Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation

MENGQI LIU, Yale University, USA
LIONEL RIEG, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France and Yale University, USA
ZHONG SHAO, Yale University, USA
RONGHUI GU, Columbia University, USA
DAVID COSTANZO, Yale University, USA
JUNG-EUN KIM, Yale University, USA
MAN-KI YOON, Yale University, USA

The reliability and security of safety-critical real-time systems are of utmost importance because the failure of these systems could incur severe consequences (e.g., loss of lives or failure of a mission). Such properties require strong isolation between components and they rely on enforcement mechanisms provided by the underlying operating system (OS) kernel. In addition to spatial isolation which is commonly provided by OS kernels to various extents, it also requires temporal isolation, that is, properties on the schedule of one component (e.g., schedulability) are independent of behaviors of other components. The strict isolation between components relies critically on algorithmic properties of the *concrete implementation* of the scheduler, such as timely provision of time slots, obliviousness to preemption, etc. However, existing work either only reasons about an abstract model of the scheduler, or proves properties of the scheduler implementation that are not rich enough to establish the isolation between different components.

In this paper, we present a novel compositional framework for reasoning about algorithmic properties of the concrete implementation of preemptive schedulers. In particular, we use *virtual timeline*, a variant of the supply bound function used in real-time scheduling analysis, to specify and reason about the scheduling of each component in isolation. We show that the properties proved on this abstraction carry down to the generated assembly code of the OS kernel. Using this framework, we successfully verify a real-time OS kernel, which extends mCertiKOS, a single-processor non-preemptive kernel, with user-level preemption, a verified timer interrupt handler, and a verified real-time scheduler. We prove that in the absence of microarchitectural-level timing channels, this new kernel enjoys temporal and spatial isolation on top of the functional correctness guarantee. All the proofs are implemented in the Coq proof assistant.

CCS Concepts: • **Theory of computation → Program verification**; **Abstraction**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: preemptive scheduler, fixed-priority scheduling, partitioned scheduling, temporal isolation, formal verification, mechanized proof

Authors' addresses: Mengqi Liu, Department of Computer Science, Yale University, 51 Prospect Street, New Haven, CT 06520-8285, USA, mengqi.liu@yale.edu; Lionel Rieg, Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, 38000 Grenoble, France, lionel.rieg@univ-grenoble-alpes.fr; Zhong Shao, Yale University, USA, zhong.shao@yale.edu; Ronghui Gu, Columbia University, USA, ronghui.gu@columbia.edu; David Costanzo, Yale University, USA, david.costanzo@gmail.com; Jung-Eun Kim, Yale University, USA, jung-eun.kim@yale.edu; Man-Ki Yoon, Yale University, USA, man-ki.yoon@yale.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/1-ART20
https://doi.org/10.1145/3371088

ACM Reference Format:
Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2020. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proc. ACM Program. Lang.* 4, POPL, Article 20 (January 2020), 31 pages. https://doi.org/10.1145/3371088

1 INTRODUCTION

Real-time systems often carry out safety-critical operations that require strong timing guarantees. For example, the flight controller of a quadcopter needs to send control signals to its actuators in a timely manner, otherwise, the stability of the quadcopter will not be ensured, potentially leading to a physical crash. Furthermore, this problem is exacerbated by the increasing trend of accommodating multiple software components, not necessarily from the same vendor, onto a single platform. On the one hand, the inevitable and intricate interferences between components make it hard to examine whether a safety-critical component always meets its timing requirements. On the other hand, the co-residence of multiple components gives rise to security issues, such as whether malicious components could infer information from other trusted ones that hold secrets. In this sense, strong isolation between components is of utmost importance to the reliability and security of such systems.

1.1 Overview of Existing Work

A real-time system usually relies on an operating system (OS) kernel to manage and schedule all its application-level constituent components, and it is the responsibility of the OS kernel to enforce strong isolation between these components.

Traditionally, OS kernels emphasize mostly *spatial isolation* [Costanzo et al. 2016; Murray et al. 2013; Sigurbjarnarson et al. 2018]. Thanks to hardware mechanisms such as virtual memory, and by carefully managing resources such as memory quotas and process IDs in software, an OS kernel provides to each component an illusion of exclusive access to memory and various kernel objects, free from interferences of other components.

However, isolation in the context of real-time systems also relies heavily on *temporal isolation*, which means that a component's capability to meet its temporal constraints is not influenced by the behavior of others. At minimum, an OS kernel should take advantage of hardware mechanisms such as timer interrupts to preempt a component from monopolizing the CPU time. On top of preemption, ideally, an OS kernel should also enforce temporal isolation at both the microarchitectural level and the algorithmic level.

Temporal isolation at the microarchitectural level. Due to the complexity of modern hardware, components may interfere with each other through shared cache lines, memory banks, etc., also known as microarchitectural level timing channels. For example, when two components share a common cache line, the execution of one of them might affect the execution time of the other one, thus jeopardizing the other's capability of meeting timing constraints. Existing work [Ge et al. 2019; Sha et al. 2016] mitigate this problem by partitioning physical resources properly so as to reduce the non-deterministic temporal behavior due to sharing between components.

Another aspect is the analysis of interrupt latency and context switch overhead. The OS kernel relies on timer interrupts to allocate time resource. Even though interrupts occur periodically, their handling may be delayed if the system is in an uninterruptible state, and the context switch overhead also influences the actual execution time allocated to the scheduled task. Existing work [Blackham et al. 2011; Sewell et al. 2017] conduct worst-case execution time analysis on the OS kernel to compute a bound on both the interrupt latency and kernel overhead, so that users of the system may compensate for this loss of execution time by declaring a larger budget for their tasks.

Temporal isolation at the algorithmic level. Temporal isolation also relies heavily on the scheduler to properly allocate time slots (i.e., the duration between successive interrupts) so that every component meets its temporal constraints. Such reasoning requires examining algorithmic properties of the concrete schedule, which describes the exact time slots a task occupies. For example, to reason about the schedulability of a real-time task, one must check whether this task is scheduled for its budgeted number of time slots during each period (see Sec. 2.2). This problem has not been fully addressed by existing work, and it is the main focus of this work.

1.2 Reasoning about Temporal Isolation at the Algorithmic Level

Temporal isolation at the algorithmic level is essential for establishing isolation between components in an OS kernel. It relies heavily on reasoning about algorithmic properties of the concrete implementation of a preemptive scheduler, which is challenging in the following ways.

Formalizing algorithmic properties of the scheduler implementation remains a challenge. Despite the rich literature in formal verification of real-time OS kernels [Andronick et al. 2016; Xu et al. 2016], most of them stop at the policy level, such as proving that the running task always has the highest priority among ready ones, or proving that there is no priority inversion. These policy-level properties are not strong enough to prove algorithmic properties of the scheduler implementation. For example, the schedulability of a task states that the task never misses its deadline even in the worst-case, which requires more sophisticated algorithmic reasoning than merely showing that the scheduled task indeed has the highest priority.

Algorithmic reasoning requires decomposition despite interferences between components. One important aspect of temporal isolation is schedulability, whose verification requires proving that tasks do not prevent each other from receiving sufficient execution time as long as they have passed a certain schedulability test. However, in most cases, the temporal behavior of a task inevitably depends on the behavior of others. For example, in fixed-priority scheduling (see Sec. 2.2), the schedule of a task is influenced by when and how long higher priority tasks execute. As a result, the OS kernel needs to enforce budget constraints on each task so that no task can monopolize the CPU time. On the other hand, a schedulability proof is needed to justify the sufficiency of those budget constraints. This intertwining makes it difficult to scale the formal reasoning without a proper abstraction of the interferences.

Partitioned scheduling brings more complexity. On top of the above challenges for flattened (i.e., task-level) scheduling, reasoning about more sophisticated policies such as partitioned scheduling [Davis and Burns 2005] brings more complexity. For example, in the QNX real-time OS [QNX 2019] and the framework presented in Kim et al. [2015], tasks are scheduled following a global priority, yet they are also subject to partition budgets. In this way, the schedule of a task is influenced by tasks both within the same partition and from other partitions, making the reasoning hard to scale. Furthermore, the isolation between components also requires that the local schedule of tasks in a partition be independent of other partitions, which adds more proof obligations to the algorithmic reasoning about the scheduler.

1.3 Contributions and Scope of This Paper

This paper studies isolation properties of real-time OS kernels. In particular, we focus on formal reasoning about algorithmic properties of the concrete implementation of a preemptive scheduler, and make the following contributions to address the above challenges. Our work is formalized in Coq and built on top of CertiKOS [Costanzo et al. 2016; Gu et al. 2015].

- A novel application of supply and demand functions [Liu 2000], called *virtual timelines*, to describe temporal properties of real-time tasks and connect them with the actual code. In this way, all high-level properties proved on this abstraction carry down to the generated assembly code of the system.
- A novel approach that uses virtual timelines to address the isolation property of a component's schedule. For flattened fixed-priority scheduling, we prove that a task's schedule is independent of the behavior of lower-priority tasks. For partitioned scheduling, we prove that the local schedule of tasks in a partition is independent of other partitions.
- A compositional framework to reason about algorithmic properties (e.g. schedulability) of each component's schedule in isolation. In particular, we propose a way to statically encapsulate interferences from other components in the system, and we show how to reason about a component as if it were running on its own virtual timeline.
- Combining all of these together, we present a fully verified OS kernel with both temporal and spatial isolation using our compositional framework.

How does this work relate to microarchitectural level details? This work focuses on algorithmic properties of the scheduler implementation, which is orthogonal to the microarchitectural level details. For instance, we rely on the user to declare a suitable budget for each task, without worrying about whether the given budget is sufficient or not. However, we believe our work integrates easily with verification on the microarchitectural level issues, such that if both efforts are successful, the user is guaranteed the exact number of time slots per period, and also guaranteed that her task indeed finishes within these time slots.

Are properties proved in this work specified in clock time? Properties proved in this work, such as schedulability of a task, are specified in discrete logical time, i.e. in units of time slots (or equivalently, number of timer interrupts). This makes sense under the assumption that timer interrupts are strictly periodic, and that the interrupt latency and context switch overhead are negligible so that a task is guaranteed the full amount of time for its execution between successive interrupts. We believe our work is flexible enough to integrate with trustworthy worst-case execution time analysis of kernel services so that these assumptions could be relaxed and these overheads could be compensated properly when a user declares the budget of a task.

Reusability of the virtual timeline abstraction. This paper uses CertiKOS as an example to demonstrate the power of virtual timelines in a concrete OS kernel. However, our technique is reusable in other systems. In fact, our approach gives a novel mechanized formal semantics for algorithmic behaviors of preemptive scheduling, which allows us to reason about various isolation properties, such as schedulability, obliviousness to other components, etc. More importantly, we show how to instantiate this semantics with a concrete runtime system, which in this case means the concrete implementation of a real-time scheduler, in a structured way (detailed in Sec. 2.1). Once it is proved that the system indeed implements this scheduling semantics, all high-level properties proved on the virtual timeline abstraction carry down to the generated assembly code.

Roadmap of the Paper. Sec. 2 gives an overview of our reasoning framework for algorithmic properties and introduces the concept of virtual timelines. Sec. 3 discusses how to reason about algorithmic properties of a component's schedule using the virtual timeline. Sec. 4 describes how we connect virtual timelines with the concrete implementation of a preemptive scheduler in an OS kernel, such that all properties proved on the virtual timeline carry down to the actual code of the system. Section 5 and Section 6 discuss proof efforts and limitations of our work. Sections 7 and 8 discuss the related work and then conclude.

