Abstract

Real-Time CertiKOS: Compositional Verification of OS Kernels with Preemptive Scheduling and Temporal Isolation

Mengqi Liu

2020

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.

This thesis presents 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.

Real-Time CertiKOS: Compositional Verification of OS Kernels with Preemptive Scheduling and Temporal Isolation

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Mengqi Liu

Dissertation Director: Zhong Shao

May 2020

© 2020 by Mengqi Liu

All rights reserved.

Contents

Acknowledgments **viii**

1 Introduction **1**

1.1 Existing Work on Isolation Between Components 2

1.2 Challenges in Verifying Temporal Isolation 3

1.3 Contributions and Scope of This Thesis 5

2 Overview of the Virtual Timeline Framework **11**

2.1 Overview 11

2.2 Real-Time Scheduling and Virtual Timelines 13

2.3 Formalization of the Virtual Timeline 16

3 Reasoning about Fixed-Priority Scheduling **19**

3.1 The Schedulability Proof 19

3.2 Extension with Variable Execution Time 25

4 Case Study: CertiKOS with Fixed-Priority Scheduling **29**

4.1 Background: CertiKOS, a Verified OS Kernel 30

4.2 Real-Time Extension on CertiKOS 31

4.3 Connecting The Schedulability Proof with The Concrete Scheduler 35

4.4 The sys_finish System Call 39

4.5 Non-Interference Between Tasks 42

5 Reasoning about Partitioned Scheduling **48**

5.1 Background: Partitioned Scheduling 48

5.2 Temporal Isolation in Partitioned Scheduling 50

5.3 Schedulability in Partitioned Scheduling 57

6 Case Study: CertiKOS with Partitioned Scheduling **59**

6.1 Static partitions scheduled under TDMA 59

6.2 Dynamically scheduled partitions 62

6.3 A Perspective on the Interface for a Real-Time Partition 70

7 Generalization with Dynamic Priority Assignment **72**

7.1 Earliest-Deadline-First and Compositionality of Workloads 73

7.2 Generalization: Virtual Timeline with Dynamic Priority Assignment 74

7.3 Earliest-Deadline-First: the Schedulability Proof 77

7.3.1 The Proof Sketch: Schedulability Through Transformation 77

7.3.2 The Virtual Time Map and Interference 81

7.3.3 Enlarging a Task 85

7.3.4 Shrinking a Task Set 96

8 Case Study: CertiKOS with Earliest-Deadline-First Scheduling **102**

8.1 The Concrete Scheduler Implementation 102

8.2 Refinement with the Virtual-Time-Based Scheduler 103

9 Related Work **107**

9.1 Mechanized Schedulability Analysis 107

9.2 Verification of OS kernels 108

9.3 Enforcement of Algorithmic Level Isolation Properties 110

9.4 Intransitive Non-Interference 111

9.5 Microarchitectural Level Isolation 113

10 Limitations, Future Work, and Conclusion **115**

10.1 Tasks with Dependencies 115

10.2 Constraint-Based Scheduling 118

10.3 Multicore Scheduling 120
10.4 Communicating Tasks 121
10.5 Kernel Overhead 123
10.6 Interrupt Driven Tasks 124
10.7 Microarchitectural-Level Interference 125
10.8 Conclusions 126

List of Figures

2.1 Fitting the virtual timeline framework into the verification of an OS kernel 12
2.2 C code of the concrete scheduler 14
2.3 A concrete schedule 14
2.4 Comparing schedule on the global timeline (left) with the schedule on the virtual timeline (right) 15
2.5 The vertical computation of time maps 17
2.6 A concrete time map 17

4.1 Coq formalization of necessary data structures 33
4.2 Coq formalization of the abstract scheduler 34
4.3 Coq formalization of the abstract scheduler 35
4.4 C Illustration of the virtual-time-based scheduler (actually defined in Coq) 36
4.5 An example schedule in global and virtual time 36
4.6 The `sys_finish` system call 40
4.7 C illustration of the virtual-time-based `sys_finish` 40
4.8 The noninterference framework with preemption 47

5.1 Partitioned scheduling 49
5.2 The two-level scheduling 50
5.3 Example for global scheduling 50
5.4 Another example of local time map for a task 52
5.5 Another example of local time map for a task 55
5.6 A counter example against work-conserving scheduling 56

6.1 The top-level TDMA scheduler . . . 60
6.2 The local scheduler for partition Π_i . . . 61
6.3 C illustration of the virtual-time-based local scheduler for Π_i . . . 61
6.4 The intermediate local scheduler for Π_i . . . 61
6.5 The global scheduler for dynamic partitions . . . 64
6.6 The local scheduler for a partition . . . 65
6.7 The intermediate version of the local scheduler for a partition . . . 66

7.1 The concrete and virtual-time-based scheduler . . . 75
7.2 Enlarging all tasks to a common period . . . 79
7.3 The sequence of enlarging operations to be applied to a task set . . . 79
7.4 Apply a sequence of enlarging operations to a task set . . . 80
7.5 The computation of temporal interferences embedded in a virtual-time-based scheduler . . . 82
7.6 The amount of interference τ_j incurs on τ_i: a breakdown . . . 83
7.7 Categorization of types of the schedule . . . 87
7.8 Whole-period interference from τ_j to τ_i before and after the enlargement . 91
7.9 When the schedule of Π' is busy, it leaks less remaining workload to the next period compared to the schedule of Π . . . 92
7.10 When the schedule of Π' is not busy, it leaks less remaining workload to the next period compared to the schedule of Π . . . 93

8.1 The C implementation of an EDF scheduler . . . 103
8.2 Coq formalization of the priority queue . . . 104
8.3 The iteration over a priority queue, used in the intermediate abstraction . . 105

10.1 Illustration of the dynamic computation of time maps for m identical cores 120

Acknowledgments

First and foremost, I owe many thanks to my advisor, Zhong Shao, who guided me with his patience, insight, and taste. I benefited a lot from insightful and sparkling discussions with him during the research, and also from his support and guidance on how to become a good researcher. He made my years in graduate school exciting and unforgettable.

I would also like to express my sincere gratitude to my committee members, Ruzica Piskac, Man-Ki Yoon, and Jan Hoffmann for valuable discussions and feedback from them. I also learned a lot from the software analysis & verification course taught by Ruzica Piskac.

Further, I'd like to thank all past and current members of the Yale FLINT group. They are wonderful people to work with. Special thanks go to Ronghui Gu, who offered help since the initiation of this line of research. This work also benefited a lot from the collaboration with Lionel Rieg, David Costanzo, Jung-Eun Kim, Man-Ki Yoon, and Hao Chen. Yuting Wang, Wolf Honore, and Lucas Paul provided valuable feedback for its improvement. Newman Wu, Quentin Carbonneaux, Jieung Kim, and Jérémie Koenig offered help every time I ran into obstacles in the Coq proof.

Last but not least, I'd like to thank all my friends throughout grad school, for their companion and all the fun together. I want to thank my parents for their constant encouragement and support. And I thank my wife, Che, for her love, support, and the happiness she brought.

Chapter 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. Such constraints are usually specified as the *schedulability* of a periodic task, meaning that a task must be scheduled for a certain amount of time (*budget*) within each of its *periods*. Here, the period of a task is an interval which dictates when this task should repeat itself.

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, different components compete with each other for time resources, leading to inevitable and intricate interferences between them, which 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 Existing Work on Isolation Between Components

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. Establishing an isolation guarantee is challenging because it requires a comprehensive analysis of all possible sequences of interactions. Furthermore, due to the complexity of the concrete implementation of OS kernels, formal verification is the only trustworthy way of guaranteeing functional correctness and isolation properties on the source code level.

Functional Correctness and Spatial Isolation Traditionally, verification of OS kernels emphasizes mostly functional correctness [1, 2] and *spatial isolation* [3, 4, 5]. Thanks to hardware mechanisms such as virtual memory, and by carefully managing resources such as memory quotas and process IDs (namespace) 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 [6]. At a 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 [7], memory banks [8], 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 [9, 10] 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 resources. 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 [11, 12] conduct worst-case execution time analysis on the OS kernel to compute an upper-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. Existing work [5] reason about a simple setting where tasks follow a static schedule to avoid interferences among each other. However, real-time systems require much stronger guarantees.

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. However, real-time scheduling algorithms often try to achieve high utilization of the processor and schedule tasks in a preemptive way, leading to inevitable interferences between them, which makes the reasoning of any individual component challenging. This problem has not been fully addressed by existing work, and it is the main focus of this thesis.

1.2 Challenges in Verifying Temporal Isolation

Temporal isolation at the algorithmic level is essential for guaranteeing the security and reliability of components scheduled by 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 [13, 14], 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 the 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.

The formal reasoning of such properties is challenging due to the lack of suitable abstractions. Existing work [3] tackled memory resource by only exposing the virtual memory space to a user, while properly managing its mapping into the physical memory space so that it is never corrupted by other users. This takes advantage of the fact that a program's behavior can be entirely specified in the virtual space, leaving maximum flexibility for the OS kernel to manage its mapping. On the contrary, when it comes to temporal resources, it is not possible to describe the behavior of a task by only looking into the virtual space. This greatly complicates temporal reasoning because any abstraction for temporal behavior must include both the virtual space (e.g. a task's execution time) and the physical space (e.g. a task's deadline), making it less obvious how an OS kernel manages to achieve temporal isolation.

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 [15] brings more complexity. For example, in the QNX real-time OS [16] and the framework presented in [17], 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 Thesis

This thesis studies the 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 [1, 3].

- A novel application of supply and demand functions [18], 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.

Budget-Enforcing Scheduling The schedulability analysis of a real-time system relies on knowing each task's maximum execution time, which is usually achieved either through budget enforcement or a worst-case execution time (WCET) analysis. With budget enforcement, the OS kernel sets up the timer to trigger periodic interrupts (asynchronous to instruction execution), dividing the CPU time into time slots, which is the basic unit for specifying the budget and period for a real-time task. The scheduler then suspends a task if this task has used up its current budget, and only refills this task's budget until the next period. As a comparison, the latter approach simply uses the worst-case execution time of each task for the system's schedulability analysis.

The following difficulties make the WCET analysis approach less favorable. Firstly, a safe and precise WCET bound is hard to obtain (elaborated in more detail in Sec. 9.5) and varies across platforms, and erring on the safety side might lead to disproportionate waste of CPU time. Secondly, this approach requires explicit accounting of kernel overhead. Otherwise, a task set passing the schedulability analysis may as well experience deadline misses if the context switch and scheduling overhead take away too much time resource. Last but not least, it weakens the temporal isolation guarantee since a task has to depend on the reliability of another task's WCET analysis.

We adopt the budget enforcement mechanism in this work. It decouples the actual instruction execution from the temporal resource management, thus is able to enforce temporal isolation even if the WCET bound computation (specified as the budget) for one component is not safe. In fact, it enables a fault-tolerant component: a component which occasionally overruns its budget may register a handler for its recovery, while the OS kernel guarantees that such overruns are properly contained and do not affect other components. In this way, we achieve stronger temporal isolation than relying on the WCET bounds for tasks.

Assumptions Our work makes the following assumptions. A task does not have direct access to the global time (the RDTSC instruction is prohibited in the user mode). Components do not communicate with each other. And the OS kernel only allows user-level preemption, not kernel preemption. This may seem to be a restrictive setting, but we justify that it is still interesting as a real-time OS kernel and we leave more detailed discussions on how to relax these restrictions in Sec. 10.4 and Sec. 10.5.

