Verifying Nested Lock Priority Inheritance in RTEMS with Java Pathfinder

Saurabh Gadia[1], Cyrille Artho[2,3], and Gedare Bloom[4]

[1] University of Southern California, Los Angeles CA, USA
[2] National Institute of Advanced Industrial Science and Technology, Osaka, Japan
[3] KTH Royal Institute of Technology, Stockholm, Sweden
[4] Howard University, Washington DC, USA

Abstract. Scheduling and synchronization algorithms for uniprocessor real-time systems benefit from the rich theory of schedulability analysis, and yet translating these algorithms to practical implementations can be challenging. This paper presents a Java model of the priority inheritance protocol for mutual exclusion, as implemented in the RTEMS open-source real-time operating system. We verified this model using Java Pathfinder to detect potential data races, deadlocks, and priority inversions. JPF detected a known bug in the RTEMS implementation, which we modified along with the Java model. Verification of the modified model showed the absence of data races, deadlocks, and established nine protocol-specific correctness properties.

Keywords: Java Pathfinder, RTEMS, priority inheritance

1 Introduction

Real-time application correctness depends on a bound on the amount of interference that high-priority tasks can cause to lower-priority tasks. When a high-priority task preempts the low-priority task and executes, the response time of the low-priority task is delayed by the preemption. Schedulability analysis considers the interference caused by such preemption: given a set of tasks, their execution times, job releases and deadlines, and priorities, one may calculate whether the tasks will be schedulable under a given scheduling algorithm. Two key assumptions made during schedulability analysis is that tasks are preemptable and do not share resources.

In case tasks share resources (e. g., shared memory) that require synchronization, the inclusion of critical sections complicates the schedulability analysis. The usual approach to create a critical section is with a semaphore or mutex lock, which leads to a priority inversion problem in which a low-priority task holding a lock interferes with any higher-priority tasks waiting on the lock, until the low-priority task eventually releases the lock. The usual assumption in real-time systems is that a lock holder can hold the lock no longer than its worst-case execution time, and therefore the lock is released eventually. However, if middle-priority tasks preempt and starve the low-priority task, the high-priority task may be blocked indefinitely since the low-priority task continues to hold the lock while the middle-priority tasks execute. Priority inversions are solved by using the priority inheritance protocol (PIP) or priority ceiling protocol (PCP) [15].

PIP works by promoting the priority of a lock holder to that of the highest-priority task waiting for the lock. Hence, when a task fails to acquire a lock, the lock holder will inherit the task's priority until the lock is released. PIP ensures that a middle-priority task cannot indefinitely block higher-priority tasks by starving low-priority lock holders. The lock holder's priority is restored to the previous value when releasing the lock. Although PIP works well for single lock acquire and release, some real-time applications require mutual exclusion for multiple resources at a time. Hence, multiple locks are acquired, one for each resource. We say these locks are *nested*. The proper implementation of PIP with nested resources requires that each time a lock is released, the lock holder's priority should be changed to that of the highest priority task still blocked on any lock held by the releasing task, or to the lock holder's normal priority, i. e., the priority it held when entering the outermost lock.

RTEMS is an open-source real-time operating system with an implementation of PIP. However, RTEMS implementation of the PIP for nested locks is incorrect. When a nested lock is obtained and priority is inherited, the task's current priority is saved. When the nested lock is released, the current priority is restored without checking what the highest priority is on the outer lock. Therefore, a task could be blocked on the outer lock with a higher-priority than the task that just released the inner lock, thus creating a priority inversion. Currently, this inversion is avoided by retaining the highest inherited priority until all locks are released, which has a different problem in that schedulability analysis needs to account for interference caused by all critical sections of any lower-priority tasks that may outer-nest a lock shared with a high priority task.

In this paper, we present our experience creating and model-checking a Java model of the locking and scheduling algorithms from the implementations in RTEMS relevant to PIP with nested resources. Our model detects priority inversion conditions via exhaustively searching through the lock and scheduler data structures. We passed the Java model through the Java Pathfinder (JPF) model checker [16] to detect potential deadlocks, data races, and priority inversions. After confirming the priority inversion in the existing RTEMS nested resource locking, we model-checked an alternative algorithm, which we then implemented in RTEMS. In our model analysis, we use intelligent pre-processing to reduce the size of the state space by a factor of 158, which made it possible to check the model repeatedly for different versions of our algorithm.