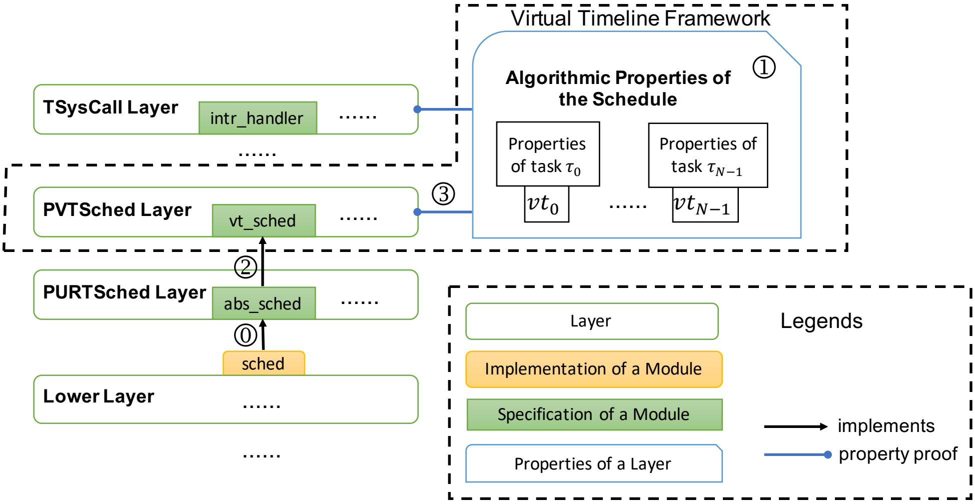


Fig. 1. Fitting the virtual timeline framework into the verification of an OS kernel

2 REASONING FRAMEWORK WITH VIRTUAL TIMELINES

This section gives an overview of our reasoning framework for algorithmic properties of the scheduler in real-time systems. We first explain how this framework fits in the overall verification of an OS kernel. We then describe in more detail the idea of virtual timelines, as well as the computation of virtual time maps.

2.1 Overview

Our work builds on top of the sequential version of CertiKOS, known as mCertiKOS [Costanzo et al. 2016; Gu et al. 2015], a verified single-core OS kernel with a cooperative round-robin scheduler. In the rest of this paper, when we refer to CertiKOS we generally mean this specific version of mCertiKOS. The CertiKOS approach adopts a layered approach for building specifications for the OS kernel, and obtains a formal contextual refinement proof between the high-level specification and low-level implementation of the kernel: a program linked with the kernel implementation and running on top of the x86 assembly semantics exhibits the same behavior as the same program running on top of the assembly semantics plus high-level specifications of system calls.

As shown in Fig. 1, we make the following extensions on top of the original CertiKOS:

(0) We extend the cooperative CertiKOS with user-level preemption and a real-time scheduler, and prove functional correctness of its implementation following a layered approach. We prove that the C implementation of the scheduler, sched (enclosed in an orange rectangle), faithfully implements its Coq specification, abs_sched.

(1) We introduce a virtual timeline for each task (denoted vt_p for task τ_p) to specify and prove algorithmic properties of a task's schedule. See Def. 3.1 and Cor. 3.3 for more details.

(2) We introduce a virtual-time-based scheduler, vt_sched, that operates on each task's virtual timeline to decide its schedule. We then prove that the concrete scheduler in the OS, abs_sched, faithfully implements vt_sched.

(3) We connect a task's virtual timeline with its concrete schedule by proving that vt_sched schedules this task in the exact same way as indicated by its virtual timeline. See Lemma 4.2 as an example.

```
int sched() {
  t++;
  for(int i = 0; i < N; i++){
    if (t % period[i] == 0){
      quanta[i] = budget[i];
    }
  }

  int pid = N;
  for (i = 0; i < N; i++) {
    if (quanta[i] > 0) {
      quanta[i]--;
      pid = i;
      break;
    }
  }
  return pid;
}
```

Fig. 2. C code of the concrete scheduler

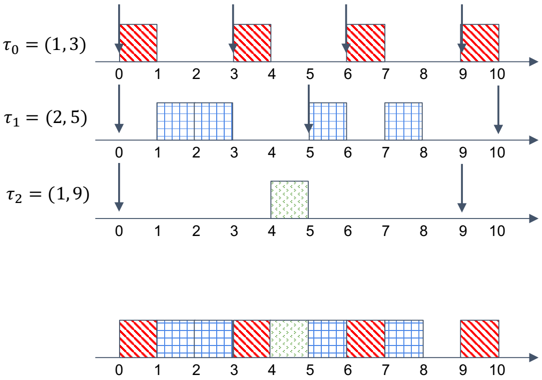


Fig. 3. A concrete schedule

Combining everything together, on top of the functional correctness guarantee of the scheduler, we obtain algorithmic properties for each task's schedule, which are formally connected to the concrete schedule produced by the scheduler in the OS kernel, and which carry down to the generated assembly code of the system.

Also, notice that this framework is not limited to CertiKOS. The abstraction of virtual timelines is generic in specifying and reasoning about fixed-priority scheduling, and it can follow the same approach to connect with other systems employing the same scheduling scheme.

2.2 Real-Time Scheduling and Virtual Timelines

In this section, we explain in more detail intuitions behind virtual timelines, and we use budget-enforcing fixed-priority scheduling as an example. Here, timer interrupts are set up to occur periodically, dividing CPU time into regular time slots. Upon each timer interrupt, its handler invokes the scheduler to decide which task to schedule for the next time slot. We assume that the set of real-time tasks is known ahead of time, denoted as $\{\tau_0, \tau_1, ..., \tau_{N-1}\}$. Each task τ_p occupies a unique priority level p, with 0 being the highest priority level. When several tasks have the same priority, a simple tie-breaker (e.g. task id) could be adopted without affecting the schedulability of the whole system. A task τ_p is specified with (C_p, T_p), representing this task's budget and period, respectively. The budget and period are integer values, in units of time slots.

Fig. 2 shows the C implementation of a budget-enforcing fixed-priority preemptive scheduler. In particular, it uses an integer array, `quanta`, to keep track of the execution time of each task, and makes sure that a task cannot be scheduled for more than its budget in any period. Fig. 3 shows a concrete schedule produced by such a scheduler, from time 0 to 10. On the bottom, we show how the schedules for different tasks are interleaved with each other. And from the top, we show the exact schedule for each task in decreasing order of priority. Here, a down arrow represents the start of a new period. We observe that the number of time slots consumed by a task within each period does not exceed its budget, thanks to the budget-enforcement mechanism. We also observe that the time consumption exactly equals its budget, which is the schedulability of a task: it is guaranteed in each period the full number of execution time slots specified by its budget.

Then, the question is how to formalize algorithmic properties of a task's schedule with isolation in mind. In particular, even though the schedule of tasks interleave with each other, as suggested by both the scheduler implementation and the concrete example, they also follow a strict conceptual

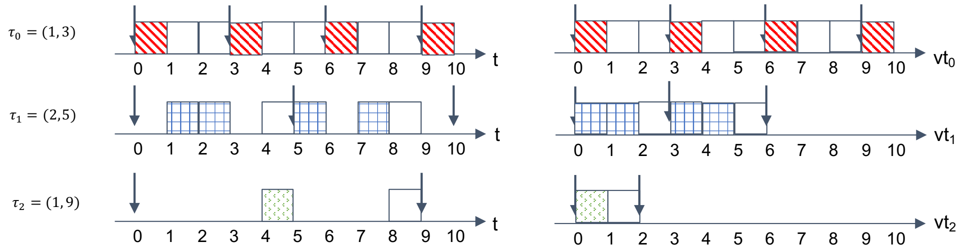


Fig. 4. Comparing schedule on the global timeline (left) with the schedule on the virtual timeline (right)

hierarchy. For example, in Fig. 3, the schedule of task τ_0 is regular because it has the highest priority. Then task τ_1 is scheduled when τ_0 is inactive. Lastly, task τ_2 is scheduled when both τ_0 and τ_1 are inactive.

This inspires the concept of virtual timelines. A *virtual timeline* for a task is intuitively the time "available" to the task. A task's virtual time includes all time that is not occupied by higher priority tasks since a task cannot preempt higher priority ones. Notice that the available time to a task may be allocated to this task, or to lower priority ones when this task has finished, or kept idle when there is no task to execute.

This idea stems from the time supply and time demand functions, from the original time-demand analysis [Lehoczky et al. 1989; Liu 2000]. The original design sums up the total time demand (budget multiplying the number of periods) across all priorities (required execution time), and compares it with the time supply (global time) to determine the schedulability of a system.

We embrace the idea of comparing supply with demand but change their meaning and the way they are computed. Instead of accumulating demand across priorities, we remove the time used by higher priority tasks by creating holes in the virtual timeline with respect to global time. In Fig. 4, we show the schedule of tasks both on the global timeline (t) and their corresponding virtual timelines (vt_p for task τ_p). Notice that the virtual timeline for τ_0 is the same as the global timeline (no holes) as it is the highest priority task. And the virtual timeline for τ_1 is the result of removing all slots occupied by τ_0 from vt_0. Lastly, vt_2 is the result of removing all schedules of τ_1 from vt_1.

The benefit of using virtual time is that once we abstract away the time taken by higher priority tasks, the current task becomes the highest priority one in the system, and hence cannot be interrupted. This means we can ignore all tasks with higher priorities and focus only on the current task, effectively creating an illusion of isolation. For example, in Fig. 4, task τ_1 is scheduled immediately after its arrival on the virtual timeline of vt_1, and finishes when its budget is exhausted. The downside is that the real-time assumptions and requirements (deadlines, periods, *etc.*) are given in the global timeline and not in the virtual one, and we need to be able to convert global time into virtual time.

2.3 Formalizing the Virtual Timeline

As demonstrated in Fig. 4, to ease the reasoning about a task's schedule, we encapsulate interferences from higher-priority tasks in its virtual timeline. This requires a suitable formalization of the virtual timeline that makes it easy to connect a task's schedule in global time to its counterpart in virtual time. We use PT and VT to denote global time and virtual time, respectively. Notice that they are integer values in units of time slots. We use σ_p to denote the connection between global and virtual time. There are various alternative ways of defining σ_p.

```
int instrumented_sched(){
  ......
    // computing pid
    // The same as sched()
  ......

  for(int i = 0; i < N; i++){
    if (i <= pid){
      vt[i][t+1] = vt[i][t] + 1;
    }else{
      vt[i][t+1] = vt[i][t];
    }
  }
  return pid;
}
```

Fig. 5. The vertical computation of time maps

▨ $\tau_0 = (1, 3)$
▦ $\tau_1 = (2, 5)$
░ $\tau_2 = (1, 9)$

t	0	1	2	3	4	5	6	7	8	9	10
$\sigma_0(t)$	0	1	2	3	4	5	6	7	8	9	10
$\sigma_1(t)$	0	0	1	2	2	3	4	4	5	6	6
$\sigma_2(t)$	0	0	0	0	0	1	1	1	1	2	2

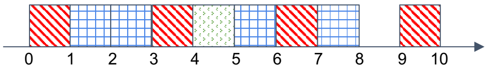


Fig. 6. A concrete time map

(1) σ_p: VT → PT. This is a close analogy to the virtual memory mechanism, where the behavior of a process is determined by its virtual memory space, and the OS kernel maintains its page table to make sure that the corresponding physical memory space does not overlap with others. However, such a formalization complicates the reasoning about temporal properties, which are usually specified in the global time. For example, schedulability requires that a task be scheduled for its full budget within each period, yet it is non-trivial to compute which virtual time duration corresponds to a global period using this formalization of virtual timelines.

(2) σ_p: VT → VT. This approach reflects our observation in Fig. 4: a task's virtual timeline is the result of taking the virtual timeline of the adjacent higher priority and removing slots already consumed by this high priority task. However, this makes the reasoning tedious because we have to compose multiple time maps to describe one virtual timeline. For example, the virtual timeline of τ_2 is described as $\sigma_2 \circ \sigma_1 \circ \sigma_0$.

(3) σ_p: PT → VT. This is the approach we finally settle on. It takes advantage of the recursive computation as in the second approach, while avoiding the tediousness of composing multiple time maps to describe one virtual timeline. We show the computation of such a time map in the following.

As explained above, we define the *time map* σ_p, a function from global time to virtual time, as a formalization of the virtual timeline for τ_p. Fig. 5 demonstrates the general principle of computing time maps, in the form of additional code instruments on top of the `sched()` function. In particular, `vt` is a two-dimensional array of integers, where `vt[i][t]` represents $\sigma_i(t)$. Whenever a time slot [t, t+1) is allocated to a task p, it is available in the virtual timeline of both this task and all higher priority ones. Fig. 6 shows an example of such vertical computation of time maps.

