Multiprocessors and the Real-Time Specification for Java

A.J. Wellings
Department of Computer Science, University of York, UK
andy@cs.york.ac.uk

Abstract

Currently, the Real-Time Specification for Java (RTSJ) is silent on multiprocessor issues It attempts not to preclude multiprocessor implementations but provides no direct support. This paper discusses the issues that need to be addressed if the RTSJ is to be better defined for execution on a multiprocessor system. It proposes new dispatching and allocation models. Issues of cost enforcement, interrupts affinity and processor failure are covered.

1. Introduction

Multiprocessor systems are becoming more prevalent. In particular symmetric multiprocessor (SMP) systems are often the default platform for large real-time systems rather than a single processor system. The scheduling of threads[1] on these 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. mixed – 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.

Furthermore, on some SMPs the number of processors allocated to an application may vary during the execution of the program.

Currently, the RTSJ is silent on multiprocessor issues. It attempts not to preclude multiprocessor implementations but provides no direct support. The `java.lang.Runtime` class allows the number of processors available to the Java Virtual Machine (JVM) to be determined by the `int availableProcessors()` method, but does not allow Java threads to be pinned to processors.

This paper discusses the support that the RTSJ could provide for SMP systems. The impact of non uniform memory architectures is left for future work.

2. Motivation

Whilst many applications do not need more control over the mapping of schedulable objects to processors in an SMP environment, there are occasions when such control is important. They include:

- To allow more flexible approaches to scheduling.

  Although the state of the art in schedulability analysis for multiprocessor systems continues to advance [1], the current state is such that partitioned systems offer more guaranteed schedulability than global systems. Quoting from [2]:

  "In favor of global scheduling, it has long been known from queueing theory that single-queue (global) FIFO multiprocessor scheduling is superior to queue-per-processor (partitioned) FIFO scheduling, with respect to average response time.

  Apparently in favor of partitioned scheduling, the application of well-known single processor scheduling algorithms appears superior to the global application of those same algorithms for some task sets with hard-deadlines. For example, it is known that all periodic implicit-deadline task sets with utilization below $m(2^{1/2} - 1)$ can be scheduled on m processors using a first-fit-decreasing-utilization (FFDU) partitioning algorithm and local rate monotonic scheduling, but Dhall's example shows that there are hard-deadline periodic task sets with total utilization arbitrarily close to 1.0 that cannot meet all deadlines if scheduled on m processors using global rate monotonic scheduling. Dhall's example also applies to global EDF scheduling, yet FFDU partitioned EDF scheduling is guaranteed up to utilization $(m + 1)/2$.

  However, the supposed advantage of partitioned scheduling above disappears if one considers

1 From now on, this paper will use the RTSJ term *schedulable object* to represent threads, tasks, processes etc.

978-0-7695-3132-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ISORC.2008.22

hybrid global priority schemes. The Dhall example can easily be handled by the $EDF - US(1/2)$ or $EDF(k_{min})$ schemes, in which top priority is given to a few "heavy" tasks, as can any implicit deadline sporadic task system with utilization up to $(m + 1)/2$. This is exactly the same bound as for FFDU partitioned scheduling!

The experiments we performed on large numbers of pseudo-randomly generated task sets were intended to provide some additional evidence on which to base a choice between these two approaches. From those experiments, statistically, the chance of being able to satisfy all the deadlines of a randomly chosen periodic or sporadic task set appears to be highest with partitioned scheduling. In particular, the partitioned EDF scheduling appeared to be the overall best performer in this statistical sense. At the same time, there are certainly specific task sets where global scheduling is more effective."

- To support temporal isolation.

  Where an application consists of tasks of mixed criticality level, some form of protection between the different levels is required. The strict typing model of Java provides a strong degree of protection in the spatial domain. The Cost Enforcement Model of the RTSJ (including processing group parameters) provides a limited form of temporal protection but at the expense of flexibility. Furthermore, this can be integrated with Java isolates. More flexible temporal protection is obtainable by allowing schedulable objects in each criticality level to be executed on partitions of the processor set.

- To obtain performance benefits.

