Formal Analysis of FreeRTOS Scheduler on ARM Cortex-M4 Cores

Chen-Kai Lin and Bow-Yaw Wang

Academia Sinica, Taiwan
kai.zsv@gmail.com
bywang@iis.sinica.edu.tw

Abstract. FreeRTOS is a real-time kernel with configurable scheduling policies. It is one of the most popular real-time kernel for embedded deviced due to its portability and configurability. We formally analyze the FreeRTOS scheduler on ARM Cortex-M4 processor in this work. Concretely, we build a formal model for the FreeRTOS ARM Cortex-M4 port and apply model checking to find errors in our models for FreeRTOS example applications. Intriguingly, several errors are found in our application models under different scheduling policies. In order to confirm our findings, we modify application programs distributed by FreeRTOS and reproduce assertion failures on the STM32F429I-DISC1 board.

Keywords: model checking · real-time kernel · ARMv7-M architecture

1 Introduction

FreeRTOS is a popular open-sourced real-time kernel [3]. It offers multi-tasking on uni-processor embedded devices. Through multi-tasking, applications can be divided into several simpler tasks sharing processor time. Multi-tasking on the other hand can induce undesirable phenomena such as deadlocks or starvation. It is crucial to prevent such errors in deployment. Multi-tasking errors nevertheless are notoriously evasive. Due to complex interleaving among tasks, a very limited number of system behaviors can be tested. Software testing often fails to detect multi-tasking errors.

Model checking is a formal technique to analyze properties about systems [8]. Behaviors of the system under verification are first specified in a formal model. Model checkers then verify the model automatically with formal properties provided by users. Different from testing tools, model checkers search model behaviors exhaustively. If a deviant behavior is found, it is reported to verifiers. If no deviance can be found, all model behaviors conform to the specified property. The formal model is thus verified.

In this paper, we develop formal models for the FreeRTOS scheduler on ARM Cortex-M4 processors and analyze its properties with the Spin model checker. We build formal models for the ARM Cortex-M4 interrupt handling mechanism based on the reference manual. Optimizing mechanisms such as tail chaining are implemented in our models. On top of our hardware model, we also build

formal models for the FreeRTOS scheduler, thread-safe data structures, and its applications by examining source codes of the FreeRTOS ARM Cortex-M4 port. Particularly, all three FreeRTOS scheduling policies are specified in our behavior models.

With our behavior models for FreeRTOS, it remains to identify formal properties to check. Such properties however can be tricky to find. For formal analysis, high-level informal properties such as deadlocks or starvation need to be specified concretely. In complex systems like FreeRTOS, high-level properties are often asserted with caveats to preclude minor or unrealistic errors. It can be very tedious to specify caveats formally. Moreover, one can not be sure of these caveats without FreeRTOS developers' help. Different developers can also have different views on properties and caveats. Formal properties specified by verifiers themselves can be contrived or even incorrect.

We solve the property specification problem by verifying example applications in the FreeRTOS distribution. In order to highlight FreeRTOS features, developers provide a number of example applications. Most example applications contain assertions to specify expected behaviors during execution. No assertion failure should be observed on any multi-tasking execution. In order to verify whether all assertions hold in all possible executions, we verify them with the Spin model checker. Intriguingly, the model checker reports errors on several example application models.

Assertion errors found in formal analysis need not imply assertion failures in real execution. In order to confirm our findings, we modify FreeRTOS example applications to reproduce error traces found by the Spin model checker. If assertion errors in formal analysis are genuine, we should observe assertion failures on real hardware. Using the STM32F429I-DISC1 board from STMicroelectronics, we successfully reproduce assertion failures in our experiments. We use the remote GDB debugger to confirm failures at intended assertions. All assertion failures require delicate interactions among tasks, the FreeRTOS scheduler, and the ARM Cortex-M4 interrupt mechanism.

This paper is organized as follows. Section 2 introduces the Spin model checker. Section 3 presents analysis methodology. Section 4 briefly introduces our behaviors models for the FreeRTOS scheduler and ARM Cortex-M4 port. Section 5 defines properties of our models. Section 6 reports verification results and discussion. Section 7 gives related works. We conclude this work in Section 8.

2 Background

Model checking is an automatic formal verification technique. In model checking, behaviors of systems under verification are specified as formal models. Properties about systems are also formalized by logical properties about formal models. Given a formal model and a logical property, a model checker automatically verifies the property against the model through mathematical reasoning. If the model is verified, the property holds in the model mathematically. If the model is not verified, the model checker returns a trace to witness the error.

Spin is a model checker designed for analyzing communicating concurrent processes [16]. It supports the Promela (PROcess MEta LAnguage) language to specify system behaviors. A Promela model consists of a set of processes. A process contains a sequence of *commands*. Commands must be *enabled* before executed. Enabled commands in different processes are executed interleavingly. That is, exactly one enabled command is executed at any time. If several commands from different processes are enabled, one of the enabled commands is executed non-deterministically. If there is no enabled command, it is a *deadlock*. The Promela language allows verifiers to specify assertions in processes. An *assertion* command contains a Boolean expression and is always enabled. When an assertion command is executed, it evaluates its Boolean expression. If the value is false, it is an assertion error. Recall that enabled commands are executed non-deterministically. Non-deterministic executions result in different *traces*. An assertion error may appear in some traces but not all of them.