This paper is organized as follows: Section 2 provides the necessary background and shows related work. Section 3 shows how we modeled the relevant parts of the RTEMS kernel for verification with JPF. Section 4 shows the flaw we found in the adaption of PIP in RTEMS, and our fix. The final results of the verification with JPF are shown in Section 5, and Section 6 concludes.

2 Background

2.1 RTEMS

The Real-Time Executive for Multiprocessor Systems (RTEMS) [1] is an open-source real-time operating system (RTOS) that provides essential RTOS services with support for POSIX interfaces. RTEMS is used in particle accelerators, satellite instruments, medical devices, military systems, robotics, and other data acquisition and real-time control applications. RTEMS has been an open-source project since 1988 and is widely used in academic, government, and private sectors around the world.

2.2 Model checking

Model checking is a technique to analyze a formal description of a system (the model) against all possible outcomes, starting from a given initial state. The system to be analyzed is usually represented by a transition system [4].

A *model checker* is a tool that verifies the model against given properties [4]. Traditionally, model checking has been used to verify hardware or protocols. Models are usually described in a domain-specific language, and properties are often expressed in temporal logics such as linear temporal logics [14]. Model checkers may explicitly construct the entire state space in memory [7] or use a symbolic representation of multiple states as a set of states [11].

2.3 Java Pathfinder

In contrast to traditional model checkers, *software model checkers* analyze an actual application (as source code or executable) code instead of a model [6, 16].

Java Pathfinder (JPF) is such a software model checker. It implements a Java Virtual Machine (JVM) that is capable of executing the full bytecode instruction set [16], and is designed to explore the full state space of a Java bytecode application. Unlike in a traditional model checker, the state space is not known a priori. Instead, the state space is derived from the execution of the system under test (SUT). The SUT is executed by JPF until an action occurs in which the outcome under different thread schedules may vary. At that point, JPF stores a copy of the full program state comprising memory and thread states, and explores the next available choice. When a given choice leads to the end of program execution, it backtracks to an earlier program states by restoring that state from the saved copy. (JPF uses depth-first search by default; other search strategies are supported as well.) A state in JPF therefore corresponds to a full program state—with the heap and the states of each thread—and a transition corresponds to a sequence of instruction executions by a thread [16].

By default, JPF checks an execution against uncaught exceptions, assertion failures, and deadlocks. It also has a built-in data race detector, which allows a user to find problematic data accesses even if no property checking the output of the program has been written. This is useful because it is difficult to write properties checking the outcome of each operation that may be affected by shared memory access.

For our work, we chose Java Pathfinder because the input language (Java) is much closer to C than the input languages of other model checkers, such as Promela used by SPIN [7]. Furthermore, many locking features used by the RTEMS kernel have close equivalents in Java or JPF, which makes it easier to model RTEMS in JPF than on other platforms (see Section 3).

2.4 Related Work

Klein et al. verified a general-purpose operating system microkernel [8] that includes thread management and many other features. The kernel has been implemented in 10,000 lines of C code; properties were verified using a theorem prover with about 480,000 lines of Isabelle [13] proofs that were developed over ten years [8]. In contrast to that, the core of RTEMS is about 34,000 lines of C code, with the part implementing mutual exclusion weighing in at about 730 lines. Our model is about the size same as that part of the C code (600 lines of code for the model, 130 lines of code for the helper program that generates all test settings), and verification is fully automatic after we apply symmetry reduction to our environment model.

The idea of symmetry-based state space reduction is common [3]. Compared to previous work, we take a staged approach, where we first pre-process the parameters of the model to reduce the state space, and then use these parameters to generate different settings at run-time. Compared to "classical" model checking [4], where a model is expressed in a domain-specific language, we express our model in Java, which is richer than other modeling languages. Verification is performed using Java Pathfinder, which executes the model as program code and generates the state space by exploring different outcomes of non-determinism at run-time [16]. This is different from most other tools, where the state space is generated as a graph structure a priori [4].

Java Pathfinder is typically used to explore different interleavings in concurrent software, to analyze whether functional properties hold for all possible interleavings [16]. For applications written in Java, Java Pathfinder can determine the worst-case execution time (WCET) of a program by assigning a cost to each instruction, and calculating the maximal total cost [10]. This previous work was not applicable to estimate the worst-case execution time in the RTEMS kernel, as it would have required a complete model of the kernel in Java, together with accurate execution cost weights that reflect the true execution cost of the original RTEMS code on different platforms. Our work checks the correctness, but not WCET, of the priority inheritance protocol in RTEMS.

