, by having a lock associated with the monitor and requiring that each method acquires (sets) the lock before it can continue its execution. Locks can be *suspension-based* or *spin-based*.

From a multiprocessor real-time perspective, the following typically apply.

- Suspension-based locking – if it cannot acquire the lock, the calling schedulable object is placed in a priority-ordered queue and waits for the lock. To bound the blocking time, priority inversion avoidance algorithms are used.
- Spin-based locking – if it cannot acquire the lock, the calling schedulable object busy-waits for the lock to become available. To bound the blocking time, the schedulable object normally spins non-preemptively (i.e., at the highest priority) and is placed in a FIFO or priority-ordered queue. The lock-owner also runs non-preemptively, and consequently is prohibited from accessing nested locks and must not suspend whilst holding a lock.

Of course, optimizations are possible, such as allowing the lock to only be acquired when contention actually occurs, or initially spinning for the lock before suspending [172]. On a single processor system it is also possible to turn off all hardware interrupts whilst holding the lock, and therefore prohibit any scheduling preemptions.

Mutual exclusion is a very heavy-weight mechanism to maintain data consistency. It reduces the concurrency in the system by blocking the progress of schedulable objects, and causes the well-known problems of deadlock, starvation, priority inversion etc.

This has led to the development of non-blocking approaches. An approach is non-blocking if the failure or suspension of any schedulable objects cannot cause the failure or suspension of any other schedulable object [172]. Non-blocking algorithms are usually classified as either *lock-free* or *wait-free*. From a multiprocessor real-time perspective, the following typically apply [79].

- Lock-free – access to the shared object is immediate but, following completion of the desired operation, the schedulable object must be able to detect and resolve any contention that may have occurred. Often resolving the conflict requires retrying the operation. When contention occurs, some schedulable object is guaranteed to make progress. However, it is not defined which one. Software transactional memory is an example where a lock-free approach is possible. Bounding the retries is a major challenge for the real-time algorithm designer.
- Wait-free – access to the shared resource is immediate and is guaranteed to complete successfully within finite time. Typically, specific wait-free data structures are highly specialized and are difficult to design and prove correct. Examples include, a wait-free queue or a wait-free stack. Generic wait-free read-write access to a single shared object can be supported (e.g. using Simpson's algorithm [379] for a single reader and single writer) but at the cost of replicating the data. Hence, data read may be stale if a writer is currently updating it.

Quoting from Brandenburg et al. [79] who give the following conclusions from an in-depth study on real-time synchronization on multiprocessors:

- when implementing shared data objects, non-blocking algorithms are generally preferable for small, simple objects, and wait-free or spin-based (locking) implementations are generally preferable for large or complex objects;
- Wait-free algorithms are preferable to lock-free algorithms;
- with frequently occurring long or deeply-nested critical sections, schedulability is likely to be poor (under any scheme);
- Suspension-based locking should be avoided for global resources in partitioned systems[3];
- using current analytical techniques, suspension-based locking is never preferable (on the basis of schedulability or tardiness) to spin-based locking;

[3]This is a paraphrase from the original which focused on a particular partitioning scheme.

- if such techniques can be improved, then the use of suspension-based locking will most likely not lead to appreciably better schedulability or tardiness than spinning unless a system (in its entirety) spends at least 20% of its time in critical sections (something we find highly unlikely to be the case in practice).

Of course, the choice of what approach to adopt is constrained by the data structures needed. There is no simple general approach to taking a particular data structure and producing a non-blocking access protocol.[4] Furthermore, it should be emphasized that this study excluded external devices with long access times for which suspension-based locking is essential.

1.3 Operating System Support

Although multiprocessors are becoming prevalent, there are no agreed standards on how best to address real-time demands. For example, the RTEM operating system does not dynamically move threads between CPU. Instead it provides mechanisms whereby they can be statically allocated at link time. In contrast, QNX's Nutrino [311] distinguishes between "hard thread affinity" and "soft thread affinity". The former provides a mechanism whereby the programmer can require that a thread be constrained to execute only on a set of processors (indicated by a bit mask). With the latter, the kernel dispatches the thread to the same processor on which it last executed (in order to cut down on preemption costs). Other operating systems provide similar facilities. For example, IBM's AIX allows a kernel thread to be bound to a particular processor.[5] In addition, AIX enables the set of processors (and the amount of memory) allocated to the partition executing an application to be changed dynamically.

Many multiprocessor systems allow interrupts to be targeted at particular processors. For example, the ARM Corex A9-MPCore supports the Arm Generic Interrupt Controller.[6] This allows a target list of CPUs to be specified for each hardware interrupt. Software generated interrupts can also be sent to the list or set up to be delivered to all but the requesting CPU or only the requesting CPU. Of course, whether the Operating System allows this targeting is another matter.