Firstly, a real-time task does not need direct access to the global time for its periodic schedule. We require that a task specifies its budget and period, then it is the OS kernel's responsibility to properly schedule this task according to the global time.

Secondly, inter-component communication is prohibited as a consequence of strict isolation. However, this is still practical because there are various ways of supporting communication under the current setting. One generic and straightforward way is to encapsulate communicating tasks inside a partition (intra-partition communication is allowed), such that they are free to influence each other, while the partition prevents their influence to other partitions. In other words, this approach divides the system into subgroups according to the desired isolation policy and allows general-purpose communication within each of them. On the other hand, for special-purpose communications, e.g. when a task communicates with another I/O task to send/receive data, we can accommodate this need by implementing the I/O task as a kernel object and transform the task-level communication into system calls. Section 10.4 contains more detailed discussions.

Thirdly, our OS kernel achieves a reasonable response time even without kernel preemption. The interrupt latency depends heavily on the maximum interrupt disable time, during which interrupt is masked. This usually corresponds to the execution of system calls. Kernel preemption reduces the interrupt disable time by allowing preemption inside a time-consuming system call. Otherwise, a system call has to run to complete before the interrupt can be handled. However, our work builds on top of CertiKOS, which takes a microkernel approach and does not have time-consuming system calls in the real-time setting. In particular, `sys_spawn` is prohibited from real-time tasks and the page fault handling can be avoided by pre-allocating physical pages for real-time tasks. Other system calls only contain simple straight-line code that exhibits reasonable execution time. Sec. 10.5 discusses

how to reduce the interrupt latency with user-level preemption in more detail.

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 work 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.

Low-level non-determinism and isolation Another concern regarding the isolation between partitions is the possible low-level non-determinism in its behavior. If tasks within a partition communicate with each other while also being able to preempt each other, the overall behavior of this partition might be non-deterministic since the preemption point in different runs might differ. Assume that both a high-priority and low-priority task access and then increment a shared counter during their execution. Depending on when this low-priority task is preempted, the order in which the two tasks access this shared counter might differ, resulting in different overall low-level behavior of this partition.

Proving the isolation between partitions in the presence of such low-level non-determinism is extremely challenging because it could involve complex probabilistic reasoning [19]. Instead, this work assumes that it is the user's responsibility to specify a security property (i.e. observation function) that tolerates such low-level nondeterminism. More detailed discussions about isolation and communication are in Sec. 9.4 and Sec. 10.4.

Previous Publication This thesis incorporates and extends the work previously published as follows.

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), 31pages. [20]

Roadmap of the Thesis Chpt. 2 gives an overview of our reasoning framework for algorithmic properties and introduces the concept of virtual timelines. Chpt. 3 discusses how to use the virtual timeline to describe and reason about the properties of the fixed-priority scheduling. Chpt. 4 describes how we connect virtual timelines with the concrete implementation of a preemptive scheduler in an OS kernel and finally obtain a formal guarantee of temporal and spatial isolation. Chpt. 5 and 6 demonstrate how virtual timelines apply to partitioned scheduling and address related isolation properties. Chpt. 7 and 8 generalize the formalization of the virtual timeline to accommodate dynamic priority assignment and demonstrate its application in verifying an earliest-deadline-first scheduler. Chpt. 9 dis-

cusses the related work. Chpt. 10 discusses the limitations of our work and possible ways to extend it, then concludes.

Chapter 2

Overview of the Virtual Timeline Framework

This chapter 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 [1, 3], a verified single-core OS kernel with a cooperative round-robin scheduler. In the rest of this thesis, when we refer to CertiKOS we generally mean this specific version of mCertiKOS. CertiKOS 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. 2.1, we make the following extensions on top of the original CertiKOS (the itemized numbering below corresponds to circled numbers in the figure):

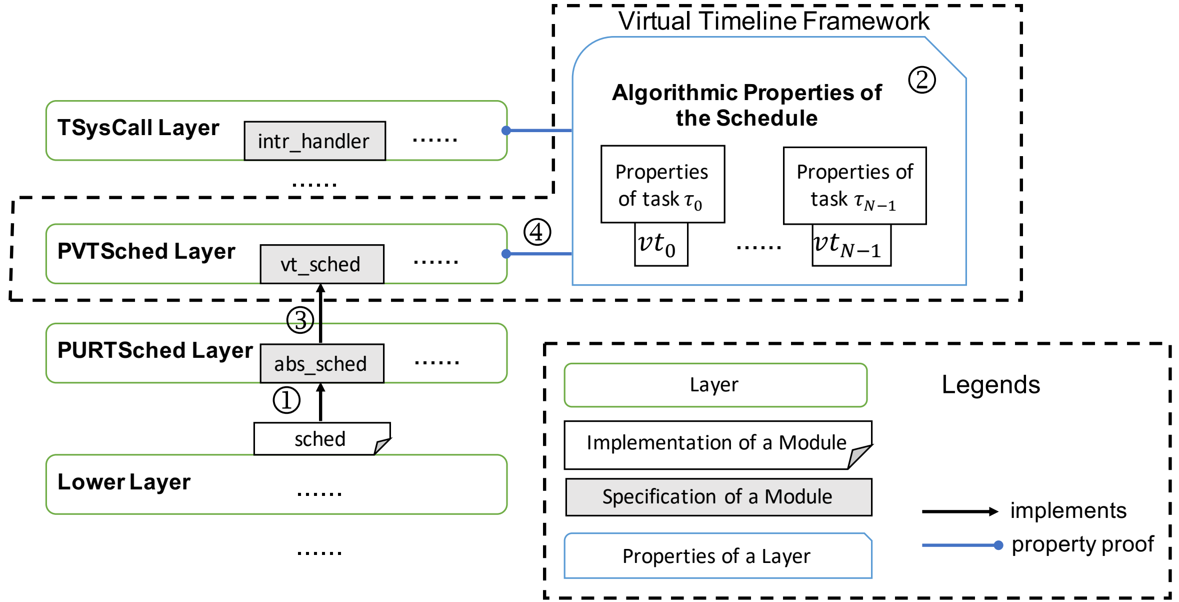


Figure 2.1: Fitting the virtual timeline framework into the verification of an OS kernel

1. 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`, faithfully implements its Coq specification, `abs_sched`.

2. We introduce a virtual timeline abstraction for each task (denoted vt_p for task τ_p). It describes the temporal behavior of a task, and is used to specify and prove the algorithmic properties of a task's schedule. See Def. 2 and Cor. 1 for more details.

3. 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`.

4. 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 5 as an example.

The intuition is that it is difficult to reason about the concrete scheduler (`abs_sched`) directly. On the one hand, the scheduler usually only maintains the current state, which is insufficient for specifying temporal properties (which may require universal quantification

over all physical periods). On the other hand, exposing low-level details makes proofs less reusable across systems with different scheduler implementations (possibly due to different levels of optimization).

The above framework tackles both issues by introducing the virtual timeline abstraction at step ②. This abstraction contains information about a task's entire schedule, thus is able to specify and reason about various temporal properties. Further, steps ③ and ④ establish the soundness of the abstraction: the schedule entailed by this abstraction is always consistent with the schedule produced by the concrete scheduler. Thus, all properties proved on the abstraction carry down to the generated assembly code of the system.

Finally, this abstraction layer based approach facilitates the reusability of this framework, enabling its adoption beyond CertiKOS. The abstraction of virtual timelines is generic in specifying and reasoning about fixed-priority scheduling, and it can be formally connected with other systems with a similar scheduling scheme through context refinement proofs (i.e. by repeating step ③). In this way, all proofs in the upper layers (② and ④) directly apply to this new system.

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. 2.2 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

```c
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;
}
```

Figure 2.2: C code of the concrete scheduler

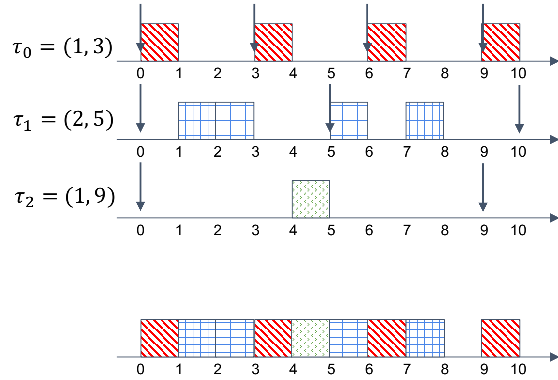


Figure 2.3: A concrete schedule

of each task, and makes sure that a task cannot be scheduled for more than its budget in any period. Fig. 2.3 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 entails the schedulability of a task: it is guaranteed in each period the full number of time slots specified by its budget.

Then, the question is how to formalize the 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 hierarchy. For example, in Fig. 2.3, the schedule of task τ_0 is regular (always occupying the first slot within each period) 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

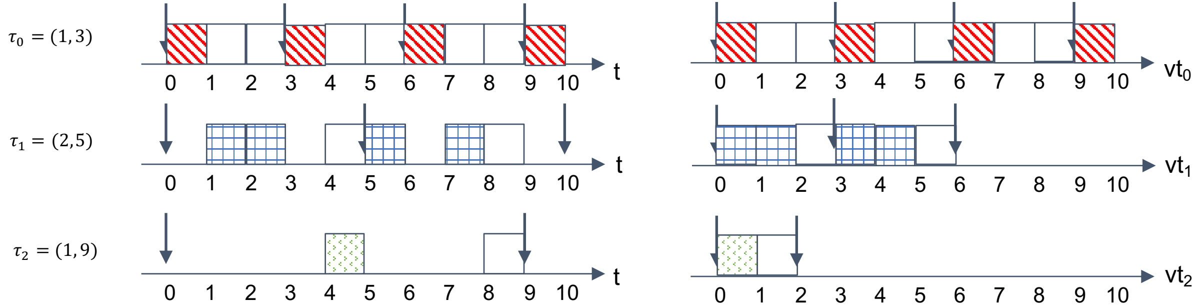


Figure 2.4: Comparing schedule on the global timeline (left) with the schedule on the virtual timeline (right)

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 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 [18, 21]. 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. 2.4, we show the schedule of tasks both on the global timeline (t) and their corresponding virtual timelines (vt_p for task τ_p). On each timeline, shaded boxes represent time slots occupied by a task, and white boxes represent time slots available to but not occupied by a task. The left side shows time slots available to a task on the global timeline. These boxes may not be consecutive due to preemptions from higher-priority tasks. The right side shows the same time slots on a task's virtual timeline. They are contiguous because the virtual timeline hides preemption from others. In particular, 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. 2.4, 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. As indicated by downward arrows in the figure, regular global periods may map to irregular virtual periods. Thus, we need to be able to convert global time into virtual time.

2.3 Formalization of the Virtual Timeline

As demonstrated in Fig. 2.4, 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.

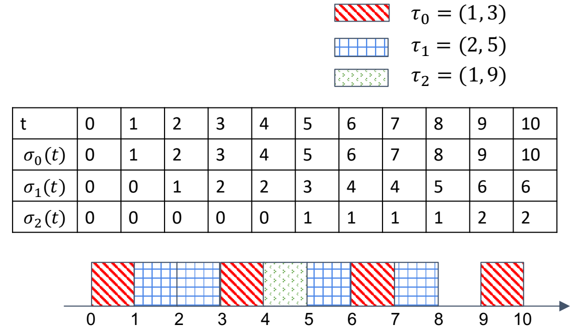
1. σ_p: `VT` $\rightarrow$ `PT`. This is a close analogy to the virtual memory mechanism, where the behavior of a process is determined by its virtual memory space, and the OS kernel maintains its page table to make sure that the corresponding physical memory space does not overlap with others. However, 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` $\rightarrow$ `VT`. This approach reflects our observation in Fig. 2.4: a task's virtual

```
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;
}
```

Figure 2.5: The vertical computation of time maps



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

Figure 2.6: A concrete time map

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` $\rightarrow$ `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. 2.5 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. 2.6 shows an example of such a dynamic computation of time maps.