Lui Sha et al. discusses that synchronization primitives can lead to uncontrolled priority inversion problem [15]. They showed that two priority inheritance class protocols called basic priority inheritance protocol and priority ceiling protocol solved this priority inversion problem. This original publication does not explicitly mention the priority inversion problem caused in basic priority inheritance protocol if a task inheriting priority is owner of more than one mutex that we found in RTEMS. As a result, we cannot rule out that a direct application of that algorithm may have the same issue.

Linux makes use of data structure called plist—'priority sorted linked list'—for implementation of real time mutex design. Every task in Linux has pi_list data structure that stores all top waiters of the mutexes that are owned by the task. Whenever a task releases any mutex, it always ensures that its priority is set to the top priority waiting task in its pi_list. This way it rules out the problem of priority inversion in case of task owning multiple mutexes.

3 Modeling RTEMS Locks and Scheduling in Java

This section describes the Java model of locking in RTEMS [5].

3.1 Mapping RTEMS kernel constructs to JPF

The POSIX threads (Pthreads) standard is a widely used standardized interface that provides concurrency primitives, in particular locking, thread creation and control, and the use of condition variables and signaling [12]. Java has been designed to allow a virtual machine to implement the thread constructs in Java readily using Pthreads; each basic Java concurrency feature can be mapped to Pthreads [2].

Table 1. RTEMS kernel data structures and constructs in Java and JPF.

RTEMS resource	Java/JPF equivalent
Lock usage	`synchronized` block usage
Thread signaling	`wait` and `notify`
Priority queue	`java.util.PriorityQueue`
Global scheduler lock	`gov.nasa.jpf.vm.Verify.beginAtomic` and `endAtomic`

The RTEMS kernel also exhibits similarities to the Pthreads interface, and by extension Java concurrency features: see Table 1 for a high-level mapping. We employ nested locking and unlocking implemented in a straightforward way using `synchronized` blocks in Java; non-nested locking could be supported using extra libraries [9]. Thread signaling and condition variables use the same semantics as in POSIX. Furthermore, priority queues in the RTEMS kernel are modeled using priority queues from the Java base library.

For uniprocessor systems, RTEMS has a mechanism that temporarily disables the scheduler making a block of code behave atomically. Java has no direct construct for such scheduler disabling, and while a global lock can be used, it only guarantees mutual exclusion with respect to other global locks. Fortunately, Java

Pathfinder has a construct that provides atomic sections: `Verify.beginAtomic` and `endAtomic`. These two functions are not available in standard Java.

Another modeling problem is that the Java scheduler does not obey thread priorities strictly, but thread priorities in RTEMS are strict, and a higher-priority thread is always scheduled before lower-priority threads. Although Java Pathfinder allows a user to provide a custom scheduler, which we could have used to model the RTEMS scheduler in addition to its locking implementation, we chose not to do so for two reasons. First, writing a custom scheduler in JPF is more difficult than writing a model in Java. Second, we wanted our lock model to be correct under any scheduler, not just under the current scheduler used by RTEMS.

3.2 Design of the lock model

We designed and implemented a Java model of the locking and scheduling algorithms of RTEMS so that JPF could be used to model check the current and alternative solutions for PIP. Our scheduler model uses a task control block (TCB) that inherits from the Java Thread Class and adds two priority fields, for the initial and current priority, and a linked list to track the mutex locks held by the thread.

To model mutex locks in Java we created a Lock class that uses Java's `synchronized` and JPF's `Verify.beginAtomic` and `Verify.endAtomic` to construct critical sections. The Lock class also adds a `validator` routine that executes on every mutex release to check whether there exists any priority inversion by iterating through all the remaining mutex locks held by the releasing thread checking that it has a higher priority than all threads waiting on any remaining mutexes it holds.

In the course of our work, we experimented with several variants of possible implementations of PIP:

1. A model using a global lock. Using a global lock facilitates a correct implementation at the expense of performance. We used this model to focus on correct thread priorities in an initial version of the model.
2. A uniprocessor model using a global scheduler lock (see above), which is derived from the first model.
3. A model using multiple fine-grained locks, allowing for more parallelism and thus better performance on a real system.

3.3 Test harness

Java Pathfinder explores the state space of a program by starting from its `main` method, as in a normal execution under the Java VM. To analyze the implementation of a multi-threaded program, and our lock implementation in particular, we need a test harness.