Since traces correspond system behaviors, a deadlock or an assertion error in any trace represent undesirable behaviors. We therefore would like to check if deadlocks or assertion errors occur among all traces. The Spin model checker systematically explores all traces with sophisticated algorithms. If a deadlock or an assertion error occurs in any trace, Spin will find the *error trace* and report it as a witness. If Spin does not find any deadlock or assertion error after exploring all traces, the model is verified.

In addition to assertions, Spin allows verifiers to specify properties with Linear Temporal Logic (LTL) formulas. Particularly, we will use the LTL formula $\Box\Diamond Loc$ where Loc denotes a process location. A trace satisfies $\Box\Diamond Loc$ if it visits the process location Loc infinitely many times. A process satisfies $\Box\Diamond Loc$ if all its traces satisfy $\Box\Diamond Loc$. The formula $\Box\Diamond Loc$ specifies that a process is free of starvation. For instance, let Loc be the location where a process finishes its job. A trace satisfies $\Box\Diamond Loc$ if the process finishes its job infinitely many times and hence without starvation.

3 Methodology Overview

To support different architectures, the FreeRTOS scheduler contains both architecture-dependent and -independent codes. Scheduling policies should be independent of underlying architectures. They provide abstract programming models for applications. Their implementations however necessarily depend on interrupt mechanisms in underlying architectures. Concretely, the FreeRTOS scheduler is called during periodic and sporadic interrupts in the ARM Cortex-M4 port. For exhaustive analysis, it is essential to consider all possible interrupt sequences. Generating such interrupt sequences for testing is infeasible. A more effective technique is required.

We develop a Promela model for the interrupt mechanism on ARM Cortex-M4 processors. Behaviors of ARM Cortex-M4 processors are carefully formalized in our model. Importantly, we model interrupts through non-determinism. Non-

deterministic interrupts allow us to explore interrupt sequences unattainable by testing.

On top of our formal model for the ARM Cortex-M4 interrupt mechanism, we then specify a PROMELA model for the architecture-independent codes in the FreeRTOS scheduler. All three FreeRTOS scheduling policies are specified in our model. Our formal model for the FreeRTOS scheduler on ARM Cortex-M4 processors enables extensive analysis on task synchronization. We moreover build formal models for thread-safe data structures widely used by FreeRTOS applications like queues and locks.

With formal models, we proceed to verify properties about the FreeRTOS scheduler. Although abstract properties such as the absence of deadlock and starvation are easily said, they are not precise enough for formal analysis. Additionally, properties are unlikely to be satisfied without provisions. Without FreeRTOS developers' inputs, contrived or even misleading properties can be verified meaninglessly.

We address this problem by verifying FreeRTOS example applications. Similar to most open-sourced projects, FreeRTOS provides example applications to illustrate its features. These applications contain assertions to specify expected behaviors. These assertions are properties written by FreeRTOS developers. No assertion failure should be observed under all circumstances. In order to verify assertions in FreeRTOS example applications, we build their formal models and check if an assertion error might occur. Intriguingly, several assertion errors were found in our analysis.

It is important to recall that formal models are different from real hardware and software by definition. Assertion errors found on the models do not necessarily correspond to assertion failures on real systems. In order to validate our findings, we examine the error traces found by the SPIN model checker and reproduce them on the STM32F429I-DISC1 board. We use the remote debugger GDB to confirm assertion failures on the ARM Cortex-M4 board. Errors found by our formal analysis are successfully realized on the development board. These assertion failures require intricate interrupt events. They are unlikely to be found by traditional testing.

4 FreeRTOS Scheduler Model

Our goal is to develop PROMELA models for the ARM Cortex-M4 interrupt mechanism, the FreeRTOS scheduler, thread-safe data structures, and example applications. An application has a number of tasks to be executed by the processor. When an interrupt is triggered, its interrupt handler will be executed by the processor. We therefore say a task or an interrupt handler is an *execution unit*. In our model, an execution unit is formalized as a PROMELA process. Commands in a process thus specify the sequential computation of the execution unit.

4.1 Execution Units

In PROMELA, an enabled command is executed non-deterministically among all such commands in all processes. Execution units however need to be scheduled by our formal scheduler model before execution. To this end, we define the global variable `EP` (for Executing Process) and assign each execution unit a unique identification number. Every command in execution units are added with the condition `EP` == *id*. The FreeRTOS scheduler model in turn assigns the variable `EP` to elect next command.

Task Typical FreeRTOS tasks loop forever and never terminate. Their models are PROMELA processes with infinite loops.