Although intuitive, this way of vertical computation is inconvenient because the dynamic construction of time maps complicates the reasoning about their properties. Instead, we adopt an equivalent approach of horizontal computation for time maps. In particular, if we know beforehand when each task arrives and what its execution time is, the time map can be pre-computed in the decreasing priority order (we discuss the relaxation of these constraints in Sec. 4.4). This is indeed the case when all tasks are periodic and always use up their budgets. Also, notice that there is only one task per priority level.

In this setting, we know exactly when the highest priority task will be executed (at the start of each of its periods) and thus build its timeline. For example, the time map for τ_{p+1} can be incrementally constructed from the one for task τ_p as follows. Below, $\sigma_p(t) - \sigma_p(\left\lfloor \frac{t}{T_p} \right\rfloor T_p)$ denotes the available virtual time for τ_p since the start of the current period. Notice that the execution time of τ_p is also bounded by its budget, C_p, and thus its actual time consumption is the minimum of the two. That means, if C_p is smaller than $\sigma_p(t) - \sigma_p(\left\lfloor \frac{t}{T_p} \right\rfloor T_p)$, task τ_p must have finished its execution at time t.

Definition 2.1 (Time map for periodic tasks in fixed-priority scheduling).

$$\begin{aligned}\sigma_0(t) \quad &:= t \\ \sigma_{p+1}(t) &:= \sigma_p(t) - \underbrace{\left\lfloor \frac{t}{T_p} \right\rfloor C_p}_{\text{time spent in the past full periods}} - \underbrace{\min\left(C_p, \sigma_p(t) - \sigma_p(\left\lfloor \frac{t}{T_p} \right\rfloor T_p)\right)}_{\text{time spent in the current period}}\end{aligned}$$

3 REASONING ABOUT ALGORITHMIC PROPERTIES USING VIRTUAL TIMELINES

In this section, we discuss how the abstraction of virtual timelines facilitates the reasoning about algorithmic properties of a component's schedule, with both flattened and partitioned scheduling. For flattened scheduling, we use fixed-priority scheduling as an example and demonstrate the proof on schedulability and obliviousness to lower priority tasks. For partitioned scheduling, we show how virtual timelines can be used in proving obliviousness to other partitions, and further in proving schedulability of tasks.

3.1 Reasoning about Fixed-Priority Scheduling

In this section, we use fixed-priority scheduling as an example to demonstrate how the virtual timelines could facilitate the reasoning about a task's schedule.

As shown in Fig. 4, the virtual timeline of a task encapsulates interferences from higher priority tasks and allows us to reason about the schedule of this task solely based on its own time map. In particular, on its virtual timeline, a task is scheduled as soon as it arrives, and keeps running until its budget is exhausted.

We use schedulability, which is essential for temporal isolation, as an example to demonstrate how to reason about a task's schedule. Schedulability requires that a task be scheduled for its full amount of budget within each period, despite the worst-case interferences from others. We formalize this property in terms of a task's virtual time.

Definition 3.1. Schedulability of task τ_p

$$\forall i \geq 0, C_p \leq \sigma_p((i+1) * T_p) - \sigma_p(i * T_p)$$

This is a sufficient and necessary condition for schedulability, which states that a task will receive enough virtual time in each of its physical periods. For example, in Fig. 4, the virtual time available to task τ_1 in its first and second period is 3, which is greater than its budget of 2, so this task is schedulable in the first two periods. Notice that the exact amount of available virtual time varies in different periods, depending on the actual interferences from higher priority tasks. However, thanks to the budget-enforcing mechanism of the scheduler, there is an upper bound on the interference that can be decided statically.

For strictly periodic tasks with no dynamic creation, Liu and Layland [1973]'s seminal work solves the scheduling problem by giving an optimal schedule. The essential observation of their analysis is the *critical instant theorem*, stated as follows using our virtual time concepts:

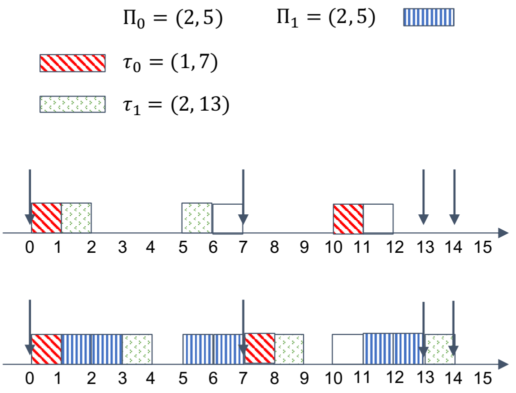


Fig. 7. Partitioned scheduling

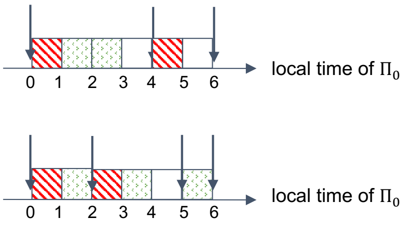


Fig. 8. Local view of the partition schedule

Theorem 3.2 (Virtual Critical Instant Theorem).

$$\textit{For all}\ p, t_0, t,\ \ \left(\forall 0 \le q < p, C_q \le \sigma_q(T_q)\right) \implies \sigma_p(t_0) \le \sigma_p(t_0 + t) - \sigma_p(t)\ .$$

This means that a task suffers the most interference (i.e., has the least available time) during its first period, since all higher priority tasks also arrive at time 0. Theorem 3.2 is formalized and proved in Coq, based on the time map computation given by Def. 2.1. An immediate corollary is the schedulability of the full system:

Corollary 3.3. *The system is schedulable at all times if each task is schedulable for its first period. Said otherwise,*

$$\left(\forall p, C_p \le \sigma_p(T_p)\right) \implies \forall p, i \ge 0, C_p \le \sigma_p((i+1) * T_p) - \sigma_p(i * T_p)$$

3.2 Reasoning about Partitioned Scheduling

Partitioned scheduling is another common scheduling paradigm, where the CPU resource is divided among multiple partitions, with each partition hosting its own set of tasks. Partitions are useful for the administration of a system, e.g. applications independently developed by different vendors could be kept in different partitions, so that tasks within a partition are allowed to communicate and cooperate, while they are also free from interference by other partitions.

Each partition Π_i is reserved a certain proportion of the CPU resource, specified with a partition budget B_i and period P_i, *i.e.*, $\Pi_i = (B_i, P_i)$. The OS kernel guarantees to schedule this partition for B_i time slots in every period, though the exact allocation of these slots may vary depending on the behavior of other partitions. Fig. 7 shows an example of partitioned scheduling. Here, we focus on partition Π_0, which consists of two tasks, where τ_0 has higher priority over τ_1. The upper side axis shows the schedule when Π_0 is the only partition in the system. This partition is scheduled periodically, and its tasks are only scheduled when the partition is active. As a comparison, the lower side shows the case when Π_0 is interleaved with Π_1. Though the partition's exact schedule varies, it is still guaranteed 2 time slots in every period.

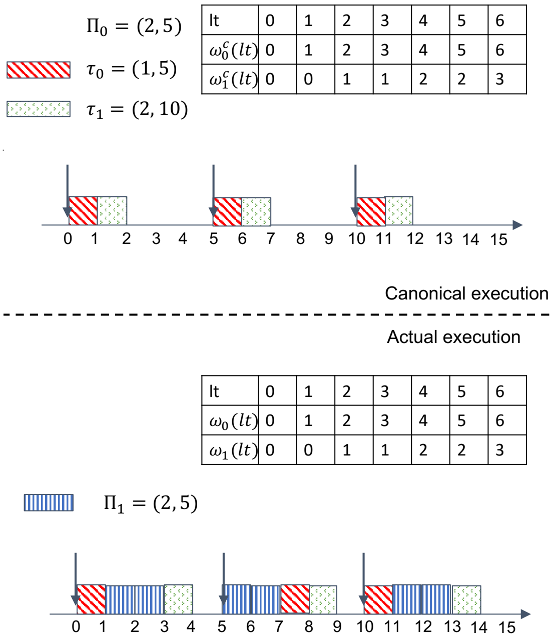
Notice that Fig. 8 demonstrates an interesting phenomenon: from the partition's local view, its tasks' schedule varies depending on the exact schedule of this partition. This is caused by the varying task arrival time in this partition's local view, which clearly leads to interferences between partitions. On the one hand, if tasks cooperate with each other in the same partition, the reordering

```
int instrumented_parsched(){
  ......
  // task τ_j ∈ Π_i is choosen

  int lt = π[i][t];
  for(int p = 0; p < numTasks; p++)
    if (τ_p ∉ Π_i){
      σ[p][t+1] = σ[p][t];
    }else if (p <= j){
      σ[p][t+1] = σ[p][t] + 1;
      ω[p][lt+1] = ω[p][lt] + 1;
    }else{
      σ[p][t+1] = σ[p][t];
      ω[p][lt+1] = ω[p][lt];
    }
  }

  for(int q = 0; q < numPars; q++){
    if (q != i){
      π[q][t+1] = π[q][t];
    }else{
      π[q][t+1] = π[q][t] + 1;
    }
  }
  ......
}
```

Fig. 9. The vertical computation of time maps with partitioned scheduling



lt	0	1	2	3	4	5	6
$\omega_0^c(lt)$	0	1	2	3	4	5	6
$\omega_1^c(lt)$	0	0	1	1	2	2	3

lt	0	1	2	3	4	5	6
$\omega_0(lt)$	0	1	2	3	4	5	6
$\omega_1(lt)$	0	0	1	1	2	2	3

Fig. 10. Another example of local time map for a task

of their schedule might jeopardize its correct operation. On the other hand, the influence from other partitions complicates the temporal property reasoning about the current partition.

Existing work on isolation properties in partitioned scheduling [Murray et al. 2013] assume a fixed schedule for partitions, and their verification framework does not cover the issue presented in Fig. 8. In this work, we address this issue using the notion of virtual timelines.

In particular, we differentiate different types of virtual timelines as follows. We use PT, VT, and ST to represent the timeline for the global time, a task, and a partition, respectively. All of them are of integer type, in units of time slots. Then we use π, ω and σ for time maps of type parmap, localmap and timemap respectively.

```
Definition parmap: PT → ST.     (*time map for a partition*)
Definition localmap: ST → VT.   (*local time map for a task*)
Definition timemap: PT → VT.    (*time map for a task*)
```

Under this notation, π_i describes the schedule of partition Π_i, σ_j describes the schedule of task τ_j, and ω_j describes the schedule of task τ_j in its partition's local view. Fig. 9 demonstrates the general principle of computing time maps with partitioned scheduling, in the form of additional code instruments. Here, the time map and local time map for tasks in Π_i are computed in the same way as Fig. 5, with the only exception that the local time map is indexed by the local time of Π_i. Virtual time for tasks in other partitions does not increment because the next time slot is not available to them. In the end, we increment the local time of Π_i only.

The isolation property requires that the schedule of tasks within Π_i does not change no matter how π_i is defined, as long as the partition schedule is consistent with its period and budget. We use the vertical computation in Fig. 9 to define ω_j. In addition, since this isolation property involves comparing two different executions, we define a *canonical* local time map for the same task, rewritten ω_j^c.

Definition 3.4 (The canonical local time map for task $\tau_j \in \Pi_i$). We define ω_j^c as the result of the vertical computation of its local time map, under the setting that Π_i is the only partition in the system. In other words, it is computed when the following holds.

$$\forall t, \pi_i(t) = \left\lfloor \frac{t}{P_i} \right\rfloor B_i + min(B_i, t - \left\lfloor \frac{t}{P_i} \right\rfloor P_i)$$

This means Π_i always occupies the first B_i time slots in every period.

The isolation property is defined as below.