Our test harness creates three threads with a given thread priority, each of which locks and unlocks two locks. Each lock is chosen from three distinct candidate locks.[5] We test reentrant locking by assigning the same candidate lock to multiple slots in the same test thread. Each thread is also assigned a priority chosen from three values, representing high, medium, and low priority. We currently do not test non-nested locking and unlocking, which could however be achieved by permuting the order of unlock operations.

```java
public class TestThread extends RTEMSThread {
  Lock availableLocks[] = {createLock(0), createLock(1), createLock(2)};

  public TestThread(int idx[], int priority) {
    super(priority); // initialize thread with given priority
    locks = new Lock[idx.length];
    for (int i = 0; i < idx.length; i++) {
      locks[i] = availableLocks[idx[i]]; // use given locks permutation
    }
  }

  public void run() {
    for (int i = 0; i < idx.length; i++) {
      locks[i].lock();
    }
    for (int i = idx.length-1; i>= 0; i--) {
      locks[i].unlock();
    }
    assert currentPriority==realPriority;
  }
}
```

Fig. 1. Test thread using a given priority and nested locks.

The test harness includes a main method that parses arguments from the command line, which indicate the lock indices and thread priorities. Because some types of cyclic deadlocks require three threads, we wanted to simulate at least as many threads in our model. In doing, so, we ran into the state space explosion problem: If each thread non-deterministically uses two locks (out of three candidate locks), and a non-deterministic priority setting taken from three possible values, we have a total of $(3^2)^3$(2 locks per thread) $* \, 3^3$(priorities) $=$ $3^9 = 19683$ combinations.

Encoding all these options as non-deterministic choices would be extremely inefficient for the following reasons:

1. A lot of symmetries exist in the state space, some of which would not be recognized by JPF and explored redundantly.
2. Exploring the entire state space at once increases memory usage and may cause JPF to run out of memory or trigger garbage collection excessively often.

[5] This design guarantees a certain degree of overlapping lock usage between threads, without which there would be no need for mutual exclusion.

3. Debugging a failed test (from a faulty model) is more difficult because the error trace by JPF does not show the lock indices or thread priorities as such. They could be made visible in other ways, through listeners or printing them on the screen, but the latter option would print a lot of clutter during the state space exploration.

3.4 State space preprocessing

The full state space is too large to be explored by JPF if equivalent configurations are not taken into account. We present an algorithm to remove redundant lock set configurations, and show how the number of thread priority configurations can be minimized.

Lock sets. We define A to be the alphabet of lock indices; in our case $A = 0, 1, 2$. A *lock set configuration* is a list of n elements, each being a sequence of m lock indices.

To reason about lock indices, we observe the following properties:

1. Locks are symbolic objects. Any configuration l' where all lock indices in l are replaced with a permutations of the indexes in A, yields a heap structure that is isomorphic to l. For example, $(00, 00, 01)$ and $(11, 11, 10)$ are isomorphic.
2. Between threads, permutations of lock index sequences are also isomorphic; e. g., $(00, 00, 01)$, $(00, 01, 00)$, and $(01, 00, 00)$ are isomorphic.

Algorithm 1 computes the set of all relevant lock permutations, based on these two observations. It starts by initializing the output sets F and B and generating all possible isomorphic mappings I. For three indices, six isomorphisms exist: $I = \{\{0 \to 0, 1 \to 1, 2 \to 2\}, \{0 \to 1, 1 \to 2, 2 \to 0\}, \ldots\}$ The algorithm reduces the set of all possible permutations C of lock indices, to topologically distinct ones.

Based on property 1, the algorithm then proceeds to generate all morphisms M from C (step 4a). Property 2 allows us to ignore different permutations of lock index sequences between threads (step 4b); the sequences are filtered by sorting. For example, the sorted list of subsequences in $(12, 01, 00)$ is $(00, 01, 12)$. These two steps are sufficient to reduce the set of 729 lock permutations to only 31 truly distinct settings.