Although intuitive, this way of dynamic construction of time maps is inconvenient because it complicates the reasoning about their properties. Instead, we adopt an equivalent approach of static 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. 3.2). 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 1 (Time map for periodic tasks in fixed-priority scheduling)**.**

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

Chapter 3

Reasoning about Fixed-Priority Scheduling

This chapter discusses how the virtual timeline abstraction facilitates the reasoning about algorithmic properties of the fixed-priority scheduling. In particular, we demonstrate the proof of schedulability and a task's obliviousness to other lower-priority ones.

3.1 The Schedulability Proof

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. 2.4, 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 runs continuously 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 2. *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. 2.4, 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, the seminal work by Liu and Layland [22] 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:

Theorem 1 (Virtual Critical Instant Theorem)**.**

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

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 arrive simultaneously at time 0. Thus, for any physical time window of length t, regardless of its starting point t_0, the lower bound of the amount of virtual time within this window is obtained by computing $\sigma_p(t)$. Below we explain how theorem 1 is proved in Coq, based on the time map computation given by Def. 1.

We assume that each task τ_i is schedulable in its first period, i.e. $C_i \leq \sigma_i(T_i)$. This assumption is necessary to ensure that the recursive computation in Def. 1 is valid: if $C_i > \sigma_i(T_i)$ then task τ_i is not schedulable and $\sigma_{i+1}(T_i)$ would be negative, which does not make sense.

We prove the above theorem by induction on i, and also by strengthening it with another

proof goal: at each moment, the virtual time of a task either stagnates or increments by 1. We detail its reasoning as the following.

Lemma 1 (The virtual time for a task either stagnates or increments by 1 at any moment)**.**

$$\forall \tau_i, t, \sigma_i(t+1) = \sigma_i(t) \vee \sigma_i(t+1) = \sigma_i(t) + 1$$

Proof. We prove the above lemma by induction on i. In the base case, $\sigma_0(t) = t$, the above property holds.

Assume that the above property, as well as the critical instant theorem, holds for task τ_i. We examine the virtual time map for τ_{i+1}. We consider an arbitrary time instant t and compare $\sigma_{i+1}(t+1)$ with $\sigma_{i+1}(t)$.

- If $(t+1) \mod T_i = 0$, $t+1$ marks the beginning of a new period for τ_i. In this case, $\lfloor \frac{t+1}{T_i} \rfloor = \lfloor \frac{t}{T_i} \rfloor + 1$, and also $\lfloor \frac{t+1}{T_i} \rfloor T_i = t+1$.

  Thus,

  $$\begin{aligned} \sigma_{i+1}(t+1) - \sigma_{i+1}(t) = & \quad \sigma_i(t+1) - \sigma_i(t) + \lfloor \frac{t}{T_i} \rfloor C_i - \lfloor \frac{t+1}{T_i} \rfloor C_i \\ & + min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) - min(C_i, \sigma_i(t+1) - \sigma_i(\lfloor \frac{t+1}{T_i} \rfloor T_i)) \\ = & \quad \sigma_i(t+1) - \sigma_i(t) - C_i + min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) \end{aligned}$$

  We also know that $\sigma_i(t+1) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i) \geq \sigma_i(T_i) \geq C_i$.

  - If $\sigma_i(t+1) = \sigma_i(t)$, we know

    $$\sigma_{i+1}(t+1) - \sigma_{i+1}(t) = 0 - C_i + C_i = 0$$

  - Otherwise, if $\sigma_i(t+1) = \sigma_i(t) + 1$, we know

    $$\sigma_{i+1}(t+1) - \sigma_{i+1}(t) = 1 - C_i + min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i))$$

And also $C_i - 1 \leq min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) \leq C_i$. Thus,

$$0 \leq \sigma_{i+1}(t+1) - \sigma_{i+1}(t) \leq 1$$

- If $(t+1) \mod T_i \neq 0$, t and $t+1$ belongs to the same period of τ_i. In this case, $\lfloor \frac{t+1}{T_i} \rfloor = \lfloor \frac{t}{T_i} \rfloor$.

  Thus,

$$\begin{aligned}\sigma_{i+1}(t+1) - \sigma_{i+1}(t) = \quad & \sigma_i(t+1) - \sigma_i(t) + \lfloor \frac{t}{T_i} \rfloor C_i - \lfloor \frac{t+1}{T_i} \rfloor C_i \\ & + min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) - min(C_i, \sigma_i(t+1) - \sigma_i(\lfloor \frac{t+1}{T_i} \rfloor T_i)) \\ = \quad & \sigma_i(t+1) - \sigma_i(t) + min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) \\ & - min(C_i, \sigma_i(t+1) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i))\end{aligned}$$

  - If $\sigma_i(t+1) = \sigma_i(t)$, it's easy to see that $\sigma_{i+1}(t+1) - \sigma_{i+1}(t) = 0$.
  - Otherwise, if $\sigma_i(t+1) = \sigma_i(t) + 1$, we know

$$-1 \leq min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) - min(C_i, \sigma_i(t+1) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) \leq 0$$

    Thus,

$$0 \leq \sigma_{i+1}(t+1) - \sigma_{i+1}(t) \leq 1$$

This lemma holds for τ_{i+1}. □

Below we show the proof of Thm. 1, which also goes by induction on i. In the base case, $\sigma_0(t) = t$ and the theorem hold. For the inductive step, we assume that this theorem holds for τ_i, and then inspect the virtual time available to τ_{i+1} in an arbitrary window of $[t_0, t_0 + t)$. The goal is to prove that $\sigma_{i+1}(t_0 + t) - \sigma_{i+1}(t_0) \geq \sigma_{i+1}(t)$.

Proof. We perform case analysis on how this window overlaps with periods of τ_i.

(1) If $t_0 \mod T_i = 0$, the start of the window aligns with the start of τ_i's period. In

this case, $\lfloor \frac{t_0+t}{T_i} \rfloor = \lfloor \frac{t}{T_i} \rfloor + \frac{t_0}{T_i}$. We also prove that

$$\begin{aligned}\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) = \ & \sigma_i(t_0+t) - \sigma_i(t_0) - \lfloor \frac{t}{T_i} \rfloor C_i \\ & -min(C_i, \sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i))\end{aligned}$$

Thus,

$$\begin{aligned}\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) - \sigma_{i+1}(t) = \ & \sigma_i(t_0+t) - \sigma_i(t_0) - \sigma_i(t) \\ & +min(C_i, \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)) \\ & -min(C_i, \sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i))\end{aligned}$$

- If $C_i \leq \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)$, It's easy to prove that

$$\begin{aligned}\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) - \sigma_{i+1}(t) = \ & \sigma_i(t_0+t) - \sigma_i(t_0) - \sigma_i(t) \\ & +C_i - min(C_i, \sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i)) \\ \geq \ & 0\end{aligned}$$

- Otherwise, $C_i > \sigma_i(t) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i)$, and

$$\begin{aligned}\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) - \sigma_{i+1}(t) = \ & \sigma_i(t_0+t) - \sigma_i(t_0) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i) \\ & -min(C_i, \sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i))\end{aligned}$$

  - If $\sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i) \leq C_i$, we prove

$$\begin{aligned}\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) - \sigma_{i+1}(t) = \ & \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i) - \sigma_i(t_0) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i) \\ = \ & \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i + t_0) - \sigma_i(t_0) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i) \\ \geq \ & 0\end{aligned}$$

– Otherwise, $\sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i) > C_i$, we prove

$$\begin{aligned}
\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) - \sigma_{i+1}(t) = \ & \sigma_i(t_0+t) - \sigma_i(t_0) - \sigma_i(\lfloor \frac{t}{T_i} \rfloor T_i) - C_i \\
\geq \ & \sigma_i(t_0+t) - \sigma_i(t_0 + \lfloor \frac{t}{T_i} \rfloor T_i) - C_i \\
= \ & \sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i) - C_i \\
\geq \ & 0
\end{aligned}$$

Thus, this lemma holds when the start of this window aligns with τ_i's period.

(2) If $t_0 \mod T_i \neq 0 \wedge \lfloor \frac{t_0+t}{T_i} \rfloor = \lfloor \frac{t_0}{T_i} \rfloor$, this window is small enough that it fits within one period of τ_i. In this case, we prove that

$$\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) = \ max(0, \sigma_i(t_0+t) - \sigma_i(t_0) - max(0, C_i - \sigma_i(t_0) + \sigma_i(\lfloor \frac{t_0}{T_i} \rfloor T_i)))$$

Similar to the previous case, we perform a case analysis to unfold the max/min operators and prove that

$$\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) - \sigma_{i+1}(t) \geq 0$$

Details are omitted in this section.

(3) This is the most generic case, where this window overlaps with possibly multiple periods of τ_i. We prove that

$$\begin{aligned}
\sigma_{i+1}(t_0+t) - \sigma_{i+1}(t_0) = \ & \sigma_i(t_0+t) - \sigma_i(t_0) - (\lfloor \frac{t_0+t}{T_i} \rfloor - \lceil \frac{t_0}{T_i} \rceil) C_i \\
& -max(0, \sigma_i(\lfloor \frac{t_0}{T_i} \rfloor T_i) + C_i - \sigma_i(t_0)) \\
& -min(C_i, \sigma_i(t_0+t) - \sigma_i(\lfloor \frac{t_0+t}{T_i} \rfloor T_i))
\end{aligned}$$

Similarly, we perform case analysis on whether $\lfloor \frac{t_0+t}{T_i} \rfloor = \lfloor \frac{t_0}{T_i} \rfloor + \lfloor \frac{t}{T_i} \rfloor$ or $\lfloor \frac{t_0+t}{T_i} \rfloor = \lfloor \frac{t_0}{T_i} \rfloor + \lfloor \frac{t}{T_i} \rfloor + 1$, and also on how $\sigma_i(t_0)$ and $\sigma_i(\lfloor \frac{t_0}{T_i} \rfloor T_i) + C_i$ compares to one another.

The intuition of the first case analysis is to distinguish whether $t_0 + t$ reaches one extra period. The second comparison decides whether there may still be some computation for the current period left at the start of this window. Proof details are omitted in this section.

Combining all three cases, this concludes the proof of Thm. 1. □

An immediate corollary is the schedulability of the full system:

Corollary 1. *The system is schedulable at all times if each task is schedulable for its first period.*
Said otherwise,

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

3.2 Extension with Variable Execution Time

The previous section only handles a restrictive scenario where all tasks must use up their budgets within each period. This section extends the reasoning with more flexibility, that is, the actual execution time for each task may vary. We still require that the accounting of time is in units of time slots. Under this setting, a task may relinquish its remaining time slots in any period so that lower-priority ones get a chance to run earlier.

Although the actual number of time slots a task is going to occupy during an arbitrary period can only be known at runtime, we conduct the schedulability analysis by considering all possible values. In particular, we maintain an auxiliary data structure, O, to represent an oracle for execution times. Specifically, $O_i(k)$ is the execution time at the k-th period $[kT_i,\ (k+1)T_i)$ of task τ_i. We then parameterize the computation of a time map over an oracle, written as σ_i^O.