Definition 3.5 (The obliviousness of the local schedule of partition Π_i).

$$\forall p, \tau_p \in \Pi_i \implies \omega_p = \omega_p^c$$

This states that the local schedule of tasks within a partition is oblivious to other partitions in the system. Since by definition, $\sigma_p = \omega_p \circ \pi_i$, we obtain an equivalent formalization of this isolation property.

Definition 3.6 (The obliviousness of the local schedule of partition Π_i: alternative formalization).

$$\forall p, \tau_p \in \Pi_i \implies \sigma_p = \omega_p^c \circ \pi_i$$

This states that the local schedule within a partition is independent of others, thus even if the partition schedule varies, it only affects the exact time slots taken by this partition, but does not change the ordering of this partition's local schedule. This is analogous to the virtual memory system: we can view ω_p^c, π_i and σ_p as the virtual memory space, the page table, and the physical memory space, respectively. Here, ω_p^c (analogous to the virtual memory space) is decided solely on its own, and the actual allocation of π_i and σ_p (analogous to the page table and physical memory allocation) are always connected by ω_p^c, no matter how both of them are influenced by other partitions in the system.

We observe that this obliviousness holds if all tasks have a period that is a multiple of their partition's period. The intuition is that, under such a constrained setting, task arrivals have to occur at the boundary of the partition's period (i.e., 'bound' task [Davis and Burns 2005]), thus their arrival time in the partition's local view does not change. Fig. 10 gives an example of this constrained setting. Here, the periods of tasks in Π_0 always synchronize with the period of the partition, so that the local schedule of tasks are not affected by other partitions in the system.

Further, under this constrained setting, there is a straightforward way of computing ω_p^c. Observe in Fig. 10 that tasks arrive at local time 0, 2, 4, ..., etc., as if they are still periodic tasks, only with a smaller period. In this case, we define a shrunk period for each task: $T'_p = \frac{T_p}{P_i} B_i$. Then we assume $\Pi_i = \{\tau_0, \tau_1, ..., \tau_{N-1}\}$, and compute local time map for tasks as follows.

Definition 3.7 (Local time map for periodic tasks with fixed-priority scheduling).

$$\begin{aligned} \omega_0^c(t) \quad &:= t \\ \omega_{p+1}^c(t) &:= \omega_p^c(t) - \left\lfloor \frac{t}{T'_p} \right\rfloor C_p - \min\left(C_p, \omega_p^c(t) - \omega_p^c(\left\lfloor \frac{t}{T'_p} \right\rfloor T'_p)\right) \end{aligned}$$

Such obliviousness to other partitions also facilitates the reasoning about schedulability properties. Recall that one of the main difficulties with schedulability analysis is the interference between different components. However, under such a constrained setting, we can transform schedulability, which is specified on the global timeline, into a property on the partition's local timeline, since a task's behavior is only affected by its own partition.

Lemma 3.8. *The schedulability analysis of task τ_p is transformed into reasoning about its local schedule ω_p. That is, for any p and $k \geq 0$,*

$$\begin{aligned} &C_p \leq \sigma_p((k+1) * T_p) - \sigma_p(k * T_p) \\ \Leftrightarrow\ &C_p \leq \omega_p^c((k+1) * T_p * \frac{B_i}{P_i}) - \omega_p^c(k * T_p * \frac{B_i}{P_i}) \end{aligned}$$

Proof. By Def. 3.6, $\sigma_p = \omega_p^c \circ \pi_i$. Since the OS kernel guarantees the full amount of schedule for the partition, we can transform a period of T_p in the global timeline into a smaller period of $T_p * \frac{B_i}{P_i}$ in the partition's local timeline. What's more, the constraint that T_p must be a multiple of P_i allows us to always match the beginning and the end of the two periods, thus we transform schedulability of a task into the reasoning on its local timeline. □

4 CONNECTING VIRTUAL TIMELINES WITH PREEMPTIVE SCHEDULERS

Section 3 demonstrates the high-level reasoning about a component's schedule on its virtual timeline, which corresponds to step ① in Fig. 1. In this section, we discuss how to connect temporal properties proved on the virtual timelines with the concrete scheduler implementation of an OS kernel.

In particular, the high-level reasoning is based on schedules over individual virtual timelines, while the concrete scheduler (e.g. Fig. 2) exhibits a global view of the whole system and produces a schedule mixing all components together. For any particular task, the relationship between its schedule in the global view and its schedule in the virtual timeline is demonstrated in Fig. 4. It is challenging to prove such a correspondence directly based on the scheduler implementation. Instead, we use a virtual-time-based scheduler as an intermediate step toward this connection. We first prove that the concrete scheduler is equivalent to the virtual-time-based scheduler (step ② in Fig. 1). We then prove that for any task, its schedule on the virtual timeline is equivalent to the schedule produced by the virtual-time-based scheduler (step ③ in Fig. 1). Combining both steps together, we connect individual virtual timelines with the concrete scheduler implementation of an OS kernel.

In this section, we first provide background on CertiKOS (Sec. 4.1), and our real-time extension on top of it (Sec. 4.2). We then discuss in detail how we connect virtual timelines with this real-time version of CertiKOS. In particular, we illustrate such a connection under three scheduling scenarios: fixed-priority scheduling with constant execution time (Sec. 4.3), fixed-priority scheduling with variable execution time (Sec. 4.4), and partitioned scheduling (Sec. 4.5).

4.1 Background: CertiKOS, a Verified OS Kernel

CertiKOS is a verified kernel whose functional correctness has been mechanized in the Coq proof assistant.

Abstraction layers. CertiKOS carries down its functional correctness guarantees to the assembly level. This is achieved by dividing the kernel into many small pieces that are verified separately, called *abstraction layers*. Implementation details of lower layers are encapsulated into abstract states and associated primitives in higher layers, thus allowing to reason directly with the specifications rather than the implementations.

We follow this approach in our work to achieve the same level of compositionality, and more importantly, to carry end-to-end guarantees from specifications all the way down to the assembly code.

Notice that temporal properties include safety and liveness. While the former is easily preserved by refinement because high-level invariants also bound the set of possible behaviors of low-level code, the latter may not necessarily be preserved in general. However, CertiKOS employs termination-sensitive and observation-preserving refinement, and every system call is proven terminating. Thus, both safety and liveness properties are preserved by refinement in this work.

The machine model. The CertiKOS kernel uses the CompCert compiler [Blazy et al. 2006; Leroy 2009] to propagate the verification done at the C source code level down to the generated assembly code for several architectures (PowerPC, ARM, x86, and RISC-V). (Actually, it is a slightly modified version of CompCert that handles new instructions and abstraction layers which is still verified.) This compiler enjoys a proof of correctness, mechanized in Coq, which ensures that any behavior of the generated assembly code is also an allowed behavior of the source code. This model does not include all hardware features but only those that the compiler actually uses. States of the system are represented as a register set (rs) and a memory state (m) [Leroy et al. 2014].

The CertiKOS machine model builds on top of CompCert's one by extending the memory model with an abstract state that can contain arbitrary information.

The small step operational semantics remains the same with two additions: new instructions and *primitives*, which are internal functions abstracted as primitive operations, and thus whose execution is seen as a single step instead of as a jump to a function body.

4.2 Real-Time Extension on CertiKOS

Preemption is essential for multiplexing the processor among multiple processes. Particularly, the ability to preempt a running user process and schedule another one immediately enables us to obtain properties that are not achievable in cooperative systems, such as liveness of any user process.

We replace the cooperative scheduler of CertiKOS with a preemptive one (Fig. 2), and prove its functional correctness following the layered approach (step ⓪ in Fig. 1). The timer interrupt is set up to occur periodically, and the scheduler is invoked by the timer interrupt handler.

Whenever a timer interrupt occurs, the interrupt handler acknowledges the receipt of the interrupt, and invokes the scheduler to decide whether to preempt the current task and schedule a new one for the next time quantum. This is similar to FreeRTOS, which also uses a constant-rate (or auto-reload) timer for scheduling. We leave the optimization of using one-shot timer for future work, since it is orthogonal to the formal reasoning in this work.

Verified interrupt handler. In addition to the preemptive scheduler, we also verify the interrupt handler which invokes it. This requires extending the machine model with an interrupt mechanism.

After each instruction, a processor checks the interrupt line to decide whether it is going to execute the next instruction or to jump to an interrupt handler. The original CertiKOS machine model only covers synchronous exceptions, e.g. page faults, but not external interrupts. We model this behavior by a 2-step process at each instruction: first, we detect interrupts with an abstract function `intr_trigger`, then we handle them using an abstract interrupt handler `intr_handler` that dispatches execution to the corresponding interrupt handler if there is a pending interrupt.

We only model the timer interrupt in this way, since polling for other device interrupts is the typical behavior for high-assurance systems [Murray et al. 2013; Sigurbjarnarson et al. 2018]. Nevertheless, our interrupt mechanism is general enough to extend to any kind of device interrupts.

Restriction to the user mode. In order to keep the modifications to the existing verification proofs of CertiKOS to a minimum, we only allow interrupts in user mode. This way, all the proofs about the kernel do not need to be modified since the assembly semantics in kernel mode has not changed.

```
int VTA(int vti[], int t){
  if ((vti[t+1] > vti[t])){
    int start =
      t / period[i] * period[i];
    int consumption =
      vti[t] - vti[start];
    if (consumption < budget[i]){
      return 1;
    }
  }
  return 0;
}

int vt_sched(){
  t++;

  int pid = N;
  for(int i = 0; i < N; i++){
    if (VTA(vt[i], t)){
      pid = i;
      break;
    }
  }

  return pid;
}
```

Fig. 11. Illustration of the virtual-time-based scheduler

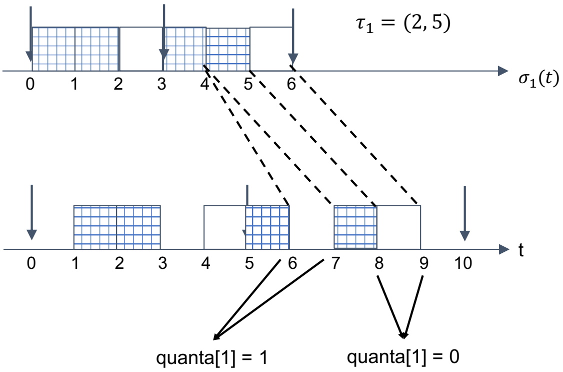


Fig. 12. An example schedule in global and virtual time

The downside of this choice is that disabling interrupts in the kernel may hurt the responsiveness of the system. However, it is possible to adopt the same top-half and bottom-half mechanism of modern operating systems (such as Linux and μC/OS-III), where the interrupt handler itself executes quickly, and the associated time-consuming work is deferred to a regular user/kernel process. In this way, the interrupt disable time could be reduced.

4.3 First Case Study: Constant Execution Time

In this section, we explain how we connect virtual timelines with the concrete scheduler implementation (Fig. 2). As shown in Fig. 1, we already proved that the C implementation of the scheduler is equivalent to its Coq formalization, abs_sched. Then, we introduce the virtual-time-based scheduler vt_sched. Fig. 11 demonstrates the basic idea of vt_sched: it iterates over all tasks until it finds one that is scheduled on the task's virtual timeline. Notice that we use C code for illustrative purposes only. The function vt_sched is actually formalized in Coq.

At the core of vt_sched is a predicate VTA (σ_p, t), which checks whether task τ_p is scheduled within global time duration [t, t+1). Fig. 12 gives a concrete example of a task's schedule in both the global and virtual timeline. In particular, VTA $(\sigma_1, 6) = 0$ (τ_1 is not scheduled at time 6), because the next time slot is not available to τ_1. VTA $(\sigma_1, 7) = 1$, since the higher priority task is finished, and task τ_1 still has remaining budget. VTA $(\sigma_1, 8) = 0$, because its budget is exhausted at this point.

Connecting abs_sched *with* vt_sched. This corresponds to step ② in Fig. 1. The concrete scheduler as shown in Fig. 2 operates on the quanta array to decide the next task to schedule. It is consistent with the virtual-time-based scheduler (as shown in Fig. 11) based on the following intuitions.