However, some lock set configurations contain a cyclic dependency between locks. For example, if thread t_1 owns lock a and tries to obtain lock b, and thread t_2 owns lock b and tries to obtain lock a, a deadlock occurs. The deadlock is due to the cyclic lock dependency between the threads. A test using such a configuration may deadlock, and if JPF is used, it will always find and report such a possibility. We can either ignore such cases or ensure that JPF actually detects a deadlock. To distinguish between “good” (deadlock-free) and “bad” (deadlocking) cases, we check the lock configuration for such cyclic dependencies (step 5 in Algorithm 1). This splits the set of 31 configurations into 25 deadlock-free and 6 deadlocking configurations (see Table 2).

```
1. Let $F$ be the set of final candidates, and $B$ be the set of “bad” lock permutations
   that result in a deadlock.
2. Generate the set $I$ of all isomorphic mapping functions (permutations of symbols
   identifying locks) $p_i \in I$ with $i_k = A \mapsto A$ for all lock indices in $A$.
3. Generate all lock permutations, called the *candidates* $C$.
4. For each candidate $c \in C$:
   (a) Generate all isomorphic variants $M$ of $c$, for each permutation in $I$:
       $\forall iso \in I, M = M \cup iso(c)$.
   (b) For each isomorphic candidate $m \in M$, sort the lock index sequences of all
       threads: $S = \text{sorted}(M)$.
   (c) We add all items in $S$ to $F$: $F = F \cup S$.
5. For each unique permutation $f \in F$, check if the lock indices form a cycle between
   all threads; if so, add that permutation to $B$: $\forall f \in F, B = B \cup f$ if $\text{cyclic}(f)$.
6. Output the set of “good” candidates, $F \setminus B$, and “bad” candidates $B$.
```

Algorithm 1. Algorithm to compute all relevant lock permutations.

Thread priorities. We also consider the impact of different thread priorities on the outcome. As only the relative priority between all three threads matters, we consider only these four cases:

1. All threads have the same priority.
2. Two threads have the same priority, one has a lower priority than the others.
3. Two threads have the same priority, one has a higher priority than the others.
4. All threads have a different priority.

We implement this as a non-deterministic choice between four settings that reflect these cases, as opposed to a non-deterministic priority choice for each thread in isolation. This reduces the thread priority state space from $3^3 = 27$ settings to just four.

The combined state space reduction from both optimizations is from 19863 to $31 * 4 = 124$ configurations, a reduction of almost 160 times. The fact that configurations for Java Pathfinder can be parameterized on the command line makes it easy to generate the parameter state space with a preprocessor, and supply it to Java Pathfinder in a second phase.

3.5 Properties

By default, JPF reports a deadlock where the program cannot proceed with execution, such as when multiple threads have a cyclic lock dependency. It can also be configured to report data races. A data race exists if at least two threads access the same memory location without mutual exclusion, and at least one of these accesses is a write access. We used deadlock and data race detection along with model-specific properties in our verification.

Specific properties are encoded as assertions (safety properties) in our model [5], and cover the following:

Property 1. The priority of a thread waiting for a lock corresponds to its actual (original) priority.

Property 2. A thread is in the correct state when acquiring a new mutex.

Property 3. A newly acquired mutex is not held by another thread, and its lock count is zero.

Property 4. The lock count of a lock being released is greater than zero. After a lock is released, there is no holder registered anymore for it.

Property 5. If there is another thread waiting on a just-released lock, that thread must be in the waiting state.

Property 6. The thread releasing a lock must contain a matching lock entry at the head of the list maintained in that thread's TCB.

Property 7. The thread releasing a lock must not contain any higher-priority threads linked from the list of lock entries in the thread's TCB.

Property 8. The promotion of a thread's priority is caused by a different thread.

Property 9. The priority of a thread is correctly reverted to the original priority after all locks have been released.

4 Fixing PIP in RTEMS

Priority inversion occurs if a higher-priority task is blocked by a lower-priority task. Ideally, a higher-priority task should be blocked no longer than the time for the lower-priority task to complete its critical section. Lui Sha et al. demonstrated that practically this blocking period of higher-priority task can be arbitrarily long and unpredictable [15]. They showed that two priority inheritance class protocols, basic PIP and priority ceiling protocol, can rectify uncontrolled priority inversion. We extend basic PIP for the case when a task inheriting priority is the owner of more than one mutex, which we found in RTEMS, and we propose an algorithm to solve uncontrolled priority inversion problem in this case.

4.1 RTEMS data structures involved in PIP

In RTEMS, associated with each mutex is a linked list *CORE_mutex_order_list*, which contains *priority_before*, a field to store the priority of the acquiring task. This field is used to restore the task's priority to what it was before acquiring that mutex, in case the priority of the task is temporarily increased due to the PIP. Each task control block (TCB) stores a last-in first-out (LIFO) linked list of acquired mutexes, which is the expected order of lock release. This is a doubly linked list, *Chain_Control*, consisting of nodes of type *Chain_Node* (see Fig. 2).