Safety critical system software must be predictable enough that most failures can be detected analytically or through systematic testing. This rules out the non-determinism implicit in flexible scheduling across multiple processors. It may even rule out interactions among threads on different processors. Currently there is limited use of general multiprocessor systems in safety critical systems. Traditionally,

[4] This can be contrasted to simply taking the current access protocol and surrounding it calls to an appropriate lock to support a blocking-based access protocol.

[5] See http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.basetechref/doc/basetrf1/bindprocessor.htm.

[6] See http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0375a/Cegbfjhf.html.

where multiprocessors are required they are used in a distributed processing mode: with boards or boxes interconnected by communications busses, and bandwidth allocation, and the timing of message transfers etc carefully managed. This "hard" partitioning simplifies certification and testing since one application cannot affect another except through well-defined interfaces.

However, there is evidence that future safety-critical systems will use SMP. For example, the LynxSecure Separation Kernel (Version 5.0) has recently been announced. The following is taken from their web site[7]:

- Optimal security and safety – the only operating system to support CC EAL-7 and DO-178B level A.
- Real time – time-space partitioned real-time operating system for superior determinism and performance.
- Highly scalable – supports Symmetric MultiProcessing (SMP) and 64-bit addressing for high-end scalability.

This work has been undertaken by Intel and LynuxWorks to demonstrate the MILS (Multiple Independent Levels of Security) architecture.[8]

In the remainder of this section, we briefly review the work that has been performed by the POSIX and Linux communities. We then consider Solaris as an example of the support provided by a vendor-specific OS.

1.3.1 POSIX

POSIX.1 defines a "Scheduling Allocation Domain" as the set of processors on which an individual thread can be scheduled at any given time. POSIX states that [286]:

- "For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used."
- "For application threads with scheduling allocation domains of size greater than one, the rules defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an implementation-defined manner."
- "The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time."

With this approach, it is only possible to write strictly conforming applications with real-time scheduling requirements for single-processor systems. If an SMP platform is used, there is no portable way to specify a partitioning between threads and processors.

[7] http://www.lynuxworks.com/rtos/secure-rtos-kernel.php.

[8] See http://www.intel.com/technology/itj/2006/v10i3/5-communications/6-safety-critical.htm.

Additional APIs have been proposed [272] but currently these have not been standardized. The approach has been to set the initial allocation domain of a thread as part of its thread-creation attributes. The proposal is only in draft and so no decision has been taken on whether to support dynamically changing the allocation domain.

1.3.2 Linux

Since Kernel version 2.5.8, Linux has provided support for SMP systems [136]. Partitioning of user processes and threads is obtained via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask [251]. A process's CPU affinity mask determines the set of CPUs on which its thread are eligible to run.

```
#include <sched.h>

int sched_setaffinity(pid_t pid,
  unsigned int cpusetsize, cpu_set_t *mask);

int sched_getaffinity(pid_t pid,
  unsigned int cpusetsize, cpu_set_t *mask);

void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);

void CPU_SET(int cpu, cpu_set_t *set);

void CPU_ZERO(cpu_set_t *set);
```

A CPU affinity mask is represented by the `cpu_set_t` structure, a "CPU set", pointed to by the mask. Four macros are provided to manipulate CPU sets. `CPU_ZERO` clears a set. `CPU_SET` and `CPU_CLR` respectively add and remove a given CPU from a set. `CPU_ISSET` tests to see if a CPU is part of the set. The first available CPU on the system corresponds to a cpu value of 0, the next CPU corresponds to a cpu value of 1, and so on. A constant `CPU_SETSIZE (1024)` specifies a value one greater than the maximum CPU number that can be stored in a CPU set.

`sched_setaffinity` sets the CPU affinity mask of the process whose ID is pid to the value specified by mask. If the process specified by pid is not currently running on one of the CPUs specified in mask, then that process is migrated to one of the CPUs specified in mask [251].

`sched_getaffinity` allows the current mask to be obtained. The affinity mask is actually a per-thread attribute that can be adjusted independently for each of the threads in a thread group.

Linux also allows certain interrupts to be targeted to specific processors (or groups of processors). This is known as SMP IRQ affinity. SMP IRQ affinity is controlled by manipulating files in the /proc/irq/ directory.

As well as being able to control the affinity of a thread from within the program, Linux allows the total set of processors (its cpuset) allocated to a process to be changed dynamically. Hence, the masks given to `sched_setaffinity()` must be moderated by the kernel to reflect those processors that have been allocated to the process. Cpusets are intended to provide management support for large systems.