Definition 3 (Computation of dynamic time map)**.**

$$\begin{aligned} \sigma_0^O(t) \quad &= t \\ \sigma_{i+1}^O(t) &= \sigma_i^O(t) - \sum_{j=0}^{\left\lfloor \frac{t}{T_i} \right\rfloor - 1} O_i(j) \\ &\qquad - \min\Big(O_i\big(\left\lfloor \tfrac{t}{T_i} \right\rfloor\big), \sigma_i^O(t) - \sigma_i^O\big(\left\lfloor \tfrac{t}{T_i} \right\rfloor T_i\big)\Big) \end{aligned}$$

Compared with Def. 1, this parameterized version follows the same idea of recursive computation in decreasing priority order, while allowing variations in the actual execution

time. We still require the scheduler to enforce the budget constraint, such that

$$\forall i, k, O_i(k) \leq C_i$$

For any run of the system, it instantiates a concrete value for this oracle, which accurately describes the actual execution time of each task. This section details the proof that as long as this oracle respects the budget constraint, the system will be schedulable if all tasks are able to receive their full budgets within the first period.

We prove that for any task τ_i, the amount of available virtual time within any window cannot decrease if tasks finish earlier, and also the virtual time either stagnates or increases by 1 at any moment. We achieve this by induction on i.

Since $\sigma_0^O(t) = t$, both properties hold. For the inductive step, we assume that both properties hold for τ_i, and we look at the virtual time for τ_{i+1}. The proof that

$$\forall t, \sigma_0^O(t+1) = \sigma_0^O(t) \vee \sigma_0^O(t+1) = \sigma_0^O(t) + 1$$

follows the same structure as in Lemma 1, and we omit its details in this section. Below, we focus on the proof that the virtual time available within each window does not decrease.

Lemma 2 (Finishing early does not lead to less amount of virtual time)**.** *Assume that O is a valid oracle representing the actual execution time and respecting budget constraints. For any arbitrary task τ_{i+1} and window $[t_1, t_2)$,*

$$\sigma_{i+1}^O(t_2) - \sigma_{i+1}^O(t_1) \geq \sigma_{i+1}(t_2) - \sigma_{i+1}(t_1)$$

Proof. Given that this property holds for τ_i, we conduct a case analysis on whether t_1, t_2 belong to the same period of τ_i.

- If they belong to the same period, there must exist an integer k such that $kT_i \leq t_1 < t_2 \leq (k+1)T_i$. Similar to the second case in the proof of Thm. 1, we prove that

$$\sigma_{i+1}^O(t_2) - \sigma_{i+1}^O(t_1) = \quad max(0, \sigma_i^O(t_2) - \sigma_i^O(t_1) - max(0, O_i(k) - \sigma_i^O(t_1) + \sigma_i^O(kT_i)))$$

Since the following inequalities hold

$$\sigma_i^O(t_2) - \sigma_i^O(t_1) \geq \sigma_i(t_2) - \sigma_i(t_1)$$
$$O_i(k) \leq C_i$$
$$\sigma_i^O(t_1) - \sigma_i^O(kT_i) \geq \sigma_i(t_1) - \sigma_i(kT_i)$$

We prove

$$\sigma_{i+1}^O(t_2) - \sigma_{i+1}^O(t_1) \geq \sigma_{i+1}(t_2) - \sigma_{i+1}(t_1)$$

- If they span multiple periods, we denote $\lceil \frac{t_1}{T_i} \rceil = k_1$ and $\lfloor \frac{t_2}{T_i} \rfloor = k_2$, where $k_1 \leq k_2$. We break down this window into different chunks and reason about the amount of available virtual time as follows. Notice that t_1 and $k_1 T_i$ belong to the same period, while t_2 and $k_2 T_i$ also belong to a same period. We prove that

$$\begin{aligned}
\sigma_{i+1}^O(t_2) - \sigma_{i+1}^O(t_1) = \; & \sigma_{i+1}^O(t_2) - \sigma_{i+1}^O(k_2T_i) \\
& + \sigma_{i+1}^O(k_2T_i) - \sigma_{i+1}^O(k_1T_i) \\
& + \sigma_{i+1}^O(k_1T_i) - \sigma_{i+1}^O(t_1) \\
\geq \; & \sigma_{i+1}(t_2) - \sigma_{i+1}(k_2T_i) \\
& + \sigma_i(k_2T_i) - \sigma_i(k_1T_i) - \sum_{j=k_1}^{k_2} O_i(j) \\
& + \sigma_{i+1}(k_1T_i) - \sigma_{i+1}(t_1) \\
\geq \; & \sigma_{i+1}(t_2) - \sigma_{i+1}(k_2T_i) \\
& + \sigma_i(k_2T_i) - \sigma_i(k_1T_i) - (k_2 - k_1)C_i \\
& + \sigma_{i+1}(k_1T_i) - \sigma_{i+1}(t_1) \\
= \; & \sigma_{i+1}(t_2) - \sigma_{i+1}(t_1)
\end{aligned}$$

Thus, this lemma holds. □

As an immediate corollary, we obtain the schedulability proof with variable execution time.

Corollary 2. *The system is schedulable at all times if each task is schedulable for its first*

period. For any valid oracle O representing actual execution time,

$$\big(\forall i, C_i \leq \sigma_i(T_i)\big) \implies \forall i, k \geq 0, C_i \leq \sigma_i^O((k+1) * T_i) - \sigma_i^O(k * T_i)$$

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

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

This is straightforward by observing in Def. 3 that the computation only relies on the execution time of higher priority tasks. We omit proof details in this section.

The above lemma illustrates one of the motivations for the virtual timeline, that is, it facilitates the reasoning about isolation properties. Since σ_i serves as a comprehensive description of τ_i's temporal behavior, any interference to τ_i must also translate to the interference on σ_i.

More importantly, comparing Lemma 3 with Corollary 2, we observe that the virtual timeline abstraction also enables a flexible definition of *isolation*. In Lemma 3, we say τ_i is affected by others if σ_i changes. In this case, isolation means the exact value of σ_i is not influenced by lower-priority tasks. However, Corollary 2 tolerates variations in a task's exact schedule as long as its schedulability still holds. In other words, the validity of a specific predicate, $\forall k, \sigma_i((k+1)T_i) - \sigma_i(kT_i) < C_i$, must hold regardless of the exact value of σ_i.

Chapter 4

Case Study: CertiKOS with Fixed-Priority Scheduling

Chapter 3 demonstrates the high-level reasoning about a component's schedule on its virtual timeline, which corresponds to step ② in Fig. 2.1. In this chapter, 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. 2.2) 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. 2.4. 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 ③ in Fig. 2.1). 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 ④ in Fig. 2.1). Combining both steps together, we connect individual virtual timelines with the concrete scheduler implementation of an OS kernel.

In this chapter, 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 when a task always uses up its budget (Sec. 4.3), and also an extension that allows a task to finish earlier (Sec. 4.4). Last but not least, we demonstrate how to achieve complete isolation proof between tasks (Sec. 4.5).

4.1 Background: CertiKOS, a Verified OS Kernel

CertiKOS [1, 3] is a verified kernel whose functional correctness has been mechanized in the Coq proof assistant.

Abstraction layers. One of the highlighting features of CertiKOS is that its functional correctness guarantees hold on the assembly level. This is achieved by dividing the kernel into many small pieces, called *abstraction layers*. Each layer defines its own abstract state and specifications of primitives operating on this state and proves that the underlying implementation is equivalent to its abstraction. In this way, it exposes specifications instead of implementations to higher layers, making it possible to scale the reasoning of a complex software system.

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 [23, 24] to propagate the verification done at the C source code level down to the generated assembly code. (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) [25].

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. 2.2) and prove its functional correctness following the layered approach (step ① in Fig. 2.1). The timer interrupt is set up to occur periodically, and this 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 a 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. Both functions are abstract in the definition of LAsm semantics and are further instantiated in each layer with a concrete effect on the corresponding abstract state.

We only model the timer interrupt in this way, since polling for other device interrupts is the typical behavior for high-assurance systems [4, 5]. 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.

The downside of this choice is that disabling interrupts in the kernel may hurt the responsiveness of the system. However, CertiKOS keeps the kernel small and avoids introducing tedious operations into the kernel mode in general. Further, 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. On the other hand, operations such as page fault handling can be avoided by pre-allocating required physical memory for real-time applications, which may not rely on dynamic memory allocation. In this way, the interrupt disable time could be reduced.

Coq formalization of the scheduler As shown in Fig. 2.1, we need to first define a Coq formalization of the scheduler, `abs_sched`, and prove that it is equivalent to the C

```
1  Inductive prio_config: Type :=
2    | mkPrioConfigValid: Z (* period *) → Z (* budget *) → prio_config.
3
4  (* A mapping from integer to the task configuration *)
5  Definition PrioConfigPool: Type := ZMap.t prio_config.
6
7  (* A mapping from interger to integer *)
8  Definition QuantumPool: Type := ZMap.t Z.
9
10 (* Fields in RData *)
11 uticks: Z  (* timer interrupt counter *)
12 quanta: QuantumPool  (* remaining budget for each task *)
13 config: PrioConfigPool   (* parameters for all tasks *)
```

Figure 4.1: Coq formalization of necessary data structures

implementation.

The C implementation, `sched`, is shown in Fig. 2.2. It relies on a `quanta` array to maintain the remaining budget for each task, it also needs to query the period and budget for each task. Finally, it maintains a global variable, `t`, to represent the current time (in units of time slots). We list the Coq formalization of the above data structures in Fig. 4.1. Here, `Z` is the Coq type for binary integers, and `ZMap.t` is a finite map from integer to a specified value type. `RData` defines the abstract state in CertiKOS, i.e. all primitives are specified as a transition function on `RData` [1].

We extend the original definition of `RData` with the following new fields to accommodate real-time scheduling.

- `uticks`: the current time of the system.
- `quanta`: the array maintaining the remaining budget for each task. In particular, this finite map ranges from 0 to N-1, where N represents the total number of tasks.
- `config:` parameters of each task. In particular, it records the period and budget for each task. This field is not to be updated by the scheduler.

On top of these, we define the Coq formalization of the scheduler as shown in Fig. 4.2 and Fig. 4.3. Here, the abstract function `tick_quanta` corresponds to lines 3 - 7 in Fig. 2.2,

```
1  Fixpoint tick_quanta (t: Z) (pri: nat) (conf: PrioConfigPool) (quanta: QuantumPool) :=
2    match pri with
3    | O ⇒ quanta
4    | S p ⇒ match (ZMap.get (Z.of_nat p) conf) with
5            | mkPrioConfigValid T C ⇒
6              if (zeq (t mod T) 0) then
7                ZMap.set (Z.of_nat p) C (tick_quanta t p conf quanta)
8              else
9                tick_quantums t p conf quanta
10             end
11   end.
12
13 Fixpoint highest_pos_quanta (pri: nat) (quanta: QuantumPool) :=
14   match pri with
15   | O ⇒ N
16   | S p ⇒ let prio := N - Z.of_nat p - 1 in
17           if (zlt 0 (ZMap.get prio quanta)) then
18             prio
19           else
20             highest_pos_quanta p quanta
21   end.
```

Figure 4.2: Coq formalization of the abstract scheduler

where the scheduler iterates over each task and refills the budget if a new period arrives. Function `highest_pos_quanta` corresponds to lines 10 - 16 in Fig. 2.2, where the scheduler iterates over all tasks in decreasing order of priority and returns the first task that has a remaining budget. Finally, function `absched` only decrements the quanta value if a real-time task is scheduled.