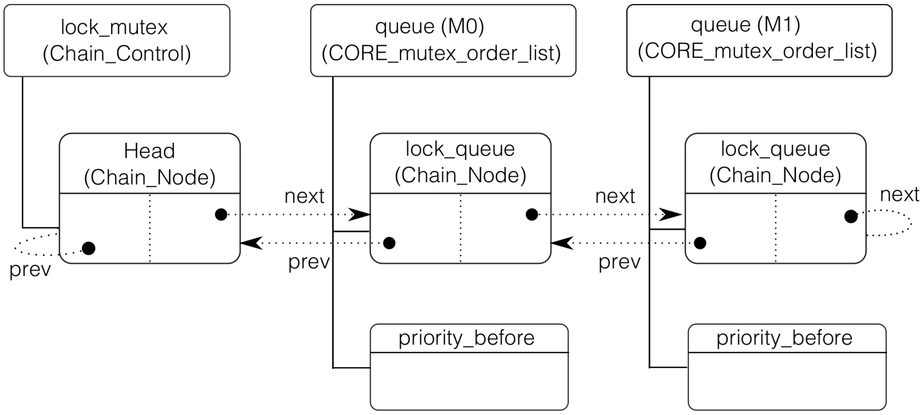

Fig. 2. Data structures linking the mutexes with the lock queues.

4.2 Uncontrolled priority inversion problem for PIP in RTEMS

The following example demonstrates the uncontrolled priority inversion problem:

1. Consider three tasks T_0, T_1, and T_2 in descending order of priority with T_0 having highest priorities of 0 and T_2 having lowest priority of 2.
2. Initially, we only have task T_2 executing in our system. T_2 acquires mutex m_0 followed by m_1 to access some shared data structure. The linked list of mutexes in T_2's TCB contains m_0 and m_1 with associated priority 2.

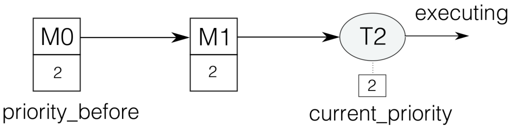

Fig. 3. Initial system state

3. Task T_0 is created and being a higher-priority task it preempts task T_2. T_0 attempts to acquire m_0 and hence the PIP will promote the priority of T_2 to be that of T_0, i. e., 0. This is the classic example of basic PIP rectifying priority inversion problem. This system state is free from priority inversion and deterministic as we are certain that task T_0 will be waiting till task T_2 releases m_0.
4. Another task T_1 is created, with medium priority. It will be in waiting state as task T_2 has the highest priority. When T_2 releases m_1, the priority stored in the mutex data structure for m_1 is written into the TCB of T_2, restoring T_2's priority to 2. Task T_1 preempts task T_2. We are now uncertain of blocking

```
function UpdatePriority(holder, queue, priority)
    // holder is the TCB for the owner of the mutex being acquired
    // queue is the CORE_mutex_order_list of the mutex
    // priority is the priority of the thread trying to acquire the mutex

    head_node ← (&holder→lock_mutex)→Head
    next_node ← (&queue→lock_queue)→next
    change_priority ← True
    while next_node ≠ head_node do
        queue ← next_node→CORE_mutex_order_list
        if queue→priority_before ≤ priority then
            change_priority ← False
            Break
        end if
        queue→priority_before ← priority
        next_node = next_node→next
    end while
    return change_priority          ▷ if True, then holder thread priority is checked
end function
```

Algorithm 2. Algorithm for updating priority

period of task T_0 (see Fig. 4) and thus the system is in uncontrolled priority inversion state.

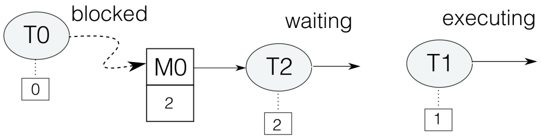

Fig. 4. Uncontrolled priority inversion

4.3 Solution to uncontrolled priority inversion

Avoiding uncontrolled priority inversion calls for more intelligence when restoring the priority of a task. Algorithm 2 imparts this intelligence when a task attempts to acquire a busy mutex. Whenever a task attempts to acquire a busy mutex, this task may update the holder task's priority and the priorities of mutexes held by that holder. Updating the priority of holder is done as usual for PIP. The updates of the held mutexes occur by traversing the linked list of mutexes stored in the holder's TCB. If the recorded priority of a mutex is lower than the priority of the acquiring task, then the recorded priority is updated. The traversal stops at an equal or higher priority, or at the head of the list.