ARM Cortex-M4 Interrupt Handler For ARM Cortex-M4 processors, an interrupt is triggered when it is set to the *pending* state. When an interrupt is pending, the processor decides whether the current execution should be interrupted. If the pending interrupt is unmasked *and* has a priority over the current execution, the current execution is interrupted by the pending interrupt and the corresponding interrupt handler is executed.

ARM Cortex-M4 processors optimizes nested interrupts. When returning from an interrupt handler, the processor checks if there is any pending interrupt with a priority over the interrupted execution. If so, the processor proceeds to the pending interrupt rather than the interrupted execution. This optimization is called *tail chaining*.

Similar to task models, our interrupt models are PROMELA processes with infinite loops. Commands within the loops are guarded with the `EP` variable. An iteration in the interrupt model is executed one time whenever the interrupt conditions (pending, masking, and priority) are satisfied. Tail chaining is also specified in our models.

4.2 FreeRTOS Scheduler

The FreeRTOS scheduler provides three scheduling policies. In *cooperative scheduling*, a running task has to yield the processor explicitly. In *preemptive scheduling without time slicing*, a running task can be preempted by tasks with higher priorities. Finally, a task can moreover be interrupted by using up its time slice in *preemptive scheduling with time slicing*. The next execution task is elected by the policy.

In the FreeRTOS Cortex-M4 port, scheduling policies are implemented via two interrupt handlers: PendSV and SysTick. The interrupt handler for the software interrupt *PendSV* elects the next execution task. The PendSV interrupt is triggered whenever a task needs to be rescheduled in all scheduling policies.

The *SysTick* interrupt implements the time slicing policy. It is triggered by a hardware clock periodically. If time slicing is enabled, the SysTick interrupt handler in turn triggers the PendSV interrupt. When the SysTick interrupt handler finishes, the PendSV interrupt handler will be executed directly by tail chaining.

Our PROMELA model follows the FreeRTOS Cortex-M4 port to specify the scheduler in PendSV and SysTick interrupt models. Since the PROMELA language is timeless, our model cannot trigger the SysTick interrupt periodically. Effectively, the SysTick interrupt is triggered arbitrarily in our formalization. The abstraction ensures that all SysTick interrupt sequences in real world are subsumed in our model. If there is any failure among all real interrupt sequences, it will be exposed in our model. However, not all interrupt sequences in our model are real. An error found in the model can be spurious. It has to be validated by corresponding failures in real hardware.

In cooperative scheduling, a task calls the FreeRTOS yield function to release the processor. The yield function triggers the PendSV interrupt to elect a task in the PendSV interrupt handler. It is straightforward to define the yield function in our model.

4.3 Task Synchronization

In addition to task scheduling, the FreeRTOS scheduler also provides basic functions for task synchronization. More concretely, a task can be *delayed* for a specified duration; it can also be *suspended* indefinitely. When a task is delayed or suspended, it is moved to a delay or suspended queue respectively and hence cannot be scheduled for execution. Delayed tasks can be rescheduled when their delay duration expire. Suspended tasks can be rescheduled when they are resumed by the running task.

In the FreeRTOS Cortex-M4 port, basic task synchronization functions are implemented by the PendSV and SysTick interrupt handlers as well. The SysTick interrupt handler checks if any task in the delay queue has expired its duration periodically. If so, the interrupt handler removes such tasks from the delay queue. For suspended tasks, they are removed from the suspended queue when they are resumed by the running task.

If preemptive scheduling is disabled, the running task continues its execution until it yields the processor. If preemptive scheduling is enabled, the PendSV interrupt is triggered when the tasks removed from the delay or suspended queues have a priority over the running task. The FreeRTOS scheduler elects a task with the highest priority for execution. A previously delayed or suspended task will continue its execution; and the running task will be preempted if it does not have the priority.

To handle the degenerative scenario where all tasks are delayed or suspended, FreeRTOS adds an *idle* task. The idle task has the lowest priority and cannot be delayed nor suspended. Instead, it can be configured to yield the processor or not. If the idle task should yield, it yields the processor to the next scheduled task immediately. Otherwise, the idle task loops until it is interrupted. The idle task is also formalized in our model.

Our model for task synchronization mostly follows the FreeRTOS Cortex-M4 port. One major difference between our model and the port is timelessness. Since delay duration cannot be formalized explicitly, we formalize delay duration by a `Tick` counter and a set of `Delay` counters. When a task model is delayed, the

corresponding `Delay` counter is set. When the SysTick interrupt handler model is executed, it increases the `Tick` counter by one. The `Delay` counter of a task model expires if it equals the `Tick` counter. When their counters expire, the SysTick interrupt handler model removes such tasks from the delay queue. All counters are reset when a task model is added to or removed from the delay queue to prevent counter overflow.