Assume abs_sched schedules τ_p at time t. The following must hold, and vice versa:

- All higher priority tasks are finished. That is, $\forall j < p$, quanta $[j] = 0$.

- There is no new arrival of higher priority tasks. That is, $\forall j < p, t \mod T_p \neq 0$. This corresponds to the first loop in Fig. 2. When a new period starts, the quantum value will be reset to this task's budget.
- Task τ_p still has remaining budget, or a new period is arriving. That is, quanta $[p] > 0 \vee t \mod T_p = 0$.

Similarly, if vt_sched schedules τ_p at time t, the following must hold, and vice versa:

- For each higher priority task τ_j, VTA $(\sigma_j$, t) = 0.
- VTA $(\sigma_p$, t) = 1.

This indicates that the connection between abs_sched and vt_sched relies on the relation between a task's virtual time and its current quantum value. In particular, we observe in Fig. 12 that a task's quantum is decremented along with the increment in its virtual time, and is eventually depleted so that this task won't be scheduled until the next period. To put it formally, we prove the lemma below to facilitate the refinement proof between abs_sched and vt_sched.

Definition 4.1. Equivalence between states of the abstract scheduler and the virtual-time-based scheduler

$$\texttt{quanta}[p] = C_p - \min\left(C_p, \sigma_p(t) - \sigma_p\left(\left\lfloor \frac{t}{T_p} \right\rfloor T_p\right)\right)$$

It means that whenever the concrete scheduler finishes scheduling, the remaining quanta is the budget minus the elapsed virtual time between the beginning of this period and the end of the next time slot. We prove in Coq that this invariant is indeed preserved by the scheduler, and we, in turn, prove that abs_sched and vt_sched always produce the same schedule.

The sufficiency of individual virtual timelines. To address one of the main challenges of temporal reasoning, that all tasks are interleaved together in the schedule, we carry out step ③ in Fig. 1 and prove that it is sufficient to examine the virtual timeline of a specific task to reason about its schedule. This is consistent with Fig. 4, where the schedules of different tasks never overlap with each other. In other words, the intuition is that, at any moment, there can be at most one task τ_p that satisfies VTA $(\sigma_p$, t).

We prove in Coq that if VTA $(\sigma_p$, t) = 1 for task τ_p, then VTA $(\sigma_j$, t) = 1 does not hold for any higher priority task τ_j. In particular, for any task τ_j that has a higher priority than τ_p, VTA $(\sigma_j$, t) = 0 must hold. This is true by contradiction. Assume that there is a higher priority task τ_j such that VTA $(\sigma_j$, t) = 1. Since time slot [t, t+1) is consumed, it becomes unavailable to task τ_{j+1} and all subsequent lower priority tasks. Thus $\sigma_p(t+1) = \sigma_p(t)$, which contradicts the assumption that VTA $(\sigma_p$, t) = 1.

In this way, we formally establish the relation between a task's schedule and its virtual timeline.

Lemma 4.2 (The virtual timeline faithfully describes the schedule of a task). *For all p, t,*

$$\texttt{vt\_sched}(t) = p \Leftrightarrow \texttt{VTA}(\sigma_p, t) = 1$$

The main theorem. Combining everything together, on top of the functional correctness of the system, we obtain a schedulability proof for each task, as formalized in Corollary 3.3. And the schedulability property carries down to the generated assembly code of the system.

4.4 Second Case Study: Variable Execution Time

In this section, we discuss how to allow a periodic task to finish early and give up its remaining quanta to others. This model is more realistic and achieves better utilization for the whole system, i.e., more CPU time can be given to batch tasks when periodic ones do not require all their budgets.

```
void sys_finish(){
  quanta[cur_pid] = 0;
}
```

Fig. 13. The sys_finish system call

```
void vt_sys_finish{
  int p = cur_pid;
  int start = t / period[p] * period[p];
  int exec_time = vt[p][t] - vt[p][start];
  O[p][t / period[p]] = exec_time;
}
```

Fig. 14. The virtual-time-based sys_finish

This requires a system call sys_finish, similar to periodic_wait in the ARINC 653 standard [ARINC 2015]. The core of its C implementation is straightforward, as shown in Fig. 13.

Despite the simplicity of its C implementation, sys_finish complicates both the computation of time maps and the connection between virtual timelines with the concrete scheduler.

Computation of the time map. Under this setting, the fixed-priority scheduler is the same as in Sec. 4.3, thus time maps are defined in the same way as demonstrated in Fig. 5, that is, a time slot is available to the scheduled task and all higher priority tasks. However, the computation in Def. 2.1 is no longer valid because it assumes that every task uses up its budget in each period, which overly simplifies the accounting of time slots taken by higher priority tasks in the current situation.

We observe that the only effect of invoking sys_finish is that the actual execution time of a task may become smaller than its budget, which can only be known at runtime, making the static computation of time maps hard. However, it is possible to maintain an auxiliary data structure, O, to represent an oracle for execution times. Specifically, $O_p(k)$ is the execution time at the k-th period $[kT_p, (k+1)T_p)$ of task τ_p. We then parameterize the computation of a time map over an oracle, written as σ_p^O.

Definition 4.3 (Computation of dynamic time map).

$$\begin{aligned}\sigma_0^O(t) &= t\\ \sigma_{p+1}^O(t) &= \sigma_p^O(t) - \sum_{j=0}^{\left\lfloor \frac{t}{T_p}\right\rfloor -1} O_p(j)\\ &\quad - \min\left(O_p\left(\left\lfloor \frac{t}{T_p}\right\rfloor\right), \sigma_p^O(t) - \sigma_p^O\left(\left\lfloor \frac{t}{T_p}\right\rfloor T_p\right)\right)\end{aligned}$$

Compared with Def. 2.1, this parameterized version follows the same idea of recursive computation in decreasing priority order, while allowing variations in the actual execution time. In the following, we show two high-level properties we prove using this dynamic time map.

Obliviousness to lower priority tasks. One of the benefits of fixed-priority scheduling is its robustness, which states that the behavior of lower priority tasks can never affect higher priority ones. In the current setting, this means despite the varying actual execution time of lower priority tasks, the virtual timelines for higher priority ones stay the same.

Lemma 4.4 (Obliviousness to lower priority tasks). *We consider the same task set with two different oracles, O^1 and O^2. The behavior of task τ_p does not change if all higher priority tasks exhibit the same behavior.*

$$\forall O^1, O^2, (\forall k < p, O_k^1 = O_k^2) \implies \sigma_p^{O^1} = \sigma_p^{O^2}$$

This is straightforward by observing in Def. 4.3 that the computation only relies on the execution time of higher priority tasks.

Schedulability of a task. Intuitively, if a task set is schedulable when every task always uses up its budget, then it is still schedulable when some tasks do not always use up their budget, since calling `sys_finish` could only result in more free time.

We define an oracle O^{init} for the case when every task must use up its budget, such that $\forall p, i, O_p^{init}(i) = C_p$. We also prove that the actual execution time is always bounded by the budget, in particular, after being updated by `sys_finish`.

$$\forall p, i, O_p(i) \leq O_p^{init}(i)$$

Using this invariant, we prove that if the time map corresponding to O^{init} is schedulable, all subsequent mappings corresponding to a bounded O are also schedulable. This reduces schedulability analysis into the simple check in Corollary. 3.3.

Connection with the concrete implementation. Lastly, we discuss how to connect this dynamic version of virtual timelines to the system implementation. Notice that the concrete scheduler is the same as in Sec. 4.3, and the virtual-time-based scheduler is also similar, except with a parameterized version of time maps. We update the equivalence of states (Def. 4.1) accordingly.

Definition 4.5 (Equivalence between states of the abstract scheduler and the virtual-time-based scheduler, under the setting of variable execution time).

$$\texttt{quanta}[p] = O_p(\left\lfloor \frac{t}{T_p} \right\rfloor) - \min\left(O_p(\left\lfloor \frac{t}{T_p} \right\rfloor), \sigma_p^O(t) - \sigma_p^O\left(\left\lfloor \frac{t}{T_p} \right\rfloor T_p\right)\right)$$

We prove that this relation is preserved by the scheduler, and further prove the refinement between the concrete scheduler and the virtual-time-based scheduler.

The remaining proof obligation is to connect the virtual-time-based system call, `vt_sys_finish` (illustrated in Fig. 14 using C code), with `sys_finish`. Here, `exec_time` represents the actual execution time of task p. Its computation exploits the fact that all lower priority tasks are not runnable until task p is finished, i.e., the entire virtual time duration is consumed by task p. We prove the equivalence of the two by showing that they also preserve the relation in Def. 4.5.

4.5 Third Case Study: Partitioned Scheduling

In this section, we discuss how the virtual time abstraction can be applied to partitioned scheduling. In particular, we apply virtual timelines in two different settings: TDMA (Time-Division Multiple Access) schedules and dynamic schedules for partitions. Throughout this section, we assume that the period of a task is a multiple of the period of its partition, which is common in choosing parameters for partitions. We adopt the computation of canonical local time map, ω_p^c as in Def. 3.7, and we discuss how to connect it with the concrete scheduler implementation.

TDMA Scheduling for Partitions. We first focus on the TDMA-based partitioned scheduling as in ARINC 653 standards, and show how to prove its schedulability, as well as non-interference using the virtual time.

Under this setting, the schedule of a partition is fixed. In fact, it employs the same period, P, for all partitions, and repeats the same partition schedule in every period. Assume the scheduling offset of partition Π_i is δ_i. Then in its k-th period, it is scheduled during the interval $[(k-1)P + \delta_i$, $(k-1)P + \delta_i + B_i)$. We use π_i to denote the schedule of Π_i. We assume Π_i uses the same fixed-priority scheduling for its tasks as discussed in Sec. 4.4. Scheduling within other partitions are irrelevant with respect to Π_i.

Notice that under TDMA, the schedule of a partition is known ahead of time.

```
int TDMA_sched() {
  t++;
  if ($\delta_i \leq t \mod P < \delta_i + B_i$){
    for(int i = 0; i < N; i++){
      if (t % period[i] == $\delta_i$){
        quanta[i] = budget[i];
      }
    }

    ......
    // fixed-priority scheduling
    // within $\Pi_i$
  }
}
```

Fig. 15. C code for the TDMA scheduler

```
int vt_TDMA_sched() {
  t++;
  if ($\pi_i(t+1) > \pi_i(t)$){
    int lt = $\pi_i(t)$;
    int pid = N;
    for(int i = 0; i < N; i++){
      if (VTA($\omega_i^c$, lt)){
        pid = i;
        break;
      }
    }
    return pid;
  }
  ......
}
```

Fig. 16. The virtual-time-based TDMA scheduler

Definition 4.6 (Partition schedule under TDMA).

$$\pi_i(t) = \begin{cases} 0 & t < \delta_i \\ \left\lfloor \dfrac{t - \delta_i}{P} \right\rfloor B_i + min(B, t - \delta_i - \left\lfloor \dfrac{t - \delta_i}{P} \right\rfloor P) & t \geq \delta_i \end{cases}$$

Now we show the equivalence between the concrete scheduler (Fig. 15) and the virtual-time-based scheduler (Fig. 16). Similar to Sec. 4.3, their equivalence relies on the following relation.

Definition 4.7 (Equivalence of states for partitioned scheduling).

$$\texttt{quanta}[p] = O_p(\left\lfloor \frac{\pi_i(t)}{T'_p} \right\rfloor) - \min\left(O_p(\left\lfloor \frac{\pi_i(t)}{T'_p} \right\rfloor), \omega^c_p(\pi_i(t)) - \omega^c_p\left(\left\lfloor \frac{\pi_i(t)}{T'_p} \right\rfloor T'_p\right)\right), \; where \; T'_p = \frac{T_p}{P} B_i$$

We prove in Coq that this relation always holds, matching states of the two schedulers with each other. The proof is similar to that of Sec. 4.4, while the only subtlety is that we need to match the beginning of new periods in both schedulers. This is achieved by proving that whenever $t \mod T_i = \delta_i$, the following holds: $\pi_i(t) \mod T'_i = 0$. This exploits the fact that T_p is a multiple of P, and indicates that "arrival" events in both global time and partition local time are indeed synchronized.