The proof that the abstract function `absched` indeed captures all behaviors of its C implementation follows the same approach established by Gu et al. [1]. We divide the scheduler implementation into multiple layers and build up the contextual refinement proof from these smaller components. We omit proof details in this section.

```
1  Function absched (adt: RData) : Z * RData :=
2    let ticks' := (uticks adt) + 1 in
3    let quanta' := tick_quanta ticks' (N - 1) config (quanta adt) in
4    let prio := highest_pos_quanta N quanta in
5    if (zlt prio N) then
6    (prio, adt {uticks: ticks'}
7                {quanta: ZMap.set prio ((ZMap.get prio quanta') - 1) quanta'})
8    else
9      (prio, adt {uticks: ticks'} {quanta: quanta'}).
```

Figure 4.3: Coq formalization of the abstract scheduler

4.3 Connecting The Schedulability Proof with The Concrete Scheduler

In this section, we explain how we connect virtual timelines with the concrete scheduler implementation (Fig. 2.2). We first introduce the virtual-time-based scheduler `vt_sched`. Fig. 4.4 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_i, t), which checks whether task τ_i is scheduled within global time duration [t, t+1). Fig. 4.5 gives a concrete example of a task's schedule in both the global and virtual timeline. In particular,

- `VTA` (σ_1, 6) $= 0$ (τ_1 is not scheduled at time 6), because the next time slot is not available to τ_1 ($\sigma_1(6) = \sigma_1(7)$).
- `VTA` $(\sigma_1, 7) = 1$, since the higher priority task is finished, and task τ_1 still has remaining budget ($\sigma_1(7) < \sigma_1(8) \wedge \textit{consumption} = 1 < C_1$).
- `VTA` $(\sigma_1, 8) = 0$, because its budget is exhausted at this point ($\textit{consumption} = 2 \geq C_1$).

Connecting `abs_sched` with `vt_sched` This corresponds to step ③ in Fig. 2.1. The concrete scheduler as shown in Fig. 2.2 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. 4.4) based on the following intuitions.

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


```

```c
int vt_sched(){
  t++;

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

  return pid;
}
```

Figure 4.4: C Illustration of the virtual-time-based scheduler (actually defined in Coq)

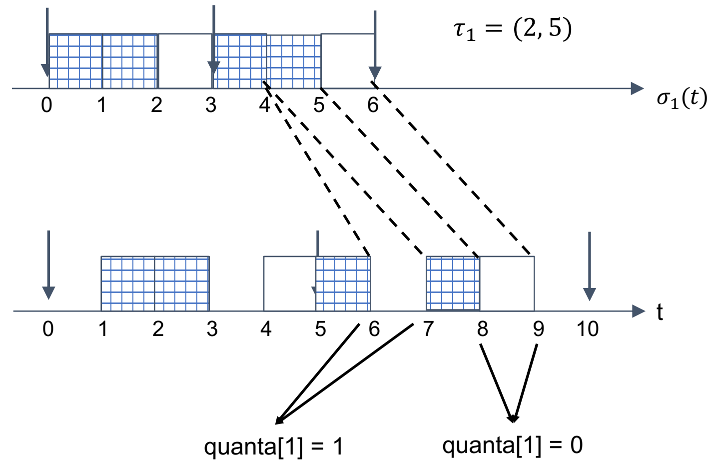


Figure 4.5: An example schedule in global and virtual time

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

- All higher priority tasks are finished. That is, $\forall j < i$, `quanta` $[j] = 0$.
- There is no new arrival of higher priority tasks. That is, $\forall j < i, t \mod T_j \neq 0$. This corresponds to the first loop in Fig. 2.2. When a new period starts, the quantum value will be reset to this task's budget.
- Task τ_i still has remaining budget, or a new period is arriving. That is, `quanta`$[i] > 0 \vee t \mod T_i = 0$.

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

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

- `VTA` $(\sigma_i, \text{t}) = 1$.

The above necessary and sufficient conditions indicate 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. 4.5 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 (Equivalence between states of the abstract scheduler and the virtual-time-based scheduler)**.** *Assume that the current quanta array is Q and the current time is t.*

$$\texttt{quanta_valid}(Q,t) \equiv \forall i, Q[i] = C_i - \min\left(C_i, \sigma_i(t) - \sigma_i(\left\lfloor \frac{t}{T_i} \right\rfloor T_i)\right)$$

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.

Lemma 4 (The equivalence relation between the quanta array and the virtual time is preserved by the scheduler)**.** *At an arbitrary moment t, assume that the current quanta array is Q. The scheduler first checks new task arrivals and updates the quanta array to Q'. It then schedules a task and further updates the quanta array to Q''. The following must hold.*

$$\texttt{quanta_valid}(Q,t) \implies \texttt{quanta_valid}(Q'',t+1)$$

Proof. We do a case analysis on whether the scheduled task is a real-time task or not.

1. Assume that a real-time task τ_k is scheduled at time t.

    - For task τ_k itself,

        - If $t \mod T_k = 0$, this marks the beginning of a new period and $Q'[k] = C_k$ must hold. Since it is scheduled, $Q''[k] = C_k - 1$. On the other hand,

$C_k - \min(C_k, \sigma_k(t+1) - \sigma_k(\left\lfloor \frac{t+1}{T_k} \right\rfloor T_k)) = C_k - min(C_k, 1) = C_k - 1$. Thus, the equivalence holds.

  – Otherwise, t and $t+1$ belong to the same period of τ_k. Again, it's easy to see that $Q''[k] = Q'[k] - 1 = Q[k] - 1$. Since τ_k is not finished by time t, we know $\sigma_k(t) - \sigma_k(\left\lfloor \frac{t}{T_k} \right\rfloor T_k) < C_k$. We prove that

$$\sigma_k(t+1) - \sigma_k(\left\lfloor \frac{t+1}{T_k} \right\rfloor T_k) = \sigma_k(t) + 1 - \sigma_k(\left\lfloor \frac{t}{T_k} \right\rfloor T_k) \leq C_k$$

  Thus, the equivalence holds.

- For a higher-priority task τ_j where $j < k$, its virtual time increases while its quantum value remains 0. We also prove that $t \mod T_j \neq 0$, otherwise, τ_j would be able to preempt τ_k. Since $\sigma_j(t+1) - \sigma_j(\left\lfloor \frac{t+1}{T_j} \right\rfloor T_j) \geq C_j$ at this moment, the equivalence holds.

- For a lower-priority task τ_j where $j > k$,

  – If $t \mod T_j = 0$, this marks the beginning of a new period such that $Q''[j] = C_j$. However, its virtual time does not increase, thus the equivalence holds.

  – Otherwise, neither its virtual time nor its quantum value changes. Since t and $t+1$ belong to the same period of τ_j, the equivalence holds.

2. If a batch task is scheduled, this means $t \mod T_i \neq 0$ holds for any task τ_i. We also prove that $\sigma_i(t+1) - \sigma_i(\left\lfloor \frac{t+1}{T_i} \right\rfloor T_i) \geq C_i$ since all tasks are finished. This entails the equivalence relation at time $t+1$.

□

Finally, we prove that `abs_sched` and `vt_sched` always produce the same schedule. Proof details are omitted in this section.

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 ④ in Fig. 2.1 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. 2.4, 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_i that satisfies VTA (σ_i, t).

Lemma 5 (The virtual timeline faithfully describes the schedule of a task)**.** *For each arbitrary task τ_i and time t,*

$$\texttt{vt_sched}(t) = i \Leftrightarrow \texttt{VTA}(\sigma_i, t) = 1$$

Proof. The forward direction is trivial: if τ_i is scheduled it must satisfy $\texttt{VTA}(\sigma_i, t) = 1$. We focus on the other direction and prove that if VTA (σ_i, t) = 1 for task τ_i, then VTA (σ_j, t) = 0 must hold for any other higher-priority task τ_j.

We prove this by contradiction. Assume that there is a higher priority task τ_j such that VTA (σ_j, t) = 1. Since the time slot [t, t+1) is occupied, it becomes unavailable to task τ_{j+1} and all subsequent lower-priority tasks. Thus $\sigma_i(t+1) = \sigma_i(t)$, which contradicts the assumption that VTA (σ_i, t) = 1. □

In this way, we formally establish the relation between a task's schedule and its virtual timeline, and we only need to examine the predicate $\texttt{VTA}(\sigma_i, t)$ to decide task τ_i's schedule at any time t.

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 1. And the schedulability property carries down to the generated assembly code of the system since we connect the virtual timeline abstraction to the implementation of the scheduler.

4.4 The sys_finish System Call

This section discusses 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. This requires a system call `sys_finish`, similar to `periodic_wait` in the

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

Figure 4.6: The `sys_finish` system call