4.4 Thread-Safe Data Structures

In addition to task synchronization, FreeRTOS provides thread-safe data structures for message passing and advanced synchronization among tasks. A thread-safe structure consists of its data and a *waiting* task queue. A task modifies a thread-safe structure if its data are ready. Otherwise, the task is *blocked*. When a task is blocked, it is moved to the waiting task queue of the thread-safe structure for specified duration. Different from task synchronization, tasks can be unblocked when its duration expires or data become ready. It is a failure if the duration of a blocked task expires before the data are ready.

FreeRTOS implements thread-safe queues for message passing. A thread-safe queue contains a bounded buffer as its data. The capacity of the buffer is specified by programmers. The buffer is ready when it is neither full for sending nor empty for receiving message. When a task modifies an unready buffer, it is blocked for the specified duration. Consider a sender is blocked by sending a message to a full buffer. When a message is removed from the buffer, the sender will be unblocked immediately. If the buffer remains full when the duration expires, the sender receives a failure. The duration can be zero. It is a failure if the thread-safe queue is currently not ready.

FreeRTOS also offers thread-safe locks. A thread-safe lock uses a counter as its data. Programmers can initialize the counter. If the counter is a positive integer, it means the lock is ready to be taken by tasks. Otherwise, the lock is not ready. Tasks are blocked for a specified duration if they try to take an unready lock. The duration can be zero or a positive integer. It is a failure if the lock remains unready when the duration expires. No task will be blocked when it gives the lock.

Thread-safe structures are widely used in FreeRTOS applications. They are specified in our models. Thanks to our ARM Cortex-M4 interrupt and FreeRTOS scheduler models, our thread-safe structure models mostly follow the FreeRTOS Cortex-M4 port.

4.5 Example Applications

Tasks behave very differently in different scheduling policies. Consider a task which never yields. In preemptive scheduling, such a task can be preempted by other tasks with sufficient priorities. Tasks with higher priorities can still be scheduled for execution. On the other hand, a never-yielding task is never preempted in cooperative scheduling. Since other tasks will not execute, no progress

is made. To ensure progress, FreeRTOS developers make low-priority or never-yielding tasks actively yield the processor when preemption is disabled. Such intricacies can be a burden to programmers.

To help programmers develop their applications smoothly, FreeRTOS provides example applications in its distribution. Particularly, mutexes and semaphores are used for task synchronization. Thread-safe queues are also found in applications for message passing. We have constructed formal models for eight example applications such as *PollQ*, *Semtest*, *BlockQ*, *QPeek*, *Dynamic*, *Countsem*, *Recmutex*, and *GenQTest*. These applications illustrate task synchronization or thread-safe structures in FreeRTOS. They undoubtedly are relevant to the formal analysis of FreeRTOS scheduler in this work.

5 Formal Properties

Formal analysis of FreeRTOS scheduler demands formal properties. Such properties nevertheless are not always obvious to verifiers. High-level properties are too obscure to be formalized. Local properties are often contrived with little relevance. To avoid such pitfalls, we verify assertions annotated by developers in FreeRTOS distribution.

Assertions in FreeRTOS scheduler detect errors in tasks and thread-safe structures at runtime. Task assertions fail when the scheduler resumes a non-suspended task. Assertions in thread-safe structures fail when a mutex is used as a semaphore, or a mutex inherits the priority of a wrong task. Other assertions check the capacity of thread-safe buffers and task priorities. Those assertions are verified in our formal analysis.

Not all assertions are similar however. To organize our presentation, we classify assertions in example applications into two categories. Intuitively, an assertion specifies a safety property if it indicates that a bad event should never happen; an assertion specifies a liveness property if it indicates that a good event should always happen.

5.1 Safety

It is straightforward to specify safety properties with assertions. Programmers only need to write a Boolean expression deemed to be true in an assertion. In FreeRTOS example applications, the following safety properties are found:

(S0) If a task is delayed for synchronization with other tasks, other tasks must finish before the delay duration expires.
(S1) If a task is blocked by thread-safe data, data must be ready when it is unblocked.
(S2) If a task expects a thread-safe data to be ready, the data must be ready.
(S3) Messages received through a thread-safe queue must preserve their order.
(S4) Mutexes and binary semaphores must ensure mutual exclusion of critical sections.

(S5) If a thread-safe lock is taken once, it must be given eventually.
(S6) A low-priority task must inherit priorities when its mutex was taken by tasks with higher priorities and recover its priority after releasing the mutex.

Property (S0) checks if task synchronization is used properly. Property (S1) checks if thread-safe data are implemented correctly. Property (S2) is a special case of property (S1) where the block duration is zero. Property (S3) checks messages are delivered in order by thread-safe queues. Properties (S4) and (S5) check mutexes and semaphores are implemented correctly. Finally, property (S6) checks whether priority inheritance is implemented correctly.

Not all properties are needed in every application. Table 1 shows the safety properties specified in the eight example applications.