Dynamically scheduled partitions. The scheduling scheme mentioned above is simple because its partition schedule is fixed. However, many partitioned scheduling algorithms result in a varying schedule of partitions, in order to achieve better average case responsiveness [Davis and Burns 2005; Sprunt et al. 1989; Xi et al. 2011]. They usually still enforce the budget on each partition, i.e. each partition only receives a portion of CPU resource specified by P and B_i. Yet, they differ in the actual way of implementing this enforcement.

Here, we focus on a global scheduler with partition budget enforcement such as the framework presented in [Kim et al. 2015], as shown in Fig. 17. It is very similar to the flattened fixed-priority scheduler, except that it also maintains an array, Q, for each partition's budget. We omit code pieces that are similar to Fig. 2. The virtual-time-based counterpart of this global scheduler is shown in Fig. 18. Here, the time map for the partition is computed dynamically, while the local time maps for tasks are computed statically.

We prove the equivalence of both schedulers relying on the same relation as Def. 4.7. Since the partition is dynamically scheduled in this case, we need to prove that a partition is indeed scheduled for its full budget, B_i, in each period. For simplicity, we assume all partitions have the

```
int global_sched() {
  t++;
  if (t % P == 0){
    for(int j = 0; j < numPar; j++){
      Q[j] = $B_j$;
    }
  }
  ......
  // Partition $\Pi_i$ is chosen
  Q[i]--;
  ......
  // fixed-priority scheduling
  // within partition $\Pi_i$
  ......
}
```

Fig. 17. C code for the global scheduler

```
int vt_global_sched() {
  t++;
  ......
  // Partition $\Pi_i$ is chosen
  $\pi_i$[t+1] = $\pi_i$[t] + 1;
  int lt = $\pi_i$[t];
  int pid = N;
  for(int i = 0; i < N; i++){
    if (VTA($\omega_i^c$, lt)){
      pid = i;
      break;
    }
  }
  return pid;
}
```

Fig. 18. The virtual-time-based global scheduler

same period, and prove the schedulability of partitions by showing that the sum of all remaining partition budgets always decrement until a new period arrives.

4.6 Non-interference Between Tasks

Our isolation guarantee relies on a proof of information flow non-interference. Non-interference is a property that guards the way information is transmitted among different components in a system. In particular, if two components are proved to be non-interfering with each other, then any modification to the state of one of them cannot alter the execution of the other. (Inter-component communications are prohibited, which would otherwise lead to direct interference between components.) This is of significant value in safety-critical systems, where critical components must never be affected by non-critical ones. Existing work mostly emphasizes spatial isolation, e.g. system calls do not leak information, so that non-interference between components is preserved. In our work, the extension with preemption brings more complexity regarding temporal isolation.

Temporal isolation refers to the absence of information flow through scheduling, that is, a component's observation on its own schedule (e.g. schedulability of a task in flattened scheduling and the local schedule of tasks in partitioned scheduling) is not affected by others. In particular, a component's observation does not include how often or how long it is preempted. Indeed, the current implementation of our scheduler does not allow a component to observe the number or duration of preemptions: timer interrupts and context switches are handled by the OS kernel, and a user task cannot access the global time.

For example, in flattened fixed-priority scheduling, higher-priority tasks can choose to either use up all the budget or yield early during a period, causing lower-priority ones to be preempted longer (and more often) or shorter (and less often), respectively. However, since real-time tasks must satisfy the schedulability test on their task parameters (budget and period), the schedulability of lower-priority tasks always holds thanks to the budget-enforcing mechanism of our scheduler. And since a low-priority task cannot observe preemptions directly (a task is scheduled by the OS kernel according to global time and status of higher-priority tasks, but that information is not exposed to this user task), there is no information leakage from higher-priority ones through scheduling. In other words, the low-priority task's observation on its schedule, which only includes its schedulability property, is not affected by others.

As another example, in partitioned scheduling, the exact schedule of a partition may change depending on the behavior of other partitions, e.g. it may be preempted in the middle by another partition whose task has higher priority than what it has run. However, this partition can enjoy temporal isolation if the local schedule of tasks remains the same regardless of how other partitions execute. This was discussed in Sec. 3.2 and Sec. 4.5.

We define the above notion of temporal isolation to allow flexibility in the global schedule of a component, which leads to better responsiveness of tasks and higher utilization of the processor, compared to a straightforward but rigid solution that makes the schedule of all tasks static. As a consequence, access to global time or time-leaking external devices is prohibited. External devices such as network cards could leak information about global time directly (e.g. by attaching a timestamp to a message) or indirectly. For example, if one knows the expected arrival rate of network packets, then that gives a pretty good indication of the passage of global time by counting how many packets have arrived. Also, as discussed in Sec. 1.3, microarchitectural level details, such as nondeterminism in the number of instructions a component executes during a time slot, are out of the scope of this work. We only focus on algorithmic properties instead.

In this section, we explain how we extend the proof of non-interference between tasks to the case studies described in Sec. 4.3 and Sec. 4.4, and we show how our framework accommodates the isolation proof with the presence of implicitly shared resources.

Non-interference of cooperative CertiKOS. The existing cooperative version of CertiKOS enjoys a mechanized proof of non-interference by Costanzo et al. [2016]. Unlike the usual approach based on security labels [Zeldovich et al. 2006], it instead defines non-interference with respect to an *observation function* $\mathfrak{O}$, which is a function from a state to an abstract observation that intuitively represents everything that can be observed, directly or indirectly. We call two states *observably equivalent* or *observably indistinguishable* regarding an observer task if the task's observation function returns identical values on both states. Non-interference is then defined as "preserving (observational) indistinguishability":

Definition 4.8 (Non-Interference). A system is said to be *non-interferent* if its execution relation step preserves indistinguishability:

$$\forall s_1\, s_2\, s_1'\, s_2',\ \mathsf{step}\ s_1\, s_1' \Rightarrow \mathsf{step}\ s_2\, s_2' \Rightarrow \\ \mathfrak{O}(s_1) = \mathfrak{O}(s_2) \Rightarrow \mathfrak{O}(s_1') = \mathfrak{O}(s_2')$$

The interest of this definition is that it makes declassification extremely natural: just let the observation function return the declassified form. For instance, we can declassify the map from virtual addresses to memory content without leaking the physical addresses. The non-interference proof is carried down to the assembly code, thanks to security-preserving simulation relations [Costanzo et al. 2016].

The non-interference proof of CertiKOS takes advantage of the fact that the only way to trigger a context switch is through the `sys_yield` system call, in particular, that context switches always happen at the same places.

Structure of the non-interference proof of cooperative CertiKOS.

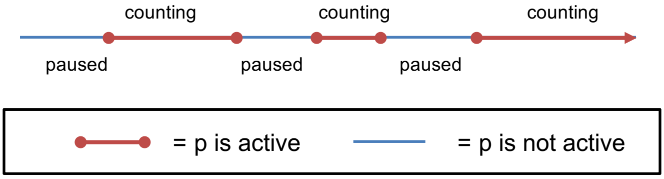
- The behavior of a task. As explained above, Costanzo et al. [2016] define a task's whole-execution behavior on top of its observation function.
  - Observation function. A task's observation includes its register state, virtual memory space, memory quota, number of children tasks, and the output buffer.
  - Whole-execution behavior. It is specified with the execution status, e.g. Terminate/Fault, etc. and a task's observation of its final state. On top of this, the whole-execution-based isolation property of a task is defined as follows. A task enjoys non-interference if starting from two observably equivalent initial states, it exhibits observably equivalent whole-execution behavior, e.g. the two executions may terminate at observably equivalent final states.

- The non-interference lemmas and frame rule. The core isolation theorem in Costanzo et al. [2016] states that whole-execution behaviors from observably equivalent initialized states s_1 and s_2 are identical. It specifies non-interference on a particular level.
  This is more of a frame rule, such that if a particular layer satisfies a group of predicates, most importantly confidentiality (observably equivalent states step to observably equivalent next states) and integrity (an inactive task's state is not tempered with by others), then the above statement is true. Notice that in this cooperative setting, context switches only happen upon voluntary yield, and this frame rule requires that two executions of a task be matched step-by-step.
  - Proof burden involving system services. Various properties, such as confidentiality and integrity, are proved on all system services when the aforementioned frame rule is instantiated at the top level of cooperative CertiKOS. System calls available to a task include querying its memory quota, spawning new tasks, yielding the current execution, and printing to its buffer.
- The end-to-end non-interference proof. The main theorem is an end-to-end property, stating that if two high-level initialized states, S_1 and S_2, are observably equivalent, then starting from the two corresponding low-level states, s_1 and s_2, the whole-execution behaviors are also observably equivalent. Its proof is mainly about showing that such security property is preserved by refinement, which is one of the main contributions by Costanzo et al. [2016]. This proof relies on the frame rule mentioned above.

Non-interference with preemption. Preemption breaks this existing proof as context switches can be triggered by interrupts that do not need to occur at the same place in all executions. In particular, we now need to prove that an execution in which a context switch is triggered is observably equivalent to an execution in which no context switch happens, which entails the correctness of context switch. These requirements exercise the specification of the context switch routine, and in doing so discovered that the original specification was not tight enough: the behavior of floating-point registers and the control register is undefined, which prevents us from establishing the equivalence of states before and after a context switch. Notice however that the existing specification was non-interferent, so that the overall non-interference proof of the cooperative CertiKOS is still correct, albeit less meaningful.

To that end, we narrow down the original specification to accurately reflect all details in the context switch. We modify the observation function to be insensitive to the task status, thus allowing to preserve indistinguishable states between a running execution and a preempted one, provided the latter gets the values of registers from its saved context. A task enjoys all system calls available in Costanzo et al. [2016] (newly-spawned tasks are always batch tasks), in addition to involuntary context switches triggered by preemption. These system services do not rely on explicitly querying another task's running status, thus they are compatible with the updated definition of observation. Notice that access to global time and communications between tasks are prohibited as a consequence of strict task-level isolation. A user task is still able to perform operations periodically without access to global time since the OS kernel is responsible for triggering new task periods according to global time.

We then extend the isolation frame rule which proves whole-execution non-interference based on single-step lemmas and instantiate it with various proofs on individual system calls and the scheduler. For example, we prove the integrity of all system calls, such that if a task is inactive (preempted), its observation doesn't change. We also prove the confidentiality of the scheduler, such that stepping from two observably equivalent active states, the next states are observably equivalent as well. Unlike in cooperative CertiKOS, confidentiality here does not require the two



```
int get_usage() {            void stop_counting() {      void start_counting() {
  int p = get_curid();         int p = get_curid();        cur = rd_counter();
  int u = rd_counter();        int u = rd_counter();     }
  return (sum_p + (u - cur));  sum_p += u − cur - ctxt;
}                            }
```

Fig. 19. Resource Accounting as Stopwatches

next states to be both active or both inactive. Instead, we prove them to be equivalent regardless of whether this task is preempted or not.

The `get_usage` *system call.* To illustrate the flexibility of our non-interference proof to address implicitly shared resources, we add `get_usage` system call. An *implicitly shared resource* is a resource whose usage is inherently tied with program execution and cannot be physically isolated, such as execution time, I/O, and instruction count. Doing so offers more measurement mechanism for the user, but also introduces a whole new category of side-channels related to the usage of this resource. However, we prove that under certain restrictions, queries of such kind of resource usage are also non-interfering.

Resource accounting relies on a cost model that associates with each instruction its cost for the resource under consideration. On the one hand, in order to avoid blatant information leakage when accessing the current total cost, we need to separately keep track of each observer's resource usage and only give each observer access to its own resource usage. On the other hand, resource accounting can be available only as global hardware monitoring, for instance, timing information is available through hardware timers. Thus, we carefully associate each resource usage to its user, through a stopwatch mechanism, depicted in Fig. 19.