```
void vt_sys_finish(){
  int i = cur_pid;
  int start = t / T_i * T_i;
  int exec_time = σ_i(t) − σ_i(start);
  O[i][t / T_i] = exec_time;
}
```

Figure 4.7: C illustration of the virtual-time-based `sys_finish`

ARINC 653 standard [26]. The core of its C implementation is straightforward, as shown in Fig. 4.6.

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. While the computation of time maps as well as various properties (e.g. schedulability and obliviousness to lower-priority tasks) are addressed in Sec. 3.2, this section addresses the connection between the virtual timeline abstraction with the concrete scheduler implementation under this more flexible setting.

Connecting the concrete scheduler with the virtual-time-based scheduler Under this setting, the concrete scheduler is the same as in Sec. 2.2, and the virtual-time-based scheduler is also similar, except with a parameterized version of time maps. In particular, the oracle O is now part of the abstract state, which the virtual-time-based scheduler relies on to compute the current time maps for each task. We update the equivalence relation between states (Def. 4) as shown below.

Definition 5 (Equivalence between states of the abstract scheduler and the virtual-time-based scheduler, under the setting of variable execution time)**.**

$$\texttt{quanta_valid}(Q,O,t) \equiv \forall i, Q[i] = O_i(\left\lfloor \frac{t}{T_i} \right\rfloor) - \min\left(O_i(\left\lfloor \frac{t}{T_i} \right\rfloor), \sigma_i^O(t) - \sigma_i^O(\left\lfloor \frac{t}{T_i} \right\rfloor T_i)\right)$$

We prove that this simulation relation is preserved by the scheduler, i.e. if the concrete scheduler causes the transition from Q to Q' at time t, then

$$\texttt{quanta_valid}(Q,O,t) \implies \texttt{quanta_valid}(Q',O,t+1)$$

Notice that the scheduler does not update time maps, thus the oracle before and after this invocation remains the same. Although seemingly different, its proof is achieved in the same way as in Lemma 4, by conducting case analysis on the relative priority between an arbitrary task with the scheduled task. On top of this equivalence relation, we further prove the refinement between the concrete scheduler and the virtual-time-based scheduler. Both proof details are omitted in this section.

Connecting `sys_finish` with its virtual-time-based abstraction The remaining proof obligation is to connect the virtual-time-based system call, `vt_sys_finish` (illustrated in Fig. 4.7 using C code), with `sys_finish`. Here, `exec_time` represents the actual execution time of task τ_p. Its computation exploits the fact that all lower priority tasks are not runnable until task τ_i is finished, i.e., the entire virtual time duration is consumed by τ_i. We prove the equivalence of the two by showing that they also preserve the relation in Def. 5.

Lemma 6 (The implementation and virtual-time-based abstraction of sys_finish preserve the equivalence relation)**.** *Assume that at time t, the concrete `sys_finish` causes the transition from Q to Q', while the virtual-time-based `sys_finish` causes the transition from O to O'.*

$$\texttt{quanta_valid}(Q, O, t) \implies \texttt{quanta_valid}(Q', O', t)$$

Proof. Assume that the calling task is τ_i. We prove that the only change regarding Q is that $Q'[i] = 0$. And the only update regarding O is that $O'_i(t/T_i) = \sigma_i^O(t+1) - \sigma_i^O(\lfloor \frac{t}{T_i} \rfloor T_i)$.

1. For τ_i itself, we prove that $\sigma_i^{O'} = \sigma_i^O$ and also $Q'[i] = 0$. Given the definition of O', the equivalence relation holds.

2. For an arbitrary higher-priority task τ_j where $j < i$, neither its quantum value nor time map changes. Thus, we prove that the equivalence relation also holds for τ_j.

3. For an arbitrary lower-priority task τ_j where $j > i$, its quantum value and the oracle value remains the same, while the time map is updated. However, we prove that this update to the time map only affects future time instants. In other words, since the

schedule up to $t+1$ does not change,

$$\forall x \leq t+1, \sigma_j^{O'}(x) = \sigma_j^{O}(x)$$

This property is proved by induction on j, whose details are omitted in this section. On top of it, we prove that the equivalence relation also holds.

This concludes the proof that Def. 5 is preserved by invoking `sys_finish`. □

On top of the above equivalence proof, we prove that the concrete and virtual-time-based `sys_finish` are equivalent. In this way, any system call that may affect the quanta array is formally connected with its virtual-time-based abstraction, such that the virtual timeline abstraction also applies to the extension with variable execution time.

4.5 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.

Scope of 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.

More specifically, a user task can execute user-mode instructions or invoke system calls to interact with the OS kernel. We prohibit instructions such as RDTSC which may leak information about the global time, and we also carefully craft system calls so that they are either independent of the passage of time or only expose time consumption by the calling task itself. In this way, a task is only sensitive to its schedulability, which holds as long as the schedulability criterion is satisfied.

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.

The above notion of temporal isolation differs from existing work [4, 5] in that it allows flexibility in the global schedule of a component, which leads to better responsiveness of tasks and higher utilization of the processor. As a comparison, existing work enforces a more 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 in our setting. 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 OS kernel with a fixed-priority scheduler.

Background: non-interference of cooperative CertiKOS The existing cooperative version of CertiKOS enjoys a mechanized proof of non-interference by Costanzo et al. [3]. Unlike the usual approach based on security labels [27], 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. Two states are *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 6 (Non-Interference)**.** *A system is said to be* non-interferent *if its execution relation step preserves indistinguishability:*

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

This definition enables a flexible way of specifying security policies. For instance, the case study in [3] specifies that a task's virtual memory space should be independent of other tasks while the exact physical pages backing up this virtual space might vary. The non-interference proof is carried down to the assembly code, thanks to security-preserving simulation relations [3].

The non-interference proof of the cooperative CertiKOS takes advantage of the fact that the only way to trigger a context switch is through the `sys_yield` system call, in particular, context switches always happen at the same places. Thus, this framework cannot handle nondeterministic occurrences of preemption (triggered by timer interrupts).

Background: structure of the non-interference proof of cooperative CertiKOS

- The behavior of a task. As explained above, a task's whole-execution behavior is defined on top of its observation function in [3].
  - 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 [3] 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 [3]. This proof relies on the frame rule mentioned above.

Extension: non-interference with preemption Preemption breaks this existing proof as context switches can be triggered by interrupts in a nondeterministic way, such that we can no longer match two executions step-by-step. 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. This is a byproduct of the non-interference proof: it requires us to examine every kernel service and make sure that its specification is tight enough to entail isolation. We 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 [3] (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 (as shown in Fig. 4.8, transi-

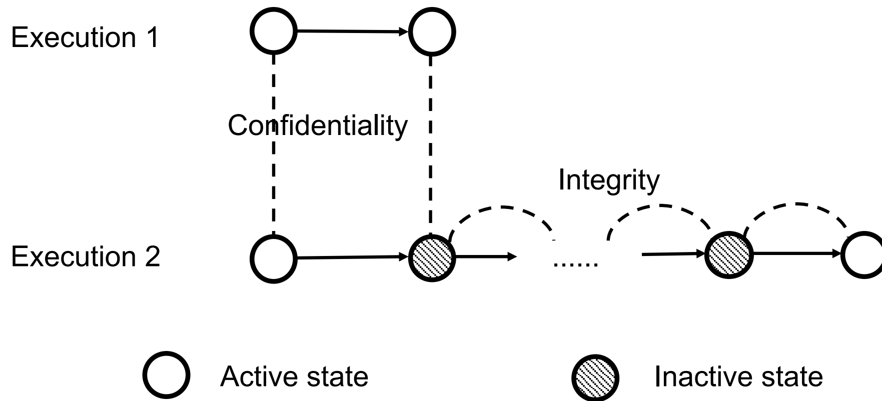


Figure 4.8: The noninterference framework with preemption

tions from inactive states preserve a task's observation). 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 (as shown in Fig. 4.8, transitions from observably equivalent active states result in equivalent states). Unlike in cooperative CertiKOS, confidentiality here does not require the two 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. In this way, this new framework allows non-deterministic preemption while still guaranteeing information-flow security between tasks.

This concludes the proof of spatial isolation. Combining it with the schedulability proof, we obtain a fully-verified OS kernel with both temporal and spatial isolation between tasks.

Chapter 5

Reasoning about Partitioned Scheduling

5.1 Background: 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. 5.1 shows an example of partitioned scheduling. Here, we focus on partition Π_0, which consists of two tasks τ_0 and τ_1, while τ_0 has higher priority over τ_1. The upper half of the figure shows the schedule when Π_0 is the only partition in the system. Under this setting, this partition always occupies the first two time slots within every period, and its tasks are only scheduled when the partition is active. As a comparison, the lower half depicts the case when Π_0 is interleaved with Π_1.

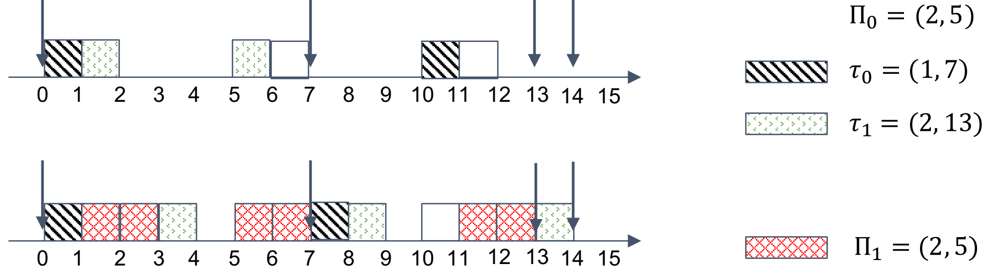


Figure 5.1: Partitioned scheduling

Though the partition's exact schedule varies, it is still guaranteed two time slots in every period.

Also notice that white boxes in Fig. 5.1 represent idle tasks in partition Π_0. We require that a partition's schedule within every period must be complete, that is, it is scheduled even if there is no active real-time task in this partition. We explain in Sec. 5.2 that this is necessary for ensuring temporal isolation between partitions.

Partitioned scheduling is a mechanism for confining groups of tasks, such that each group enjoys its own share of the CPU resource. There are various ways of scheduling partitions, among them are two-level scheduling and global scheduling.

Two-level scheduling Also known as hierarchical scheduling, this scheme first schedules a partition, then schedules a task within this partition, as shown in Fig. 5.2. Notice that M represents the number of partitions in the system, while `parQuanta` maintains the remaining budget for each partition, similar to how `quanta` maintains the budget for each task. Also, notice that the local scheduler for partition Π_i is independent of the partition scheduling algorithm. In other words, it has the flexibility to adopt either fixed-priority scheduling, earliest-deadline-first scheduling, or even round-robin scheduling.

In this way, two-level scheduling can be viewed as a mechanism for allocating CPU resources among partitions: the local scheduler within each partition is independent of one another.

Global scheduling Despite all the flexibility provided by two-level scheduling, it is restrictive in the sense that it prohibits task preemption across partitions. For instance, it

```
int two_level_sched(){
  t++;
  // refill partition budgets
  for(int i = 0; i < M; i++){
    if (t % P_i == 0){
      parQuanta[i] = B_i;
    }
  }

  // schedule partition Π_i
  ......
  parQuanta[i]--;

  // invoke the local scheduler
     for Π_i
  return local_sched(i);
}
```

Figure 5.2: The two-level scheduling

```
int global_sched(){
  t++;

  // refill partition budgets
  ......

  // refill budgets for all tasks
  for(int i = 0; i < N; i++){
    if (t % T_i == 0){
      quanta[i] = C_i;
    }
  }

  // schedule a task τ_i
  ......
  quanta[i]--;

  int par = get_par(i);
  parQuanta[par]--;