  For example, 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 invalidation that occurs when a schedulable object ceases to execute on one CPU and then recommences execution on a different CPU [3]. Furthermore, configuring interrupts to be handled on a subset of the available processors allows performance-critical schedulable objects to run unhindered.

- To be able to respond to dynamic changes to the processor set.

  In a parallel computing environment the set of processors allocated to an application may vary depending on the global state of the system. An application may be able to optimize its algorithms if it is informed when these changes in the processor set occur.

3. Previous Work

Although multiprocessors are becoming prevalent, there are no agreed standards on how best to address real-time demands. For example, 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[4] 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 tries to dispatch the thread to the same processor on which it last executed (in order to cut down on preemption costs). Other operating systems provide slightly different facilities. For example IBM's AIX allows a kernel thread to be bound to a particular processor [2]. Further more, the set of processors (and the amount of memory) allocated to a partition in AIX can change dynamically.

Currently there is limited use of general multiprocessor systems in safety critical systems. Traditionally, 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 systems will use SMP. For example, the LynxSecure Separation Kernel has recently been announced. The following is taken from their web site[3]:

- 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/Safety) architecture[4].

In the remainder of this section, we briefly review the work that has been performed by the POSIX and Linux communities.

2 see http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/-com.ibm.aix.basetechref/doc/basetrf1/bindprocessor.htm

3 http://www.lynuxworks.com/rtos/secure-rtos-kernel.php.

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

3.1. POSIX

Although POSIX does not currently provide specific support for SMP systems, the issue has been raised [5]. POSIX.1 defines the "Scheduling Allocation Domain" as the set of processors on which an individual thread can be scheduled at any given time. POSIX states that [6]:

- "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.

Additional APIs have been proposed 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[5].

3.2. Linux

Since Kernel version 2.5.8, Linux has provided support for SMP systems [3] via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask. A process's CPU affinity mask determines the set of CPUs on which it is 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.

`sched_getaffinity` allows the current mask to be obtained.

An error is returned if the affinity bitmask mask contains no processors that are physically on the system, or `cpusetsize` is smaller than the size of the affinity mask used by the kernel.

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.

4. The Proposed Model

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

1. The dispatching model – the current specification has a conceptual model which assumes a single run queue per priority level.
2. The allocation model – the current specification provides no mechanisms to support processor affinity,
3. The cost enforcement model – the current specification does not consider the fact that processing groups can contain scheduling objects which might be simultaneously executing,
4. The affinity of interrupts (happenings) – the current specification provides no mechanism to tie interrupts (happenings) and their handlers to particular processors, and
5. The failure model – the current specification makes no statements about partial failures of the underlying platform.

This section considers each of the above issues in turn. An appendix gives the proposed API.

4.1. 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 and generalised away from priority to execution eligibility [5].

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. An example of a possible implementation-defined execution resource is a page of physical memory, which needs to be loaded with a particular page of virtual memory before a schedulable object can continue execution.
2. Processors are allocated to schedulable objects based on each schedulable object's active priority.
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.
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 and other available resources. A schedulable object can be on the ready queues of more than one processor.
6. Each processor also has one running schedulable object, which is the schedulable object currently being executed by that processor. Whenever a schedulable object running on a processor reaches a schedulable object dispatching point it goes back to one or more ready queues; a schedulable object (possibly the same schedulable object) is then selected to run on that processor. The schedulable object selected is the one at the head of the highest priority nonempty ready queue; this schedulable object is then removed from all ready queues to which it belongs.
7. While a schedulable object is running, it is not on any ready queue. Any time the schedulable object that is running on a processor is added to a ready queue, a new running schedulable object is selected for that processor.

In a multiprocessor system, a schedulable object can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready schedulable objects, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.

4.2. The Allocation Model

In the general case, the following assumptions are made about the possible support provided by the underlying platform (operating system and hardware).