Table 1: Properties in FreeRTOS Applications

	Safety							Liveness
	S0	S1	S2	S3	S4	S5	S6	
PollQ	✓		✓	✓				✓
Semtest	✓	✓	✓		✓			✓
BlockQ		✓	✓	✓				✓
QPeek		✓	✓	✓				✓
Dynamic	✓		✓	✓	✓			✓
Countsem			✓			✓		✓
Recmutex		✓			✓	✓	✓	✓
GenQTest		✓	✓	✓	✓		✓	✓

5.2 Liveness

If a task does nothing, no bad event can happen. The task thus satisfies all safety properties. To avoid such vacuous safety, liveness properties are specified. FreeRTOS developers in fact write assertions to ensure tasks are making progress. Concretely, a task maintains a counter which is incremented when a job is finished. The counter is checked by a monitor task periodically. An assertion failure occurs if the counter remains unchanged between checks.

Monitor tasks do not contribute to the computation. They also interfere with the scheduler. Accuracy of checking liveness properties by monitor tasks may be in doubt. Instead of checking progress by monitor tasks, we specify liveness properties by LTL formulas and get rid of monitor tasks in our models. Our analysis hence removes unnecessary disruption in the scheduler. It is more precise than testing liveness with monitors.

Let $Loc_{SysTick}$ be the location triggering the SysTick interrupt and Loc_i the location where task model i finishes its job for $1 \leq i \leq n$. Consider the LTL formula:

$$\Box\Diamond Loc_{SysTick} \rightarrow (\Box\Diamond Loc_1 \wedge \Box\Diamond Loc_2 \wedge \cdots \wedge \Box\Diamond Loc_n)$$

Informally, the formula states that all tasks finish their jobs infinitely many times if the SysTick interrupt is triggered infinitely many times. In our formal models, SysTick interrupts represent the progression of time. If the LTL formula is satisfied in our models, it means that all task models must finish their jobs infinitely often as time progresses. No task can stop making progress indefinitely. The liveness property is required for all FreeRTOS application models in Table 1.

Table 2: Verification Time in Seconds

Applications	Cooperative Scheduling		Preemptive w/o Time Slicing		Preemptive w/ Time Slicing	
	Safety	**Liveness**	**Safety**	**Liveness**	**Safety**	**Liveness**
PollQ	1.6	33.6	3.7	111.0	5.2	154.0
Semtest	0.1	✗	58.0	✗	517.0	✗
*QPeek**	< 0.1	1.3	< 0.1	1.1	< 0.1	✗
*Recmutex**	7.2	305.0	5.6	267.0	39.5	✗
*Countsem**	< 0.1	0.4	< 0.1	✗	3.2	✗
*GenQTest**	0.9	98.3	< 0.1	✗	197.0	✗
*Dynamic**	5.8	131.0	0.1	✗	S0 ✗	✗
*BlockQ**	1.3	210.0	2.2	549.0	S1 ✗	✗

* Some tasks in the application actively yield the processor when preemption is disabled.

6 Verification Results

For each scheduling policy, we use the model checker Spin to verify properties shown in Table 1. The model checker first verifies safety properties in an application model. After checking safety properties, the liveness property is verified on the application model. In our experiments, we use Spin 6.5.2 on an Ubuntu 20.04 server with two 3.2GHz octa-core CPUs and 512GB RAM.

Table 2 gives the verification results for safety and liveness properties in eight applications under three scheduling policies. If all safety properties in an application are satisfied, the verification time (in seconds) is shown. If not, the failed property is shown with a cross mark in the table. For the liveness property, verification time is shown if an application satisfies the property. Otherwise, a cross mark is shown.

6.1 Analysis of Safety Properties

Almost all applications satisfy their safety properties. Spin finishes the verification with at most 40GB of memory in 10 minutes. For failed safety properties, the model checker also reports error traces with 10GB memory in 1 minute.

Under preemptive scheduling with time slicing, Spin reports that *Dynamic* and *BlockQ* violate safety properties (S0) and (S1) respectively. In error traces reported by Spin, we find that a task may not execute even though it is scheduled by the FreeRTOS scheduler. To see how it happens, consider the SysTick interrupt is triggered while the PendSV interrupt handler is running. Since both interrupts have the same priority, the SysTick interrupt is pending until the PendSV interrupt handler finishes. Recall that the PendSV interrupt handler calls the scheduler to elect a task for execution. Let us call the elected task as the *victim*. The victim task is scheduled to execute after the exception returns.

However, the SysTick interrupt is still pending. Due to tail chaining, the SysTick interrupt handler will execute before the victim task. In the time slicing policy, the SysTick interrupt handler will trigger another PendSV interrupt to schedule a task. The scheduler incorrectly believes the victim task has used up its time slice and chooses another task for execution. In error traces, the victim task repeatedly misses its time slice and hence cannot prepare the thread-safe queue shared with another blocked task. When the blocked task expires its duration, the shared thread-safe queue is still not ready. *Dynamic* and *BlockQ* hence violate safety properties (S0) and (S1) respectively.