The real-time scheduler of the PREEMPT_RT patchset uses cpusets to create a *root domain*. This is a subset of CPUs that does not overlap with any other subset. The scheduler adopts an active push-pull strategy for balancing real-time tasks (tasks who priority range from 0 to (MAX_RT_PRIO-1)) across CPUs. Each CPU has its own run queue. The scheduler decides [169]:

1. Where to place a task on wakeup.
2. What action to take if a lower-priority task is placed on a run queue running a task of higher priority.
3. What action to take if a low-priority task is preemted by the wake-up of a higher-priority task.
4. What action to take when a task lowers its priority and thereby causes a previously lower-priority task to have the higher priority.

Essentially, a *push* operation is initiated in cases 2 and 3 above. The push algorithm considers all the run queues within its root domain to find one that has a lower priority task (than the task being pushed) at the head of its run queue. The task to be pushed then preempts the lower priority task.

A *pull* operation is performed for case 4 above. Whenever the scheduler is about to choose a task from its run queue that is lower in priority than the previous one, it checks to see whether it can pull tasks of higher priority from other run queues. If it can, at least one higher priority tasks is moved to its run queue and is chosen as the task to be executed. Only real-time tasks are affected by the push and pull operations.

1.3.3 Solaris

The Solaris operating system provides facilities similar to those of Linux, though the details of the API's and the low-level semantics differ. The `processor_bind` system call can be used to bind one or more threads (or Lightweight Processes – LWPs – as they are known on Solaris) to a particular processor.

```
#include <sys/types.h>
#include <sys/processor.h>
#include <sys/procset.h>

int processor_bind(idtype_t idtype, id_t id,
                   processorid_t processorid,
                   processorid_t *obind);
```

Solaris allows threads to be grouped according to process,"task", "project", "process contract" and zone, as indicated by the `idtype_t`, which in turn dictates what kind of `id_t` is expected. The same system call can be used to query existing bindings, or to clear them, by using the special `processorid` values of `PBIND_QUERY` and `PBIND_NONE` respectively. The `obind` parameter is used to return the previous binding, or the result of a query.

The binding of threads to multiple processors is done through the Solaris processor set facility. Processor sets are similar to Linux's CPU sets, but rather than being hierarchical they form a flat, disjoint, grouping. Threads will not run on processors in a processor set unless the thread, or its process, are explicitly bound to that processor set. Programs can be run on a specific processor set using the *psrset* $-e$ $<$ *set*# $><$ *program* $>$ command-line utility. When a process is bound to a processor set in this way, its threads are restricted to the processors in that set – the `processor_bind` system call can then be used to further restrict a thread to a single processor within that. An unbound process can only run on an unbound processor, so there must always be at least one unbound processor in a system. The `psrset` command is an administrative tool for creating and configuring processor sets, as well as a means to execute programs on a particular set – there is a corresponding programmatic API for this as well. The `pset_bind` system call can then be used within a program to bind one or more threads to a different processor set to that of the process. For example, you might give all NHRTs a dedicated processor set to run on.

```
#include <sys/pset.h>

int pset_bind(psetid_t pset, idtype_t idtype,
              id_t id, psetid_t *opset);
```

A `psetid_t` is a simple integer that names a particular processor set. The special values `PS_QUERY` and `PS_NONE` indicate a request to query, or clear, the current processor set binding, respectively. The `opset` parameter is used to return the previous binding, or the result of a query.

Processors are managed by the `psradm` utility (or the equivalent programmatic API) which allows for processor to be taken offline (they won't execute any threads, but might field I/O device interrupts), or to be declared `no-intr` (they will execute threads but won't field any I/O device interrupts). Some interrupts will always be handled by a specific processor, such as per-cpu timer expirations.

Effective processor set management requires extremely detailed knowledge of the underlying hardware and how the operating system identifies individual "processors". On multi-core systems, for example, you need to understand how cores are numbered within a CPU and/or within a chip, and what resources (e.g. L2 cache) might be shared between which cores.

1.4 Requirements for RTSJ Version 1.1

Section 1.2 has summarized the motivations for providing more explicit support for multiprocessor systems in the RTSJ, along with the constraints that must be imposed to maintain predictable scheduling. From this, the following requirements are distilled.

- It shall be possible to limit the dispatching of a schedulable object to a single processor – this is in order to support the motivations given in Sects. 1.2.1 and 1.2.2 and to reflect the constraints given for schedulability in Sect. 1.2.3.

  This is impossible to achieve in Version 1.02 of the specification. Although, the `java.lang.Runtime` class allows the number of processors available to the Java Virtual Machine (JVM) to be determined by the `availableProcessors()` method, it does not allow Java threads to be pinned to processors.
- Global scheduling of schedulable objects across more than one processor shall be possible – this is primarily to reflect the constraints given for schedulability in Sect. 1.2.3, and also because this is the default position for RTSJ version 1.0.2 implementations.
- Spin-based queue locking protocols for shared objects shall not be precluded – this is to reflect the predictability constraints given in Sect. 1.2.4.