1. An application program may be allocated (by the operating system) the full set of the processors in the system or only a subset of them. An initial allocation is performed at the start of program execution time.
2. The operating system may only support global scheduling of threads or it may allow threads to be constrained to one or more processors in the set allocated to the program.
3. The operating systems may dynamically change the allocation of processors allocated to a program during the program's execution. If it does this, it is done in a safe manner.
4. Mechanisms may be provided by the operating system to inform the application (if the operating system supports thread to processor allocation) or they may not (if it only supports global scheduling).

The requirement is for a minimum interface that allows the pinning of a schedulable object to one or more processors. The challenge is to define the API so that it allows a range of operating system facilities to be supported. The minimum functionality is for the operating system to allow the real-time JVM to determine how many processors are available for the execution of the Java application.

5 This model is based this on the dispatching model defined for Ada 2005, which does address some aspects of multiprocessor execution.

The number of processors that the real-time JVM is aware of is represented by a BitSet that is returned by the static method `availableProcessors()` in the `RealtimeSystems` class. For example, in a 64 processor system, the real-time JVM may be aware of all 64 or only a subset of those. This is the length of the bitset. Of these processors, the real-time JVM will know which processors have been allocated to it (either logical processors or physical processors depending on the operating system). Each of the available processors is set to one in the bit set. Hence, the cardinality of the bit set represents the number of processors that the real-time JVM thinks are currently available to it.

The proposed API allows for systems that support the dynamic addition and removal of processors from the set allocated to the JVM. If an operating system does not support this facility then the set will not dynamically change. An operating system is also allowed to maintain a set of logical processors allocated to the JVM and to transparently change its logical to physical mapping. Again, from the JVM perspective the set has not changed. However, it should be noted that this may have an impact on the application if a) it is handling interrupts directly on the processor or b) if the change undermines any feasibility analysis assumptions. For many RTSJ applications this may not be a problem. In all of the above circumstances the static method `affinityChangeNotificationSupported()` defined in the `RealtimeSystems` class returns false.

If the operating system does support dynamic changes to the processor set, the assumption is that it will inform the real-time JVM of the changes (e.g. via a signal). The JVM will pass this information to the application via the firing of the appropriate asynchronous event (`ProcessorRemoved` or `ProcessorAdded`) declared in the `RealtimeSystems` class. In this circumstances `affinityChangeNotificationSupported()` defined in the `RealtimeSystems` class returns true. An application can specify an asynchronous event handler to run in response to the firing of the above events. The assumption is that the application will maintain its own list of which schedulable objects are mapped to which processors (logical or physical). It will then undertake whatever reconfiguration it deems appropriate.

The default affinity can be set at run-time and is scheduler dependent, and must be documented. The affinity of a specific schedulable object can be set.

4.3. The cost enforcement model

The RTSJ manages the CPU time allocated to a schedulable object via two optional complementary mechanisms. The first is via the `cost` parameter component of the `ReleaseParameter` class. If supported, this requires a schedulable object to consume no more than this value for each of its releases. In an SMP systems (assuming the processors execute at approximately the same speed), this approach easily scales from the single processor model.

The second, more problematic, mechanism is that provided by `ProcessingGroupParameters`, which allows a group of schedulable objects to be allocated a utilization. Whilst conceptually, this could be applied to multiple threads from the same group executing in parallel with global scheduling, it is not clear that the general model should be applied to a partitioned system. Indeed, the implementation challenge of detecting budget overrun may make the whole approach untenable. Instead, it is necessary to place a constraint that all schedulable objects allocated to a processing group must be allocated to the same processor.

The alternative is to provide a mechanism to set the affinity of a processing group. At each period, the processor would be allocated to the group and maintained on that group for the duration of the release. This would override any individual schedulable object's affinity.

A third alternative is to extend `ProcessingGroupParameters` so that they contain a CPU budget per processor. A schedulable object can only be allocated a processing group if its affinity set intercepts with the `ProcessingGroupParameters` affinity set.

4.4. The affinity of interrupts(happenings)

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.