Applied to the above example, Algorithm 2 works as follows:

1. When task T_0 attempts to acquire mutex m_0, it traverses the acquired mutex list of holder task T_2. Traversal moves to the mutex next in the list from m_0, which is m_1, towards the head of the list. The recorded priority of m_1 is 2, lower than T_0's priority 0, so is changed to 0 before going to the next mutex in the list. The head of the list is reached, therefore the priority of T_2 is compared with that of T_0 and is boosted to 0, and the algorithm is finished. T_2 will resume executing at its new, higher priority (see Fig. 5).

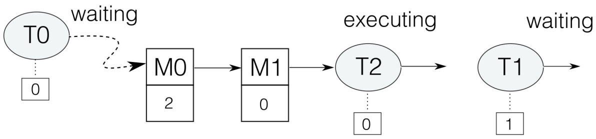

Fig. 5. Priority update as per proposed algorithm

2. When T_2 releases m_1, the priority stored in the mutex data structure for m_1 is written into the TCB of T_2, restoring T_2's priority to 0. At this point, task T_1 is waiting as T_2 still has the highest priority in the system. This way our algorithm ensures there is no priority inversion in the system (see Fig. 6).

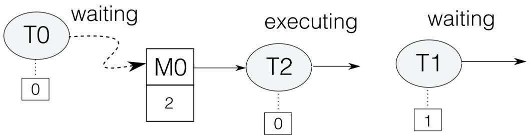

Fig. 6. Deterministic system with no priority inversion

3. Note that if T_2 was blocking on another mutex (see Fig. 7), it would be re-blocked after potentially boosting priority of that mutex's owner through a transitive call to Algorithm 2. Hence, we always have a system which is free from priority inversion and in deterministic state.

5 Model-Checking Results using JPF

We used Java PathFinder version 8.0, rev. 28, to analyze the Java models of RTEMS locking, after eliminating redundant configurations in the parameter state space by preprocessing it (see Section 3). We ran the experiments on a Mac Pro with two 3.7 GHz quad-core Intel Xeon E5 CPUs. This allowed us to use multiple cores at once, speeding up the verification of 31 different lock set

Table 2. Model checking time, number of states, and number of instructions for all distinct scenarios. Lock usage is shown as a triple of sequences (of length two) of lock IDs. These IDs correspond to the two locks used by the test harness of each thread. Each lock configuration was tested for all relevant priority settings.

Deadlock-free configurations

Lock configuration	Time	Number of states	Number of instructions
(00,00,00)	00:05:51	3,597,839	39,134,614
(00,00,01)	00:08:31	5,359,776	53,499,140
(00,00,10)	00:29:50	20,530,383	158,243,065
(00,00,11)	00:22:19	15,321,110	117,084,710
(00,00,12)	00:31:16	21,614,670	159,649,787
(00,01,01)	00:10:26	7,277,751	71,695,034
(00,01,02)	00:10:21	7,277,751	71,695,034
(00,01,11)	00:25:07	17,704,921	137,981,768
(00,01,12)	00:37:20	26,655,214	198,517,948
(00,01,20)	00:46:39	32,883,641	249,273,627
(00,01,21)	00:49:55	35,524,795	262,654,448
(00,01,22)	00:34:44	23,805,000	181,621,827
(00,10,10)	00:27:31	19,231,201	164,864,689
(00,10,12)	00:38:42	27,290,669	221,112,177
(00,10,20)	02:06:17	93,616,077	713,877,172
(00,10,21)	01:41:56	72,693,793	556,663,234
(00,10,22)	01:33:02	66,929,870	529,446,674
(00,11,22)	01:27:25	61,758,697	504,124,972
(00,12,12)	00:45:33	32,800,449	266,311,244
(01,01,01)	00:11:18	9,172,281	90,384,827
(01,01,02)	00:13:13	9,327,571	91,742,294
(01,01,12)	00:41:50	29,339,526	222,928,606
(01,01,20)	00:52:03	45,084,155	339,817,731
(01,01,21)	01:04:19	50,192,733	371,521,061
(01,02,12)	00:40:47	38,795,617	290,831,840