The extension of the observation function is straightforward: we add a new field returning the value of the system call. Once the observation function has been updated, the non-interference proof presents no challenge: we only deal with the new case for the `get_usage` system call and update the proof of `sys_yield` (which modifies the stopwatch information).

5 EVALUATION

We build on top of CertiKOS, a fully verified preemptive OS kernel with both spatial and temporal isolation. The extracted assembly code is further compiled by gcc into machine code, and runs on an Intel Sandy Bridge based x86 machine.

5.1 Proof Effort

For the case studies in Sec. 4.3 and Sec. 4.4, we prove schedulability analysis of tasks, and non-interference at the task level. For the case studies in Sec. 4.5, we prove that the schedulability analysis of tasks is correct and that the local schedule within a partition is independent of others.

More detailed changes are summarized below:

- Extension of CertiKOS with user-level preemption, a verified interrupt handler (Sec.4.2), and a verified preemptive scheduler.

Feature	Breakdown	LoC	Subtotal
Preemption	Updates on the Machine Model & Framework		**4,975**
⓪ Fixed-Priority Scheduler	Data Structures & Lemmas	1,750	**23,671**
	Primitive Definitions & Lemmas	5,373	
	Layer Defs, and Functional Correctness Proof	16,548	
① Virtual Timelines	Arithmetic Lemmas	644	**2,041**
	Definitions & Properties of Virtual Timelines	1,397	
②③ Virtual-Time-Based Scheduler	Data Structures & Lemmas	3,383	**6,134**
	Layer Definitions & Refinement with `abs_sched`	2,751	
Propagation to Top Level	Timer Interrupt Handler & Misc		**4,732**
Non-Interference	Confidentiality of Kernel Services	8,599	**16,883**
	Integrity of Kernel Services	5,103	
	Security Framework & Main Theorem	3,181	
⓪②③ Partitioned Scheduling under TDMA	Data Structures & Lemmas	1,066	**4,597**
	Primitive Definitions & Lemmas	748	
	Layer Defs, and Functional Correctness Proof	2,783	
⓪②③ Partitioned Scheduling with Global Scheduling	Data Structures & Lemmas	2,421	**13,764**
	Primitive Definitions & Lemmas	1,940	
	Layer Defs, and Functional Correctness Proof	9,403	
Grand Total			**76,797**

Fig. 20. Changes with respect to the sequential version of CertiKOS

- Functional correctness proof of the preemptive scheduler and the `sys_finish` system call (step ⓪ in Fig. 1).
- Formalization of virtual timelines, and algorithmic reasoning on top of it, including schedulability analysis and obliviousness properties (Sec. 3, also step ① in Fig. 1).
- Formalization of the virtual-time-based scheduler, its refinement proof with the concrete scheduler, and its formal connection with individual virtual timelines (Sec.4.3, 4.4, and 4.5, also step ②③ in Fig. 1).
- Non-interference between tasks. We extend the framework of previous work by Costanzo et al. [2016] to a preemptive scheduler (Sec. 4.6) and fix an existing loose specification in the kernel.
- The functional correctness proof of the `get_usage` system call, while preserving the non-interference property.

Fig. 20 provides a more detailed insight into the proof efforts involved. We have mainly used the one-tactic-per-line style of proof script; and we have also relied on automation techniques and Ltac scripts to simplify the proofs. The numbers in Fig. 20 are estimated LoC of changes against the original version of CertiKOS, obtained by analyzing the result of running `git diff` with `--numstat` option. Notice that we only count modified lines once in our estimation: one insertion and one deletion are merged to represent modification on one line. We avoid counting duplicated and reused proofs. For example, the proof for partitioned TDMA scheduler makes use of definitions and lemmas developed for the simple fixed-priority scheduler, yet we did not include those LOC in the statistics for TDMA. From this table we can observe that:

- The functional correctness proof of the fixed-priority scheduler takes more than 16k LoC, partly due to the fact that we introduce many intermediate layers to simplify the proof, thus we also have to duplicate many invariant proofs across those layers.
- The formalization and reasoning about virtual timelines take around 2k LoC, which indicate that this is indeed a suitable abstraction for reasoning about algorithmic properties of fixed-priority preemptive scheduling.
- The virtual-time-based scheduler serves as a bridge connecting individual virtual timelines with the concrete scheduler in the system. It includes more than 3k LoC proof for data structures and lemmas mainly involving VTA (Sec. 4.3), which we believe is not specific to CertiKOS, and is applicable to other systems so that they could also benefit from this abstraction of virtual timelines.

Lessons learned. The main challenge with formalizing a preemptive scheduling algorithm is to find a suitable abstraction that facilitates the connection between the abstract scheduling behavior (e.g. virtual timeline in this work) and the concrete scheduler. We discuss possible choices of defining the virtual time map in Sec. 2.3, and our final decision leads to a simple form of simulation relation between the two schedulers, as shown in Def. 4.1.

Mechanized proofs allow us to address rigorous requirements on a complex system, such as schedulability in this work. However, the heavy proof burden coming with it makes it challenging to address complex mathematical reasoning, such as probability-based analysis if soft real-time scheduling algorithms are used. In this work, we select fixed-priority scheduling since it is relatively simple but still meaningful and practical for hard real-time system.

5.2 Subtleties Found in the Kernel

During the proof development, we found certain subtleties in the implementation of the kernel, which either warns us of limitations of our semantics models, or helps us prevent bugs that could invalidate the whole system.

The timer counter. The kernel uses a timer counter to keep track of the number of timer interrupts that have occurred, providing a time measure for the scheduler. However, since the system can run indefinitely, a fixed-sized counter (which we call a bounded counter) will eventually overflow.

We observe that the computation of task arrivals is identical after each *hyper-period*, that is, the least common multiple of periods of all tasks. We then introduce an unbounded counter in the system, which resides only in the abstract state. The unbounded counter keeps increasing, while the bounded one is reset every time it hits the hyper-period. We prove that as long as these two counters are equal modulo the hyper-period, the result of the scheduler is the same, and all proofs on temporal behaviors still hold.

A context switch subtlety. As mentioned in Sec. 4.6, the context switch primitive can satisfy non-interference in a cooperative setting without being tight at all. And this is actually exactly what happened: the existing context switch primitive of CertiKOS exhibits an undefined behavior regarding the CR register (where the test flags are set for instance), although it does satisfy non-interference in a cooperative setting.

The fact that non-interference with preemption requires to prove that the saved and restored contexts are identical forces us to strengthen this specification.

6 LIMITATIONS & FUTURE WORK

User-level preemption. Similar to [Sigurbjarnarson et al. 2018], our current extension on CertiKOS only allows for preemption at the user level. This assumption simplifies the correctness proof of

CertiKOS, which assumes that kernel functions are executed atomically. However, the downside is that the lack of kernel preemption may result in longer interrupt disable time (interrupt latency), lowering the responsiveness of the system. Nevertheless, this is not an inherent limitation and could be mitigated in future work.

One solution is to adopt the bottom-halves mechanism (as used in μC/OS-III [Labrosse 2011]) and offload time-consuming work to user mode, effectively bounding the latency introduced by kernel services. This approach achieves better responsiveness, while still only requiring preemption in user mode so that the current functional correctness proof of CertiKOS would still be applicable.

It should also be noted that our reasoning framework is compatible with the future extension of kernel preemption. Indeed, enabling kernel preemption would require a careful handling of concurrent accesses to kernel objects even on a single-core machine, resulting in more complexity in both the kernel implementation and its proof. However, in a setting in which system calls do not block each other (which is reasonable for a single-core real-time system), kernel preemption is largely irrelevant to the scheduler. The scheduler is still invoked periodically by timer interrupts and selects the highest-priority available task in the system. Likewise, the schedulability and obliviousness properties proved on the scheduler still hold regardless of whether kernel preemption is supported or not. In this sense, our reasoning framework is compatible with verified single-core preemptible kernels.

Kernel overhead. A real-time OS kernel relies on the timer interrupt, which is triggered at a constant rate, to preempt user tasks. However, if the interrupt happens in the middle of a system call, its handling will be delayed until control is given back to user mode. Similarly, the interrupt handler itself also consumes time. In both cases, these small overheads prevent a task from executing for its full budget.

Ignoring kernel overhead is a fairly common assumption for schedulability analysis in a first approximation. However, such overhead can be accounted for without having to update our existing framework. At any timer interrupt, the maximum delay incurred by the kernel is the interrupt latency plus the context switch overhead. Given a trustworthy WCET on both overheads (which can be obtained by a sound analysis if the underlying real-time processor exhibits predictable timing behavior or a conservative estimation in case of a general-purpose processor), we can characterize the amount of recurring interference as another task with higher priority, a period that equals to that of the timer interrupt, and the worst-case execution time that equals to the sum of the maximum interrupt latency plus context switch overhead. In this way, the analysis of the interference of both overheads fit into the existing framework.

Nevertheless, as we do not allow real-time tasks to invoke certain system calls such as `sys_spawn`, and the C code of the rest of our system calls is simple enough (they are essentially straight-line programs, and we measure their average execution time to be $0.47\mu s$ on a 2.8 GHz machine), a statistical analysis should give a reasonable estimate.

Interrupt Driven Tasks. Currently, we only allow timer interrupts to preempt a task. However, users are able to sample a device periodically for interrupts or new data, which is common in time-critical cyber-physical systems [Murray et al. 2013; Sigurbjarnarson et al. 2018]. Thus the current periodic task set is already a reasonable setting for verification work.

One possible extension is to consider sporadic tasks, or interrupt driven tasks, that are not strictly periodic but triggered by interrupts. It is not difficult to support these tasks in our reasoning framework. The critical instant theorem we prove (Thm. 3.2) states that as long as the schedulability test is satisfied, a task enjoys its full budget within any physical time window whose length equals its period, no matter when this time window starts. This already entails schedulability in case of interrupt-driven tasks, whose minimum inter-arrival time, called period in real-time literature, is bounded and known. Hence, if a task is analyzed to be schedulable with the assumption of the minimum inter-arrival time, it is always schedulable as long as actual inter-arrival times are not lower than the bound. We leave this extension for future work.

7 RELATED WORK

Verification of schedulability analysis. The first mechanized proof about schedulability proves optimality of EDF [Wilding 1998]. Dutertre [2000] uses a state machine model in the automated theorem prover PVS to describe and verify the behavior of the priority ceiling protocol. Zhang et al. [2012] formally specify the correctness requirement (no priority inversion) of priority inheritance protocol using Isabelle/HOL, and prove the correctness of an implementation, but temporal properties are not modeled.

The PROSA [Cerqueira et al. 2016] project demonstrates readable mechanized proofs of various schedulability analyses written in Coq. It defines an abstract model of real-time tasks and real-time schedulers, and proves that all tasks will complete on time if certain conditions hold. Guo et al. [2019] connect PROSA with the real-time extension of CertiKOS described in Sec. 4.2, and obtains a schedulability proof for it. This connection requires a ghost variable to map kernel states to Prosa schedules and is based on the proof that the scheduler indeed follows a given scheduling policy. Unlike us, they only prove the schedulability of tasks under fixed-priority scheduling and do not address isolation. Our work highlights a suitable abstraction with isolation in mind. In addition to fixed-priority scheduling, our work also addresses partitioned scheduling, which involves more subtle isolation issues.

Verification of OS kernels. For a full overview of the verification efforts about operating systems, we refer the reader to Klein [2009] and Zhao et al. [2017]. Among all the OS verification works, we place our emphasis mainly on those conducted at the C or assembly code level.

SeL4 [Klein et al. 2009] is a formally-verified operating system kernel. It achieves multitasking in the kernel by polling timer interrupts at specified preemption points. Its machine model includes interrupts as abstract events and user transitions as arbitrary non-deterministic updates to the state, and the correctness of assembly routines is assumed [Sewell et al. 2013]. As a comparison, the machine model of this work is an extension of the CompCert assembly semantics, with checking on interrupt lines and also incorporated instruction level resource measurement such as instruction count. Most of the assembly routines are verified against this more realistic machine model.