The current RTSJ specification provides no mechanism that would allow the programmer to tie interrupts and their handlers to particular processors. Asynchronous event handlers are the main RTSJ mechanism for handling interrupts. These should be viewed as third-level interrupt handlers. The first-level interrupt handlers are the code that the platform executes in response to the hardware interrupt (or signal). First level interrupts are assumed to be executed at an execution eligibility (priority) dictated by the underlying platform. The second-level interrupt handler is the code that the JVM implements as a result of being notified by the first-level interrupt handler that the interrupt is targeted at the RTSJ application. Its job is to find the asynchronous

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

event and then release the associated handlers (third-level interrupt handler).

Currently there is a proposal within JSR 282 to allow priorities (or more generally, exceution eligibilities) to be given to second-level interrupt handlers. The main points are as follows:

1. Each asynchrnonous event that has been associated with an external event (called a "happening" in the RTSJ) is given a execution eligibility. By default this is the highest in the system. To allow the programmer to set the execution eligibility the following method is added to the asynchronous event class:

   ```
   public SchedulingParameters bindTo(String happening,
          Eligibility : SchedulingParameters);
   ```

   The method returns the current execution eligibility associated with the asynchronous event. If the asynchronous event is associated with more than one happening, the execution eligibility is the highest value. This is reflected in the returned SchedulingParameters object.
2. When an event is fired (by software), the firing code occurs at an eligibility of the firing schedulable object.
3. When an interrupt occurs, the second level interrupt handler (once it has identified the associated asynchronous event) executes at the defined execution eligibility for that event.

It would be trivial to modify the above method to allow the second-level interrupt handling code to be given an affinity.

4.5. Failure Model

The RTSJ has no explicitly-defined failure model. In a multiprocessor environment, if a processor fails and the platform cannot transparently recover, the real-time JVM ends abnormally (with assumed fail-stop semantics). Any recovery must be performed outside of the JVM. This is because a processor failure can leave the application and JVM in an inconsistent state (e.g. with a corrupt heap) from which it is unlikely to be able to recover.

5. Conclusions

As the complexity of real-time and embedded systems continues to grow, there is little doubt that they will in future rely on multiprocessor/multicore platforms. As with all resources, it is necessary to allow the real-time programmer to have the ultimate control over their use. Hence, facilities must be made available at the operating system level and then via language-level APIs. Whilst, the RTSJ tries not to stand in the way of multiprocessor implementations it does not facilitate control over the resources in a portable way. This paper has considered how to address these concerns in future versions of the specification.

Acknowledgements

The author gratefully acknowledges the discussions that he has had with the members of JSR 282, the members of JSR 302, Alan Burns, Ted Baker and Sanjoy Barauh.

APPENDIX – APIs

In this appendix example APIs are given that address the issues raised in this paper.

The following methods are added to the `RealtimeSystems` class.

```
package javax.realtime;
public class RealtimeSystems {

  /**
   * @return - True is the system supports affinity
   */
  public static boolean setAffinitySupported();


  /**
   * @return - True is the system supports notification
   *  following a change of the processor set allocated
   * to the program.
   */
  public static boolean affinityChangeNotification();

  /**
   * The async events that are fired as a result of the
   * processor set changes
   */
  public static AsyncEvent ProcessorRemoved,
                           ProcessorAdded;


  /**
   * @return A Bitset object describing the processors
   * available to the real-time JVM. The length of the bitset
   * is the number of processors on the system.
   * Of these processors, the JVM will know which
   * processors have been allocated to it (either
   * logical processors or physical processors
   * depending on the operating system).
   * Each of the available processors is set to one in
   * the bit set. Hence, the cardinality of the bit set
   * represents the number of processors
   * that the JVM thinks are currently available to it.
   * The returned object is a new object that is
   * allocated in the current memory area.
   */
  public static java.util.Bitset availableProcessors();


  /**
   * Sets the default affinity for heap using
   * schedulable object.
   * @param Processors - A bit set giving the
   *   default affinity
   * @return The old default affinity
   * @thows ProcessorAffinityException if one or
```

```
   *  more of the indicated processors has not been
   *  allocated to the real-time JVM
   * @throws UnsupportOperationException if not
      supported
   * There is no association maintained between the
   * parameter passed and the default. i.e. copy
   * semantics - changes the parameter object at a later
   * stage will NOT result in a change of the default.
   * The returned object is a new object that is
   * allocated in the current memory area.
   */
  public final static java.util.BitSet
    setDefaultAffinity(java.util.BitSet Processors)
          throws ProcessorAffinityException;


  /**
   * Sets the default affinity for no heap schedulable
   * objects.
   * @param Processors - A bit set giving the default
   *  affinity
   * @return The old default affinity
   * @throws IllegalArgumentException if size of the
   *  given given bitset does not match the current size
   *  of the bitset returned from availableProcessors
   *  or if the given bitset is null.
   * @throws ProcessorAffinityException if one or
   *  more of the indicated processors has not been
   *  allocated to the real-time JVM
   * @throws UnsupportOperationException if not supported
   * There is no association maintained between the
   * parameter passed and the default. i.e. copy
   * semantics - changes the parameter object at a later
   * stage will NOT result in a change of the default.
   * The returned object is a new object that is
   * allocated in the current memory area.
   */
  public final static java.util.BitSet
              setDefaultNoHeapAffinity(
          java.util.BitSet Processors)
          throws ProcessorAffinityException;
}
```