Deadlock-prone configurations

Lock configuration	Time	Number of states	Number of instructions
(00,01,10)	00:00:01	8,486	270,677
(00,12,21)	00:00:01	8,486	270,677
(01,01,10)	00:00:01	8,486	271,022
(01,02,10)	00:01:37	940,738	7,995,738
(01,10,20)	00:02:21	1,402,381	11,219,277
(01,12,20)	00:03:22	2,062,672	16,476,034

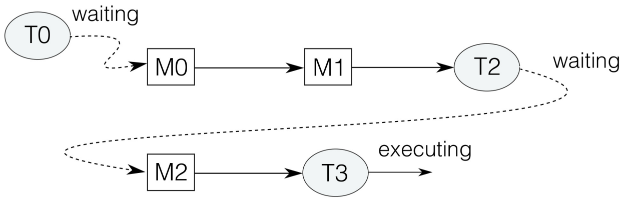

Fig. 7. Nested priority inversion

configurations. Table 2 shows the results of the experiments on the final, correct version of the model.

In configurations that do not cause deadlocks, JPF has to explore the entire state space of the model. Configurations with few locks preclude much possible parallelism, and lock priority changes, in the behaviors. Their state space is therefore much smaller than the state space of more complex settings. Because of this, verification runs for a given lock set configuration ranges between barely six minutes and over two hours; the number of explored program states, and executed bytecode instructions, is proportional to the analysis time (see Table 2, top). Deadlock-prone configurations have at least one possible interleaving that leads to a deadlock due to a cyclic lock dependency. JPF sometimes finds such a deadlock immediately and aborts the search after one second; in other cases the search takes a few minutes, but still finds the bug after only a small fraction of the state space has been searched (see Table 2, bottom).

The results of our experiments confirm that our revised implementation of PIP in the RTEMS kernel is free of data races, deadlocks, and incorrect priority assignments. Deadlock-prone lock usage of application-level tasks is also detected as expected. In total, verification of the improved PIP implementation took 11 hours and 43 minutes of CPU time, which translated to about three hours of real time when running 5–6 instances in parallel on eight cores.

6 Conclusion

Despite the rich, robust theoretical frameworks that have been built around real-time scheduling, the correctness of scheduler design and implementation—especially with synchronization—is challenged by system complexity. In this paper, we have presented a Java model of a real-time operating system's PIP implementation that we model-checked in JPF to look for deadlocks, race conditions, and priority inversions, the latter by way of nine correctness properties that were encoded as assertions in the Java model. Key to the efficient model checking is the state space preprocessing of Algorithm 1, which reduces JPF's search space. JPF found a potential priority inversion, a correctness error, that was known to exist in the C language implementation. We further proposed a fix to the PIP implementation and validated it to be free of potential deadlocks, race conditions, and priority inversions. Future work may consider validating multi-core scheduling algorithms, which are even more complex and less well-understood than the established uniprocessor algorithms.

Acknowledgments This material is based upon work supported by the Google Summer of Code program, the National Science Foundation under Grant No. CNS 0934725, and the Office of Naval Research under ONR Award No. N00014-14-1-0386. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Google, NSF, or ONR.

References

1. RTEMS real time operating system (RTOS), 2016. https://www.rtems.org/.
2. C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto. Model checking of concurrent algorithms: From Java to C. In *Proc. Conf. on Distributed and Parallel Embedded Systems (DIPES 2010)*, volume 329 of *IFIP AICT*, pages 90–101, Brisbane, Australia, 2010. Springer.
3. E. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1):77–104, 1996.
4. E. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press, 1999.
5. S. Gadhia, C. Artho, and D. Ramirez. Model locks with thread priority from RTEMS, 2015. https://github.com/saurabhgadia4/lock-model.
6. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. 10th Int. SPIN Workshop (SPIN 2003)*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.
7. G. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2004.
8. G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, 2014.
9. D. Lea. *Concurrent Programming in Java, Second Edition.* Addison-Wesley, 1999.
10. Gary Lindstrom, Peter C. Mehlitz, and Willem Visser. Model checking real time Java using Java PathFinder. In Doron A. Peled and Yih-Kuen Tsay, editors, *Proc. 3rd Int. Symposium on Automated Technology for Verification and Analysis (ATVA 2005)*, volume 3707, pages 444–456. Springer, 2005.
11. K. McMillan. *Symbolic Model Checking.* Springer, 1993.
12. B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming.* O'Reilly, 1998.
13. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
14. A. Pnueli. The temporal logic of programs. In *Proc. 17th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57, Rhode Island, USA, 1977. IEEE, IEEE Computer Society Press.
15. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, September 1990.
16. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.