Although assertion errors are found in our formal analysis, they are not necessarily failures in reality. It is important to recall that our application models are not FreeRTOS example applications. During model construction, abstraction and simplification are indispensable for effective formal analysis. For instance, the SysTick interrupt is not triggered periodically in our timeless formal models. It is therefore important to reproduce assertion failures in real hardware. To this end, we install the FreeRTOS V10.5.1 on the STM32F429I-DISC1 board with an ARM Cortex-M4 processor and modify FreeRTOS example applications to reproduce Spin error traces on the board. An on-board LED will flash with high frequency if an assertion failure does occur.

The failed safety properties in *Dynamic* and *BlockQ* under preemptive scheduling with time slicing are successfully reproduced on STM32F429I-DISC1. For the safety property (S1) in *BlockQ*, we add a spoil task to the example application. When the spoil task is scheduled for execution, it runs for the time slightly shorter than the SysTick period and then yields. After the spoil task yields, the FreeRTOS scheduler will choose a victim task such as a producer task in *BlockQ*. However, the SysTick interrupt is triggered but remain pending due to our spoil task. The victim task will be preempted before it executes. An assertion failure in the victim producer task is observed.

In reality, the SysTick interrupt may not be triggered shortly after the spoil task yields. The spoil task simply repeats itself and yields the processor shortly before the next time tick. The assertion failure will be observed eventually. The failed safety property (S0) in *Dynamic* is reproduced similarly. Two assertion failures found by our formal analysis are reproduced successfully.

After the reproduction, we find that a similar pattern had been independently exploited in 2007. Tsafrir et al. [19] made non-privileged applications arbitrarily monopolize processors by controlling processor cycles between two clock ticks. They concluded that any periodically ticking system at that time is vulnerable to their exploit. Their exploit and our reproduction are similar in controlling processor cycles between ticks, but different in the cause of the problem.

6.2 Analysis of Liveness Property

Table 2 also reports verification results for the liveness property in all example application models under different scheduling policies. Spin uses up to 20GB of memory within 10 minutes for each verification run. Many example application models do not satisfy the liveness property.

Liveness under cooperative scheduling Only one application model violates the liveness property in Table 2. The error trace reported by SPIN shows that two of the task models in *Semtest* never yield. Since preemption is disabled, other task models cannot be scheduled for execution. No progress can be made. The liveness property fails.

Reproducing the error on the STM32F429I-DISC1 board is easy. We configure FreeRTOS to use the cooperative scheduling policy. The on-board LED indicates an assertion failure without modifying the *Semtest* application.

Liveness under preemption without time slicing The application models *Semtest*, *Countsem*, *GenQTest*, and *Dynamic* violate the liveness property under the preemptive scheduling without time slicing (Table 2). After examining their error traces, we find a task never yields and other tasks are not delayed in each model. Since time slicing is not enabled, the SysTick interrupt handler does not trigger the PendSV interrupt. No task will be scheduled for execution. When thread-safe structures become not ready, never-yielding tasks will be moved to waiting task queues. No progress can be made afterwards. The liveness property subsequently fails.

It is easy to reproduce assertion failures in *Semtest*, *Countsem*, and *GenQTest*. After configuring FreeRTOS with the scheduling policy, assertion failures in check tasks are observed without any modification.

Most interestingly, *Dynamic* requires some efforts to reproduce assertion failures under preemptive scheduling without time slicing. Recall that a monitor task is used to check progress in these example applications (Section 5.2). This monitor task has the highest priority with non-zero delays. The never-yielding task in *Dynamic* is preempted by its monitor task periodically; other tasks will then be scheduled for execution. Progress can still be made due to the monitor task in *Dynamic*. To reproduce the assertion failure in *Dynamic*, we changes the execution order of consumer and producer tasks in the example application. After this simple modification, an assertion failure in the monitor task is observed in *Dynamic*.

Liveness under preemption with time slicing Surprisingly, the liveness property fails in almost all application models under preemptive scheduling with time slicing. After examining error traces, the problem in Section 6.1 is observed again. When the SysTick interrupt is triggered while the PendSV interrupt handler model is running, recall that a victim task will miss its chance of execution. In the extreme scenario, a task can be the victim whenever it is scheduled. The victim task will never execute and starve. The liveness property hence fails.

It is tricky to reproduce the starvation on real hardware. As a proof of concept, we choose the example application *Countsem* with two never-yielding tasks to reproduce the failure. The idle task is configured to yield in the application. Similar to Section 6.1, we add a spoil task to *Countsem*. The spoil task occupies the ARM Cortex-M4 processor for a fixed time. It ensures the SysTick interrupt is triggered shortly after the idle task yields. When the idle task yields, a task

is elected and becomes the victim. The second task will be elected. After the second task finishes its execution, the spoil task repeats and forces the first task to be the victim again.

Incidentally, *PollQ* satisfies the liveness property. We observe two factors that make *PollQ* immune from this problem. First, other tasks have priorities higher than the idle task. This prevents the idle task from preempting the descendant task when the idle task should yield the processor. Second, tasks in *PollQ* delay themselves after synchronization. Recall that a running task may unblock others through the thread-safe structure. If the unblocked tasks have a priority over the running task, the running task is preempted. When the preempted task continues its execution, it then delays itself. The delay prevents the running task from repeatedly preempting the descendant task.