An early extension of CertiKOS [Chen et al. 2016] tackles the problem of an interruptible kernel. By abstracting away interrupts using an approach based on abstraction layers, it proves a simulation relation between different interrupt models, and eventually achieves a model that is suitable for reasoning about device drivers in an interruptible kernel. Concurrent CertiKOS (known as mC2) [Gu et al. 2016, 2018] is the first verified kernel that can support multicore concurrency with fine-grained locking but its scheduler is still not preemptive.

Nemati et al. [2015] prove the design of a hypervisor based on ARMv7 with isolation properties, including caches. Unlike us, however, there is no verified implementation, as the proofs are only done at the specification level.

Nelson et al. [2017] report the push-button verification of a kernel using the z3 SMT solver. To achieve that though, they must make compromises on the interfaces to make them amenable to automated verification by removing all quantifiers.

Verification of real-time OS kernels. INTEGRITY-178B is, to the best of our knowledge, the first real-time kernel to enjoy a machine-checked mathematical proof of non-interference [Richards 2010], as well as the avionics certification DO-178B level A. Nevertheless, this proof only applies to a model of the kernel, and has no formal link with its implementation.

The eChronos OS [Andronick et al. 2016] is a real-time operating system designed for uniprocessor embedded systems. It enables interrupts in most parts of the kernel to achieve minimum latency. It uses an interleaving model of the application code and different OS components [Andronick et al. 2015] and proves that the currently running process always has the highest priority among ready tasks. This work only models the behavior of different components and hardware interrupts without an end-to-end proof.

Xu et al. [2016] verify μC/OS-II [Labrosse 1998], a real-time operating system featuring a preemptive kernel, using an ownership-transfer semantics. They prove the contextual refinement between the system's implementation and specification; they also demonstrate the ability to prove high-level properties, such as priority inversion freedom. However, it does not address more sophisticated high-level properties such as schedulability or temporal isolation.

Enforcement of isolation properties. Murray et al. [2013] prove a non-interference property for the seL4 kernel, which schedules partitions in a round-robin manner, and each partition adopts preemptive priority-based scheduling for its tasks. The non-interference proof depends on the setting that partitions are scheduled statically, so that there is no information flow from a partition to the scheduler. In contrast, our work focuses on formalizing a different problem in which partitions can be scheduled dynamically (e.g. it can depend on the global priority of its task), such that there can be information flow from a partition to the scheduler, yet the local schedule of tasks within one partition remains uninfluenced by others. Our work also addresses the schedulability analysis of real-time tasks, which is essential for temporal isolation.

Sigurbjarnarson et al. [2018] prove intransitive non-interference of a system, so that only information flow explicitly allowed in the flow control graph can occur in the system. They implement NiStar, an OS kernel with decentralized information flow control, and prove that there is no covert channel among processes through the scheduler. This is achieved by statically assigning each time quantum to a thread, and a thread cannot transfer its quantum to others unless allowed by the information flow policy. It also addresses the partition isolation of a formalization of the ARINC 653 specification, which assumes a fixed schedule for partitions, in contrast to the dynamic partition schedule we focus on in this work.

Lyons et al. [2018] propose a mixture of capability-based resource management and sporadic-server-based scheduling. They provide to the user an interface for reserving a portion of CPU time for a specific critical task. They report 2k lines of code on top of the original implementation of the OS kernel, and they use testing to show that this mechanism indeed constrains the CPU utilization of a particular task. As a comparison, our work focuses on strict temporal isolation, such as schedulability and obliviousness to other components, and we formally prove these properties in Coq with more than 76k LoC. Our work uses periodic servers for partitions, instead of sporadic servers used in [Lyons et al. 2018]. However, the same reasoning framework applies in both cases.

WCET analysis of verified OS kernels. Blackham et al. [2011] conduct a static WCET analysis on the seL4 compiled binary code's interrupt response time, with the help of invariants from the formal proof to exclude infeasible paths. They show that the WCET is sufficiently low without making the kernel fully preemptible. Sewell et al. [2017] achieve automation in determining and proving loop bounds, though manual annotation is still needed in some places.

8 CONCLUSIONS

This paper presents a novel compositional framework for verifying preemptive schedulers with temporal isolation. We introduce virtual timelines to describe temporal behaviors of components and to connect them with the actual system implementation. Using this abstraction, we prove the schedulability of each component in isolation, and prove various isolation properties regarding the schedule of components. In flattened fixed-priority scheduling, we prove that a task's schedule is not influenced by lower priority ones. In partitioned scheduling, we prove that a partition's local schedule of tasks is independent of other partitions, under a constrained setting.

As a case study, we apply this framework to CertiKOS, a formally verified single-core cooperative OS kernel. We extend it with a verified timer interrupt handler and a verified preemptive scheduler. We connect virtual timelines with the concrete implementation of the system by introducing a virtual-time-based scheduler, and proving that it is consistent with both the concrete scheduler and individual virtual timelines. In this way, all properties we prove on virtual timelines carry down to the generated assembly code of the system. We also prove that all services provided by our kernel preserve the integrity and confidentiality of user tasks, by showing that non-interference holds for this kernel. We also implement a secure `get_usage` system call for resource accounting and prove it to be non-interfering, by carefully defining what value it returns. Combining everything together, we achieve both temporal and spatial isolation between different components.

To the best of our knowledge, this is the first work that provides a formal abstraction for verifying the actual implementation of preemptive schedulers with temporal isolation.

ACKNOWLEDGMENTS

We would like to thank anonymous referees for helpful feedbacks that improved this paper significantly. This research is based on work supported in part by NSF grants 1521523, 1715154, and 1763399 and DARPA grants FA8750-16-2-0274 and FA8750-15-C-0082. The second author has been partially supported by the PEPS INS2I JCJC 2019 Vefose grant. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan, and Christine Rizkallah. 2016. Proof of OS Scheduling Behavior in the Presence of Interrupt-Induced Concurrency. In *Proceedings of 7th International Conference on Interactive Theorem Proving (ITP)*. Springer International Publishing, Nancy, France, 52–68. https://doi.org/10.1007/978-3-319-43144-4_4

June Andronick, Corey Lewis, and Carroll Morgan. 2015. Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible eChronos Embedded Operating System. In *Proceedings of 2015 Workshop on Models for Formal Analysis of Real Systems (MARS)*. EPTCS, Suva, Fiji, 10–24. https://doi.org/10.4204/EPTCS.196.2

ARINC. 2015. *ARINC Specification 653 Part 1.* ARINC, Annapolis, MD.

B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. 2011. Timing Analysis of a Protected Operating System Kernel. In *2011 IEEE 32nd Real-Time Systems Symposium (RTSS'11)*. IEEE Computer Society, Washington, DC, 339–348. https://doi.org/10.1109/RTSS.2011.38

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-end. In *Proceedings of the 14th International Conference on Formal Methods (FM'06)*. Springer-Verlag, Berlin, Heidelberg, 460–475. https://doi.org/10.1007/11813040_31

F. Cerqueira, F. Stutz, and B. B. Brandenburg. 2016. PROSA: A Case for Readable Mechanized Schedulability Analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS'16)*. Schloss Dagstuhl, Germany, 273–284. https://doi.org/10.1109/ECRTS.2016.28

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16), Santa Barbara, CA, USA, June 13-17, 2016.* ACM, New York, 431–447. https://doi.org/10.1145/2908080.2908101

David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16), Santa Barbara, CA, USA, June 13-17, 2016.* ACM, New York, 648–664. https://doi.org/10.1145/2908080.2908100

R. I. Davis and A. Burns. 2005. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. IEEE Computer Society, Washington, DC, USA, 389–398. https://doi.org/10.1109/RTSS.2005.25

B. Dutertre. 2000. Formal analysis of the priority ceiling protocol. In *Proceedings 21st IEEE Real-Time Systems Symposium (RTSS'00)*. IEEE Computer Society, Washington, DC, 151–160. https://doi.org/10.1109/REAL.2000.896005

Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys'19)*. ACM, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/3302424.3303976

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, GA, 653–669.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, JÃľrÃľmie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, 646–661.

Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. 2019. Integrating Formal Schedulability Analysis into a Verified OS Kernel. In *Computer Aided Verification - 31st International Conference (CAV'19), July 15-18, Proceedings*. Springer, Berlin, Heidelberg, 496–514.

J. Kim, T. Abdelzaher, and L. Sha. 2015. Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*. IEEE Computer Society, Washington, DC, 221–231. https://doi.org/10.1109/RTAS.2015.7108445

Gerwin Klein. 2009. Operating system verification—An overview. *Sadhana* 34, 1 (01 Feb 2009), 27–69. https://doi.org/10.1007/s12046-009-0002-4

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

Jean J. Labrosse. 1998. *Microc/OS-II* (2nd ed.). Focal Press, New York.

Jean J. Labrosse. 2011. *Microc/OS-III*. Micrium Press, Austin, TX.

J. Lehoczky, L. Sha, and Y. Ding. 1989. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings. Real-Time Systems Symposium (RTSS'89)*. IEEE Computer Society, Washington, DC, 166–171. https://doi.org/10.1109/REAL.1989.63567

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://doi.acm.org/10.1145/1538788.1538814

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2014. The CompCert memory model. In *Program Logics for Certified Compilers*, Andrew W. Appel (Ed.). Cambridge University Press, Cambridge, UK. http://vst.cs.princeton.edu/

C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

Jane W. S. W. Liu. 2000. *Real-Time Systems* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-context Capabilities: A Principled, Light-weight Operating-system Mechanism for Managing Time. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*. ACM, New York, NY, USA, Article 26, 16 pages. https://doi.org/10.1145/3190508.3190539

Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy (SP'13), Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, Washington, DC, 415–429. https://doi.org/10.1109/SP.2013.35

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17), Shanghai, China, October 28-31, 2017*. ACM, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

Hamed Nemati, Roberto Guanciale, and Mads Dam. 2015. Trustworthy Virtualization of the ARMv7 Memory Subsystem. In *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*. Springer-Verlag, Berlin, Heidelberg, 578–589. https://doi.org/10.1007/978-3-662-46078-8_48

QNX. 2019. Neutrino RTOS. http://blackberry.qnx.com/en/products/neutrino-rtos/neutrino-rtos

Raymond J. Richards. 2010. *Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel.* Springer US, Boston, MA, 301–322. https://doi.org/10.1007/978-1-4419-1539-9_10

Thomas Sewell, Felix Kam, and Gernot Heiser. 2017. High-assurance timing analysis for a high-assurance real-time operating system. *Real-Time Systems* 53, 5 (01 Sep 2017), 812–853. https://doi.org/10.1007/s11241-017-9286-3

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13).* ACM, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. 2016. Real-Time Computing on Multicore Processors. *Computer* 49, 9 (Sept 2016), 69–77. https://doi.org/10.1109/MC.2016.271

Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18).* USENIX Association, Carlsbad, CA, 287–305.

Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. *Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System.* Technical Report CMU/SEI-89-TR-011. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Matthew Wilding. 1998. A Machine-Checked Proof of the Optimality of a Real-Time Scheduling Policy. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98).* Springer-Verlag, London, UK, UK, 369–378. http://dl.acm.org/citation.cfm?id=647767.733638

S. Xi, J. Wilson, C. Lu, and C. Gill. 2011. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT).* ACM, New York, 39–48.

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *Computer Aided Verification: 28th International Conference (CAV'16), Toronto, ON, Canada, July 17-23, 2016, Proceedings*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Berlin, Heidelberg, 59–79.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation.* USENIX Association, Berkeley, CA, USA, 19–19. http://dl.acm.org/citation.cfm?id=1267308.1267327

Xingyuan Zhang, Christian Urban, and Chunhan Wu. 2012. Priority Inheritance Protocol Proved Correct. In *Interactive Theorem Proving (ITP'12).* Springer, Berlin, Heidelberg, 217–232.

Yongwang Zhao, Zhibin Yang, and Dianfu Ma. 2017. A Survey on Formal Specification and Verification of Separation Kernels. *Front. Comput. Sci.* 11, 4 (Aug. 2017), 585–607. https://doi.org/10.1007/s11704-016-4226-2