The following methods are added to the `RealtimeThread` class.

```
package javax.realtime;
public class RealtimeThread implements Schedulable {
  ...

  /**
   * Sets the affinity of the real-time thread
   * @param Processors - A bit set giving the new affinity
   * @return The old affinity
   * @throws IllegalArgumentException if size of the
   *  given given bitset does not match the current
   *  size of the bitset returned from
   *  availableProcessors or if the given bitset is null.
   * @throws ProcessorAffinityException if one or more
   *  of the indicated processors has not been allocated
   *  to the real-time JVM
   * @throws UnsupportOperationException if not
   *         supported
   * There is no association maintained between the
   * parameter passed and the default. i.e. copy
   * semantics - changes the parameter object at a later
   * stage will NOT result in a change of the default.
   * The returned object is a new object that is
   * allocated in the current memory area.
   * Changes only occur when the setAffinity method
   * is called. The actual affinity will be changed
   * between the time the thread finishes its current
   * release and the time it starts it next release.
   * It must be complete by the time the next release
   * starts.
   *
   */
public java.util.Bitset setAffinity(
            java.util.BitSet Processors)
       throws ProcessorAffinityException;

  /**
   * @return The last bitset that was set by a call
   * to setAffinity (or the default if there was no call).
   * The returned object is a new object that is
   * allocated in the current memory area.
   * @throws UnsupportOperationException if not supported
   */
public java.util.BitSet getAffinity();
```

In `AsyncEvent` class, the following method is added:

```
package javax.realtime;
public class AsyncEvent {
  public SchedulingParameters bindTo(String happening,
                   Eligibility : SchedulingParameters,
                   java.util.BitSet Processors);
}
```

In `BoundAsyncEventHandler` class, the following additions:

```
package javax.realtime;
public class BoundAsyncEventHandler
             implements Schedulable {
  ...

  /**
   * Sets the affinity of the bound event handler.
   * Same functionality as the real-time thread version
   *
   */
  public java.util.Bitset setAffinity(
       java.util.BitSet Processors)
            throws ProcessorAffinityException;

  /**
   * Gets the affinity.
   * Same functionality as the real-time thread version
   */
  public java.util.BitSet getAffinity();
```

References

[1] T.P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems*, 32(1-2):41–71, 2006.

[2] T.P. Baker. Global versus partitioned scheduling in multiprocessor systems, private communication, 2006.

[3] Linux Manual Page. sched_setaffinity(), 2006.

[4] QNX Neutrino RTOS 6.3.2., www.qnx.com/download/download/16841/multicore_user_guide.pdf, 2007.

[5] M. Gonzalez Harbour. Supporting SMPs in POSIX, private communication, 2006.

[6] Open Group/IEEE. The open group base specifications issue 6, ieee std 1003.1, 2004 edition. IEEE/1003.1 2004 Edition, The Open Group, 2004.

[7] R. Rajkumar. Synchronization in Real-Time Systems: A Priority Inheritance Approach, Kluwer Academic Press, 1991.