6.3 Discussion

In our formal analysis, we find three types of assertion failures in FreeRTOS example applications. Section 6.1 reports assertion failures where a task may be preempted before its scheduled execution. Under preemptive scheduling with time slicing, the FreeRTOS scheduler can be invoked twice by consecutive executions of the PendSV and SysTick interrupt handlers. The task elected by the first invocation is preempted by the second invocation before its scheduled execution. If a task is continuously preempted by the synchronous PendSV and SysTick interrupts, it will continuously miss its scheduled execution. To observe such failures, the PendSV and SysTick interrupts need be synchronized. We are not aware of any report about such assertion failures.

In the reproduction of these failures, we let a victim task miss its scheduled execution by controlling the predecessor of the victim task. After that, we report our discovery on the FreeRTOS forum. [1] The FreeRTOS community states that the actual behavior of FreeRTOS time slicing depends on how tasks are programmed (as CPU or I/O bound tasks). Since we intensely control one task in the reproduction, the community thinks our reproduction is unnatural. We agree programmers would not intensely produce failures in their application. However, our reproduction shows these failures might happen by accident if the slicing algorithm is not changed.

Assertion failures of the second type are reported in Section 6.2. Under preemptive scheduling without time slicing, never-yielding tasks can lead to starvation when they use thread-safe structures. In this case, other tasks cannot be scheduled because never-yielding tasks are running. When thread-safe structures become not ready, never-yielding tasks are moved to waiting task queues and applications cannot progress. The second type of assertion failures can be elusive. Since FreeRTOS example applications add monitor tasks to check progress. Because these monitor tasks change FreeRTOS scheduling, starvation may not happen. Even though monitor tasks do not contribute to computation actively,

[1] https://forums.freertos.org/t/consecutive-executions-of-the-scheduler/14891

applications must be shipped with monitor tasks to prevent starvation. This is perhaps the most interesting lesson learned from our formal analysis.

Section 6.2 reports assertion failures of the third type. These failures are closely related to the first type. A victim task is preempted before its scheduled execution. Different from the first type that violating safety properties, these failures violate the liveness property. These failures show a victim task eventually stops progressing while the others keep progressing. It is almost impossible for testing to find such assertion failures. Yet we have successfully produced one failure in a FreeRTOS example application with the help of our formal analysis.

7 Related Work

The FreeRTOS official has applied formal methods on the kernel. Chong et al. [7] formally verify the FreeRTOS queue implementation is memory safe and synchronization safe. In comparison with our work, the authors do not verify application code. Besides real-time kernel, FreeRTOS network interface is checked against memory safety with bounded model checking [6].

Chandrasekaran et al. [4] model a custom implementation of multi-core FreeRTOS in Promela. They use Spin to verify their model against data-race and deadlock. Different approaches are used to check the FreeRTOS kernel. In [5, 10, 12, 18], theorem provers are used to prove FreeRTOS functional correctness. In comparison with our work, our models can check liveness properties. In [10, 18], theorem provers are used to prove abstract models were indeed refined by the FreeRTOS source code. In comparison with our work, we choose to validate our abstract model by reproducing error traces on a real hardware and consult the FreeRTOS community about our findings. Asadollah et al. [2] use runtime verification on FreeRTOS. They parse FreeRTOS' runtime events to discover concurrent bugs such as deadlock and starvation. None of the above works considers architecture effects such as tail chaining. Architecture effects are highly relied on interrupt handling that often changes the processor context at runtime.

Task scheduling on a uni-processor requires complex interaction between tasks and interrupts. In [11, 20], separation logic is used to formalize such systems. The authors of [20] further verify functional correctness on their model. Our work has a similar goal, but further involves specific architecture effects.

Other real-time kernels are also analyzed formally. de Oliveira et al. [9, 17] develop a Linux kernel module to find unexpected events of the Linux PREEMPT_RT kernel at runtime. In comparison with our work, we analyze the error traces generated from our abstract models instead of logged events. Andronick et al. [1] use a theorem prover to prove the eChronos real-time operating system against its functional correctness.

Timed properties of real-time tasks are formally analyzed in other works. Hladik et al. [15] propose a tool to generate an executable code and a verifiable model from specifications. The code is guaranteed to satisfy time constraints when it is executed on a specific execution engine. Guo et al. [13] formally prove that real-time tasks on the real-time CertiKOS kernel satisfy their specified

deadlines and scheduling policies. Guo et al. [14] verify communications among network nodes against timed properties. In comparison with our work, we choose to build a timeless model because FreeRTOS tasks are not explicitly constrained by hard deadlines and any definition of time in an abstract model is not real.

8 Conclusion

We have presented a formal model for the FreeRTOS scheduler and a number of standard FreeRTOS applications on ARM Cortex-M4 cores. The application models contains assertions to specify expected behaviors. Through model checking, we find several assertion errors under certain scheduling policies. Those assertion errors are analyzed and reproduced on a physical development board with the ARM Cortex-M4 core.