  return i;
}
```

Figure 5.3: Example for global scheduling

is possible for a critical task to be delayed by another non-urgent task simply because the critical one's partition does not get the chance to run.

Global scheduling is a scheme that achieves a good balance between isolation and responsiveness. Fig. 5.3 shows one way of scheduling tasks from all partitions globally, while also requiring that each task being subject to its partition budget. On the one hand, a high-priority task is able to preempt other lower-priority ones even if they belong to a different partition. On the other hand, each partition is still guaranteed to receive its full budget within each period.

The major difference between the two schemes is that the two-level scheduling imposes an inherent priority among partitions, while the latter relies on the global task priority and is thus more dynamic.

5.2 Temporal Isolation in Partitioned Scheduling

Notice that Fig. 5.1 demonstrates an interesting phenomenon: from partition Π_0's local view, its tasks' schedule varies depending on the exact schedule of this partition. For

example, when it is the only partition in the system (the upper half of Fig. 5.1), its tasks interleave with each other as follows: $\tau_0, \tau_1, \tau_1, idle, \tau_0, idle$. As a comparison, when Π_0 is scheduled alongside Π_1, the interleaving of its tasks is shown below: $\tau_0, \tau_1, \tau_0, \tau_1, idle, \tau_1$.

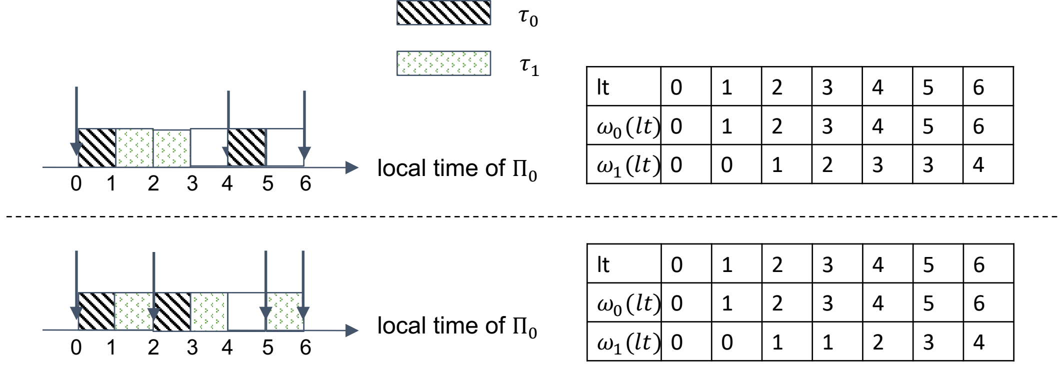
This is caused by the varying task arrival time in this partition's local view, which clearly leads to interferences between partitions and invalidates the isolation requirement between partitions. On the one hand, if tasks need to cooperate with each other in the same partition, the reordering of their schedule might jeopardize its correct operation. On the other hand, this leads to information flow between partitions, weakening the security guarantee each partition enjoys.

Existing work on isolation properties in partitioned scheduling [4] assumes a fixed schedule for partitions (by using a round-robin scheduler to schedule a fixed set of partitions), which is a restrictive setting as discussed above. They prove that there is no information flow through the scheduler since the scheduling of partitions is statically determined. However, their verification framework does not extend to the issue presented in Fig. 5.1. For instance, if the number of partitions differs in two different runs of the system, the aforementioned isolation property vanishes. The fundamental limitation is that existing work does not include temporal behavior in their semantics, thus unable to specify nontrivial temporal properties such as schedulability and isolation. In this work, we address this issue using the notion of virtual timelines.

The local time map for a task This section describes how to adapt the concept of the virtual timeline to describe a task's behavior from its enclosing partition's point of view. 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.

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

Under this notation, π_i describes the schedule of partition Π_i. In particular, $\pi_i(t)$ represents the partition local time at global time t, meaning the accumulative amount of



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

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	3	4

Figure 5.4: Another example of local time map for a task

time slots allocated to this partition. Assume that the OS kernel indeed guarantees its full budget within each period, the constraint below always holds.

$$\forall k \geq 0, \pi_i(kP_i) = kB_i$$

The schedule of task τ_j in its partition's local view is thus denoted by ω_j. Similar to the discussion in Sec. 2.3, $\omega_j(t)$ represents the amount of available virtual time for τ_j, including both time available to itself and available to lower-priority tasks. In this sense, ω_j and σ_j are computed in the same way as Fig. 2.5 (both of them increment at the same time), with the only exception that the local time map is indexed by the local time of Π_i. This makes ω_j a suitable description for a task's local schedule: it ignores time slots not available to the enclosing partition.

For example, Fig. 5.4 depicts the local time maps for τ_0 and τ_1 corresponding to the schedule in Fig. 5.1. Here, ω_0 includes all time slots available to this partition since τ_0 has the highest priority in this partition. Further, ω_1 excludes time slots occupied by τ_0, similar to the way σ_1 is constructed by removing the schedule of τ_0 as shown in Fig. 2.6.

We further observe that the local time map for τ_1 changes once partition Π_0 is scheduled together with Π_1. This demonstrates that the local time map indeed captures a task's local behavior faithfully. The rest of this section investigates how to specify and reason about temporal isolation between partitions using this new notion of local time map.

Temporal isolation between partitions 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 local time map to describe a task's schedule within a partition. In addition, since this isolation property involves comparing two different executions, we define a *canonical* local time map for the same task, rewritten $\omega_j^{\Pi_i}$.

Definition 7 (The canonical local time map for task $\tau_j \in \Pi_i$)**.** *We define $\omega_j^{\Pi_i}$ as the local time map for task τ_j, 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.

This is called the canonical schedule because it corresponds to the situation when there is only one partition in the system. Again, this relies on the assumption that the partition-level scheduling is complete, that is, a partition must receive its full budget within each period even if there is no available task to schedule. In other words, it cannot save its budget for future periods. We discuss why this is necessary at the end of this section.

Given the canonical local schedule, a partition is said to enjoy temporal isolation if its local schedule of tasks is always consistent with the canonical schedule, regardless of other partitions in the system.

Definition 8 (The local schedule of partition Π_i is oblivious to other partitions)**.**

$$\forall p, \tau_p \in \Pi_i \implies \omega_p^{\Pi_0 \ldots \Pi_M} = \omega_p^{\Pi_i}$$

Here, $\omega_p^{\Pi_0 \ldots \Pi_M}$ denotes the local schedule of τ_p if there are M partitions in the system. The above definition states that the local schedule of tasks within a partition is not influenced by other partitions in the system. Since by definition, in each run of the system, $\sigma_p^{\Pi_0 \ldots \Pi_M} = \omega_p^{\Pi_0 \ldots \Pi_M} \circ \pi_i^{\Pi_0 \ldots \Pi_M}$, we obtain an equivalent formalization of this isolation property.

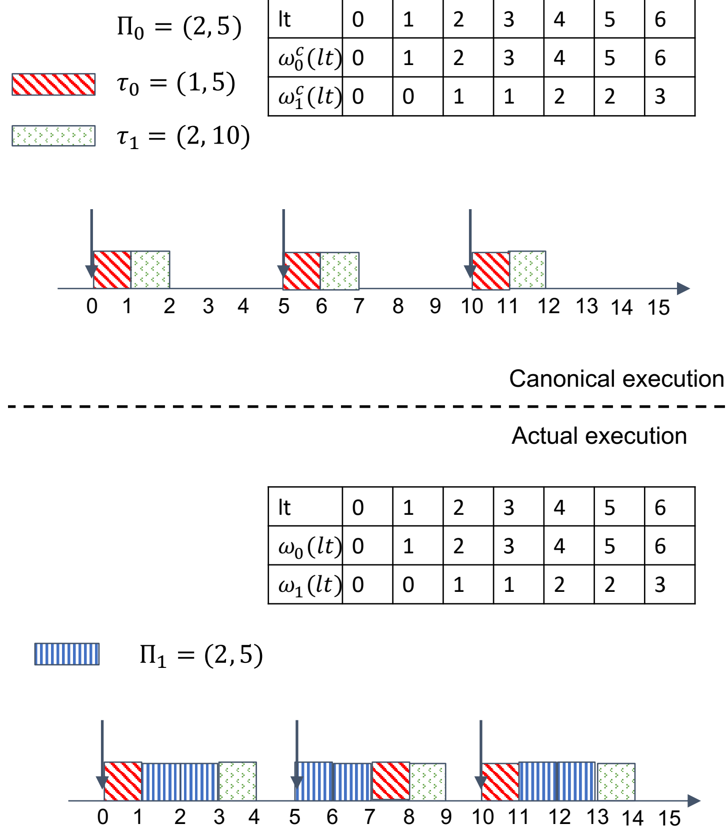
Definition 9 (The local schedule of partition Π_i is oblivious to other partitions: alternative formalization)**.**

$$\forall p, \tau_p \in \Pi_i \implies \sigma_p^{\Pi_0 \ldots \Pi_M} = \omega_p^{\Pi_i} \circ \pi_i^{\Pi_0 \ldots \Pi_M}$$

This statement is equivalent to Def. 8: that the local schedule within a partition is independent of other partitions. However, it also entails an analogy with the kind of spatial isolation obtained through virtual memory mechanism. We can view $\omega_p^{\Pi_i}$, $\pi_i^{\Pi_0 \ldots \Pi_M}$ and $\sigma_p^{\Pi_0 \ldots \Pi_M}$ as the virtual memory space, the page table, and the physical memory space, respectively. Here, $\omega_p^{\Pi_i}$ (analogous to the virtual memory space) is decided solely on the partition itself, similar to the case that an independent task's virtual memory space only depends on its own execution. When multiple components reside on the same platform, the exact allocation of time slots (denoted by $\pi_i^{\Pi_0 \ldots \Pi_M}$ for the partition and $\sigma_p^{\Pi_0 \ldots \Pi_M}$ for the task) may vary depending on the behavior of other partitions, yet the interleaving of tasks inside a partition does not change. This is analogous to the case that a task's page table and physical memory allocation are influenced by others, but not its virtual memory space.

We observe that this obliviousness holds if all tasks have a period that is a multiple of their partition's period, which is a common practice of assigning the partition 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 [15]), thus their arrival time in the partition's local view does not change. Fig. 5.5 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. In this way, both the local time map for τ_0 and τ_1 does not change regardless of other partitions in the system.

Further, under this constrained setting, there is a straightforward way of computing $\omega_p^{\Pi_i}$. Observe in Fig. 5.5 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.



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

Figure 5.5: Another example of local time map for a task

Definition 10 (Local time map for periodic tasks with fixed-priority scheduling)**.**

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

The intuition is that, when enclosed inside a partition, these tasks need to execute for the same amount of time slots while only being given a smaller portion of the CPU resource. This is equivalent to shortening their periods while keeping the same budgets since time slots occupied by other partitions are not visible to these tasks.

An argument against work-conserving scheduling Now we briefly discuss why the partition must receive its full budget within each period, even if there is no available task to schedule. This is similar to periodic servers, where the server budget cannot be accumulated

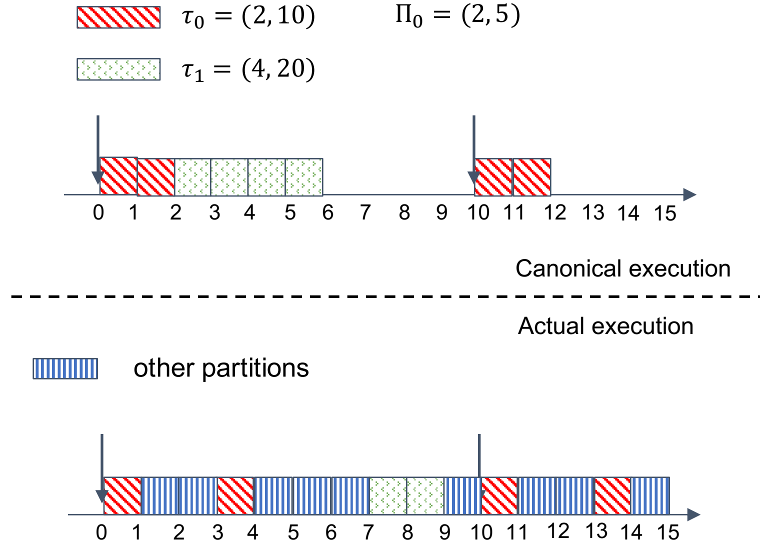


Figure 5.6: A counter example against work-conserving scheduling

for future periods.

Assume that we adopt other scheduling schemes that allow a partition to run more than B_i time slots within a period. This is true for the constant bandwidth server, which lowers a partition's priority as it runs longer, but never really cuts it off until another higher-priority partition is ready to run.

Fig. 5.6 shows a counterexample against work-conserving scheduling. In the canonical execution, the partition budget is not strictly enforced since there is only one partition in the system. This allows τ_1 to finish before the next period of τ_0. However, when there are more partitions in the system, in the worst case, Π_0 can only be scheduled for two time slots within each period. In this way, τ_1 will be preempted by τ_0 in its second period, causing inconsistency with the canonical execution.

What's more, even if we force the actual schedule to mimic the canonical one, this will lead to an unbounded delay of τ_0 and jeopardize the schedulability of the system. Thus, we don't consider work-conserving scheduling for partitions as a suitable choice if they are supposed to enjoy temporal isolation between each other.

5.3 Schedulability in Partitioned Scheduling

The isolation property between 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, Def. 9 ensures that there is no interference across partitions. Moreover, it connects a task's scheduling with its local behavior, allowing us to transform schedulability, which is specified on the global timeline, into a property on the partition's local timeline.

Lemma 7. *The schedulability analysis of task $\tau_j \in \Pi_i$ is transformed into reasoning about its local schedule ω_j. That is, for any j and $k \geq 0$, assume that each task's period is a multiple of the partition period, and this partition enjoys temporal isolation as defined in Def. 9, the following must hold.*

$$\begin{aligned} C_j &\leq \sigma_j^{\Pi_0 \ldots \Pi_M}((k+1) * T_j) - \sigma_j^{\Pi_0 \ldots \Pi_M}(k * T_j) \\ \Leftrightarrow C_j &\leq \omega_j^{\Pi_i}((k+1) * T_j * \frac{B_i}{P_i}) - \omega_j^{\Pi_i}(k * T_j * \frac{B_i}{P_i}) \end{aligned}$$

Proof. By Def. 9, we transform the requirement into a task's local schedule.

$$\begin{aligned} & \sigma_j^{\Pi_0 \ldots \Pi_M}((k+1) * T_j) - \sigma_j^{\Pi_0 \ldots \Pi_M}(k * T_j) \\ = \; & \omega_j^{\Pi_i} \circ \pi_j^{\Pi_0 \ldots \Pi_M}((k+1) * T_j) - \omega_j^{\Pi_i} \circ \pi_j^{\Pi_0 \ldots \Pi_M}(k * T_j) \end{aligned}$$

Since the OS kernel guarantees the full amount of budget for the partition within each period, and that T_j is a multiple of the partition period P_i, we prove that

$$\forall k, \pi_j^{\Pi_0 \ldots \Pi_M}(k * T_j) = kT_j * \frac{B_i}{P_i} = kT'_j$$

This essentially transforms a physical period into a shrunk period from the partition's point of view, thus,

$$\begin{aligned} & \omega_j^{\Pi_i} \circ \pi_j^{\Pi_0 \ldots \Pi_M}((k+1) * T_j) - \omega_j^{\Pi_i} \circ \pi_j^{\Pi_0 \ldots \Pi_M}(k * T_j) \\ = \; & \omega_j^{\Pi_i}((k+1)T'_j) - \omega_j^{\Pi_i}(kT'_j) \end{aligned}$$

Thus, this lemma holds. □

The above lemma transforms a task's schedulability into the reasoning on its partition's local timeline, such that it is schedulable if and only if the shrunk task ($\tau'_j = (C_j, T_j\frac{B_i}{P_i})$) is schedulable.

Chapter 6

Case Study: CertiKOS with Partitioned Scheduling

In this chapter, we discuss how we implement partitioned scheduling in CertiKOS and how to connect it with the virtual timeline abstraction. In particular, we apply virtual timelines in two different settings: TDMA (Time-Division Multiple Access) schedules and dynamic schedules for partitions. Throughout this chapter, 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, $\omega_p^{\Pi_i}$ as in Def. 10, and we discuss how to connect it with the concrete scheduler implementation.

6.1 Static partitions scheduled under TDMA

We first focus on the TDMA-based partitioned scheduling as in ARINC 653 standards. This is a restrictive setting where 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 the k-th period, this partition 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. It must be of the following form.