  The Java intrinsic synchronization approach is built on suspension-based locking. The RTSJ, with its support for priority ceiling emulation, potentially allows a no-lock implementation of Java monitors on a single processor but only for constrained synchronized methods and statements (i.e. ones that do not self suspend). Implementing non-locking Java monitors is difficult to support in general. The RTSJ wait-free queue facility provides an alternative mechanism for schedulable objects to communicate.
- The impact of the Java-level handling of external events should be constrainable to a subset of the available processors – this is to reflect the motivations given in Sect. 1.2.1.

  In the RTSJ all external events are associated with *happenings*. Whilst it may not be possible to constrain where the first-level interrupt code executes, it should be possible to control where the Java code executes that responds to the interrupt.

1.5 The RTSJ Version 1.1 Model

The support that the RTSJ provides for multiprocessor systems is primarily constrained by the support it can expect from the underlying operating system. The following have had the most impact on the level of support that has been specified.

- The notion of processor *affinity* is common across operating systems and has become the accepted way to specify the constraints on which processor a thread can execute.

  RTSJ 1.1 directly supports affinities. A *processor affinity set* is a set of processors that can be associated with a Java thread or RTSJ schedulable object. The internal representation of a set of processors in an `AffinitySet` instance is not specified, but the representation that is used to communicate with this class is a `BitSet` where each bit corresponds to a logical processor ID. The relationship between logical and physical processors is implementation defined.

  In some sense, processor affinities can be viewed as additional release or scheduling parameters. However, to add them to the parameter classes requires the support to be distributed throughout the specification with a proliferation of new constructor methods. To avoid this, support is grouped together within the `ProcessorAffinitySet` class. The class also allows the addition of processor affinity support to Java threads without modifying the thread object's visible API.
- The range of processors on which global scheduling is possible is dictated by the operating system. For SMP architectures, global scheduling across all the processors in the system is typically supported. However, an application and an operator can constrain threads and processes to execute only within a subset of the processors. As the number of processors increase, the scalability of global scheduling is called into question. Hence, for NUMA architectures some partitioning of the processors is likely to performed by the OS. Hence, global scheduling across all processors will not be possible in these systems.

  The RTSJ supports an array of *predefined affinity sets*. These are implementation-defined. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for, say, Java threads, non-heap real-time schedulable objects etc. A program is only allowed to dynamically create new affinity sets with cardinality of one. This restriction reflects the concern that not all operating systems will support multiprocessor affinity sets.
- Many OSs give system operators command-level dynamic control over the set of processors allocated to a processes. Consequently, the real-time JVM has no control over whether processors are dynamically added or removed from its OS process.

  Predictability is a prime concern of the RTSJ. Clearly, dynamic changes to the allocated processors will have a dramatic, and possibly catastrophic, effect on the ability of the program to meet timing requirements. Hence, the RTSJ assumes that the processor set allocated to the RTSJ process does not change during its execution. If a system is capable of such manipulation it should not exercise it on RTSJ processes.

In order to make the RTSJ 1.1 fully defined for multiprocessor systems, the following issues needed to be addressed.

1. The dispatching model – version 1.02 of the specification has a conceptual model which assumed a single run queue per priority level.
2. The processor allocation model – version 1.02 of the specification provides no mechanisms to support processor affinity.
3. The synchronization model – version 1.02 of the specification does not provide any discussion of the intended use of its monitor control policies in multiprocessor systems.
4. The cost enforcement model – version 1.02 of the specification does not consider the fact that processing groups can contain scheduling objects which might be simultaneously executing.
5. The affinity of external events (happenings) – version 1.02 of the specification provides no mechanism to tie happenings handlers to particular processors.

The remainder of this section considers each of the above issues in turn. An appendix gives the associated APIs.

1.5.1 The Dispatching Model

The RTSJ dispatching model specifies its dispatching rules for the default priority scheduler. Here, the rules are modified to address multiprocessor concerns.

1. A schedulable object can become a running schedulable object only if it is ready and the execution resources required by that schedulable object are available.
2. Processors are allocated to schedulable objects based on each schedulable object's active priority and their associated affinity sets.
3. Schedulable object dispatching is the process by which one ready schedulable object is selected for execution on a processor. This selection is done at certain points during the execution of a schedulable object called *schedulable object dispatching points*. A schedulable object reaches a *schedulable object dispatching point* whenever it becomes blocked, when it terminates, or when a higher priority schedulable object becomes ready for execution on its processor.
4. The schedulable object dispatching policy is specified in terms of *ready queues* and schedulable object states. The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation.
5. A ready queue is an ordered list of ready schedulable objects. The first position in a queue is called the head of the queue, and the last position is called the tail of the queue. A schedulable object is ready if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of schedulable objects of that priority that are ready for execution on that processor, but are not running on any processor; that is, those schedulable objects that are ready, are not running on any processor, and can be executed using that processor. A schedulable object can be on the ready queues of more than one processor.