In addition to the ARM Cortex-M4 architecture, we also model RISC-V RV32 interrupt mechanism and FreeRTOS RV32 port. Thanks to the portability of the FreeRTOS kernel, our scheduler and applications models remain unchanged. It is worth noting that the models of ARM Cortex-M4 interrupt mechanism and the corresponding FreeRTOS port are 406 lines of Promela code while the models of RISC-V RV32 interrupt mechanism and the corresponding port are 345 lines of Promela code. Our full model is approximately 4,400 lines of Promela code. Finally, our model and reproduction are both available online. [2]

References

1. Andronick, J., Lewis, C., Matichuk, D., Morgan, C., Rizkallah, C.: Proof of os scheduling behavior in the presence of interrupt-induced concurrency. In: Blanchette, J.C., Merz, S. (eds.) Interactive Theorem Proving. pp. 52–68. Springer International Publishing, Cham (2016)
2. Asadollah, S.A., Sundmark, D., Eldh, S., Hansson, H.: A runtime verification tool for detecting concurrency bugs in freertos embedded software. In: 2018 17th International Symposium on Parallel and Distributed Computing (ISPDC). pp. 172–179 (2018). https://doi.org/10.1109/ISPDC2018.2018.00032
3. AspenCore: 2019 embedded markets study. Tech. rep., EE Times and Embedded (March 2019), https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf
4. Chandrasekaran, P., Shibu Kumar, K.B., Minz, R.L., D'Souza, D., Meshram, L.: A multi-core version of freertos verified for datarace and deadlock freedom. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE). pp. 62–71 (2014). https://doi.org/10.1109/MEMCOD.2014.6961844
5. Cheng, S., Woodcock, J., D'souza, D.: Using formal reasoning on a model of tasks for freertos. Form. Asp. Comput. **27**(1), 167–192 (Jan 2015). https://doi.org/10.1007/s00165-014-0308-9
6. Chong, N.: Ensuring the memory safety of freertos part 1. https://www.freertos.org/2020/02/ensuring-the-memory-safety-of-freertos-part-1.html (Feburary 2020)

[2] https://doi.org/10.5281/zenodo.12209492

7. Chong, N., Jacobs, B.: Formally verifying freertos' interprocess communication mechanism. In: Proceedings of the Embedded World Conference. Nürnberg, Germany (03 2021), https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (2000)
9. de Oliveira, D.B., de Oliveira, R.S., Cucinotta, T.: Untangling the intricacies of thread synchronization in the preempt_rt linux kernel. In: 2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC). pp. 1–9. IEEE, Valencia, Spain (2019)
10. Divakaran, S., D'Souza, D., Kushwah, A., Sampath, P., Sridhar, N., Woodcock, J.: Refinement-based verification of the freertos scheduler in vcc. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering. pp. 170–186. Springer International Publishing, Cham (2015)
11. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 170–182. PLDI '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375603
12. Ferreira, J.F., He, G., Qin, S.: Automated verification of the freertos scheduler in hip/sleek. In: 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering. pp. 51–58 (2012). https://doi.org/10.1109/TASE.2012.45
13. Guo, X., Lesourd, M., Liu, M., Rieg, L., Shao, Z.: Integrating formal schedulability analysis into a verified os kernel. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 496–514. Springer International Publishing, Cham (2019)
14. Guo, X., Lin, H.H., Aoki, T., Chiba, Y.: A reusable framework for modeling and verifying in-vehicle networking systems in the presence of can and flexray. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). pp. 140–149 (2017). https://doi.org/10.1109/APSEC.2017.20
15. Hladik, P.E., Ingrand, F., Dal Zilio, S., Tekin, R.: Hippo: A Formal-Model Execution Engine to Control and Verify Critical Real-Time Systems. Journal of Systems and Software **181**, 111033 (Nov 2021). https://doi.org/10.1016/j.jss.2021.111033
16. Holzmann, G.J.: The model checker spin. IEEE Trans. Softw. Eng. **23**(5), 279–295 (May 1997). https://doi.org/10.1109/32.588521
17. de Oliveira, D.B., Cucinotta, T., de Oliveira, R.S.: Efficient formal verification for the linux kernel. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 315–332. Springer International Publishing, Cham (2019)
18. Sanan, D., Yang, L., Yongwang, Z., Zhenchang, X., Hinchey, M.: Verifying freertos' cyclic doubly linked list implementation: From abstract specification to machine code. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 120–129. IEEE Computer Society, Los Alamitos, CA, USA (dec 2015). https://doi.org/10.1109/ICECCS.2015.23
19. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Secretly monopolizing the CPU without superuser privileges. In: 16th USENIX Security Symposium (USENIX Security 07). USENIX Association, Boston, MA (Aug 2007)
20. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive os kernels. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 59–79. Springer International Publishing, Cham (2016)