```
int TDMA_sched(){
  t++;

  ......
  if (δ_i <= t%P && t%P < δ_i + B_i){
    return sched_local_i();
  }
  ......
}
```

Figure 6.1: The top-level TDMA scheduler

Definition 11 (Partition scheduling under TDMA)**.**

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

There are multiple alternatives to implementing the partition-local scheduler, depending on whether we allow partitions to choose their local scheduling algorithm. For example, if we require that all partitions must adopt fixed-priority scheduling, a suitable choice would be to let them share the same scheduler but with different task lists. In this chapter, we emphasize isolation and allow partitions the flexibility of choosing their own scheduler. As a consequence, we need to maintain a separate scheduler for each partition.

Fig. 6.1 illustrates the top-level TDMA scheduler. It uses the current time to decide which partition to schedule and invokes the corresponding partition's local scheduler. This essentially disentangles the top-level scheduling with the local scheduling. Instead of designing a virtual-time-based top-level scheduler, we prove that invoking the implementation of the local scheduler is equivalent to invoking its virtual-time-based abstraction. Thus, the top-level scheduler is instead built on top of a series of virtual-time-based local schedulers. This takes advantage of the abstraction layer approach, where only the specification, instead of the implementation of lower layers, is exposed to upper layers.

We assume Π_i uses the same fixed-priority scheduling for its tasks as discussed in Sec. 2.2. However, its implementation is slightly different because this partition only occupies a fixed portion of the processor time. As shown in Fig. 6.2, the local scheduler for partition Π_i first replenishes budgets for each task, then finds the highest-priority ready one in the same

```
1 int sched_local_i() {
2   for(int i = 0; i < N; i++){
3     if (t % T_i == δ_i){
4       quanta[i] = C_i;
5     }
6   }
7
8   // fixed-priority scheduling
9   ......
10 }
```

Figure 6.2: The local scheduler for partition Π_i

```
1 int vt_sched_local_i(){
2   lt++;
3   int pid = N;
4   for(int j = 0; j < N; j++){
5     if(VTA(ω_j^{Π_i}, lt) == 1){
6       pid = j;
7       break;
8     }
9   }
10
11   return pid;
12 }
```

Figure 6.3: C illustration of the virtual-time-based local scheduler for Π_i

```
1 int sched_local_i() {
2   for(int j = 0; j < N; i++){
3       if (π_i(t) % (B_i/P)T_j == 0){
4       quanta[j] = C_j;
5     }
6   }
7
8   // fixed-priority scheduling
9   ......
10 }
```

Figure 6.4: The intermediate local scheduler for Π_i

way as in Fig. 2.2. However, its virtual-time-line based abstraction (shown in Fig. 6.3) is different from that in Fig. 4.4. It maintains a local time for this partition, instead of relying on global time. It iterates over local time maps for tasks, instead of using global time maps. All of these differences result from the fact that partition Π_i is only given a portion of the CPU resource.

Now the only proof obligation is to show that the local scheduler and its virtual-time-based abstraction are indeed equivalent. This is more challenging than in Sec. 4.3 due to the following discrepancy: the local scheduler implementation relies on original task parameters while the virtual-time-based scheduler makes use of the shrunk set of task parameters. This makes it hard to reuse existing proofs directly.

To mitigate this gap, we first prove that this local scheduler implementation is equivalent to an intermediate version (Fig. 6.4) which uses the partition local time and the shrunk set

of task parameters. Notice that π_i is defined in Def. 11.

Lemma 8 (The local scheduler is equivalent with the intermediate version which relies on partition local time and the shrunk set of task parameters). *Assume that the current time slot belongs to partition Π_i, and the quanta array for tasks is Q. The local scheduler updates it to Q' and schedules task τ_l. The intermediate version must also update it to Q' and schedule task τ_l.*

Proof. The only place that a task's period matters is when the scheduler replenishes the budget for each task. We prove that the period of a task in the global timeline is always aligned to its shrunk period in the partition's local time, such that

$$t \mod T_j = \delta_i \Leftrightarrow \pi_i(t) \mod T'_j = 0, \quad \textit{where} \;\; T'_j = \frac{B_i}{P} T_j$$

This is done through pure arithmetic reasoning. It 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.

On top of this, we prove that the two schedulers refill budgets for the same set of tasks, such that the resulting quanta array is the same, which leads to the same schedule and identical resulting quanta array. □

Finally, we prove that the intermediate version is equivalent to the virtual-time-based abstraction, in the same way as described in Sec. 4.3. This relies on a proof that the local time maintained by the virtual-time-based scheduler, `lt`, is always consistent with $\pi_i(t)$ when this partition is scheduled.

In this way, we connect the virtual timeline abstraction with one partition scheduled under TDMA. Scheduling algorithms used by other partitions are irrelevant to this partition.

6.2 Dynamically scheduled partitions

The scheduling scheme mentioned in the previous section indeed ensures temporal isolation since it imposes a fixed schedule for each partition. However, it is also restrictive in the

sense that this policy may cause a high-priority task to be delayed while another low-priority one from a ready partition gets the chance to run. This is not an ideal policy in terms of responsiveness.

It is possible to make a tradeoff between isolation and responsiveness. Since tasks are not allowed to access the physical time directly, they are oblivious to the exact schedule in the global timeline, giving us the opportunity to vary the enclosing partition's schedule. However, this new scheme should still guarantee schedulability, since tasks are sensitive to deadline misses. It should also preserve the relative ordering between tasks as discussed in Sec. 5.2, which we can prove following the same approach in the previous section.

There are many partitioned scheduling algorithms that allow us to vary the schedule of partitions to achieve better average case responsiveness [28, 15, 29]. They usually still enforce the budget on each partition, i.e. each partition only receives a portion of CPU resource specified by its period P_i and budget B_i. Yet, they differ in the actual way of implementing this enforcement, such as whether the time instant for budget refill is fixed or dynamic, whether the refill is achieved at once or supplied gradually, etc. Other than responsiveness, we carefully weigh its implication for the isolation proof when making our design decisions.

In this section, we focus on a scheme with dynamically scheduled partitions. Under this setting, each partition still adopts a local scheduler to decide its own task schedules. However, instead of imposing a fixed priority for each partition, we allow users to specify priorities among tasks across partitions. In this way, the priority between two partitions is decided by the priority between the task each of them proposes to execute, and is thus dynamic.

This design draws inspiration from QNX. When all partitions adopt the same local policy, say, fixed-priority scheduling, the overall behavior of the system is equivalent to scheduling all tasks together, except that they are also subject to a partition budget. In this sense, partitioning serves as a high-level mechanism for distributing time resources. However, our current design is more generic in that it allows different partitions to adopt different local policies. Yet, it keeps the benefit of imposing cross-partition priorities for tasks that could lead to better average-case responsiveness.

```
1  int select(int id0, int id1){
2    if (globalPrio[id0] <
       globalPrio[id1] &&
3        parQuanta[0] > 0){
4      return 0;
5    }else{
6      return 1;
7    }
8  }
9
10 int dynamic_sched(){
11   t++;
12
13   // refill partition budgets
14   ......
15
16   /* each partition proposes a
        task to schedule */
17   int id0 = local_propose0(t);
18   int id1 = local_propose1(t);
19
20   // select a partition
21   int par = select(id0, id1);
22
23   // schedules the partition
24   parQuanta[par]--;
25
26   if (par == 0){
27     return local_sched0();
28   }else{
29     return local_sched1();
30   }
31 }
```

Figure 6.5: The global scheduler for dynamic partitions

Implementation of the global and local scheduler Recall that Fig. 5.3 illustrates the global scheduling when all partitions adopt the same local policy. In this section, we demonstrate how to implement a more generic alternative such that each partition could choose its own policy. For simplicity, we assume there are two partitions in the system, each has a partition budget B_i and shares a common partition period P. The global scheduler is shown in Fig. 6.5. Here, it lets each partition propose a task to execute (`local_propose`). It then selects a partition to schedule based on the proposed task, decrements the partition's budget, and finally schedules that partition. Notice that at any moment, invoking `local_propose` and `local_sched` always returns the same task. However, the former does not have side effects while the latter may update partition-local data structures.

Also, notice that the exact implementation of `select` can be diverse. It only needs to guarantee that the selected partition indeed has remaining budget, but the policy for imposing priorities can be arbitrary. Finally, even though this example code only accommodates two partitions, it is straightforward to extend it with any arbitrary number of partitions. The only requirement is that each additional partition implements its own `local_propose` and `local_sched`, and make sure that they are consistent with each other.

Assume that the partition we investigate, Π_0, uses fixed-priority scheduling for its tasks. Fig. 6.6 shows its implementation of the two functions. Here, in addition to the `quanta`

```
int local_propose0(){
  int pid = N;

  for(int i = 0; i < N; i++){
    if (quanta[i] > 0 ||
      index[i] != t / T_i){
      pid = i;
      break;
    }
  }

  return pid;
}



int local_sched0(){
  lt++;

  for(int i = 0; i < N; i++){
    if (index[i] != t / T_i){
      index[i] = t / T_i;
      quanta[i] = C_i;
    }
  }

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

  return pid;
}
```

Figure 6.6: The local scheduler for a partition

which maintains the remaining budget for each task, it uses an `index` array to track whether there is a new period arriving for each task. Each time `local_sched0` is invoked, it checks whether a task's index corresponds to the current period. If not, it updates both the index value and this task's remaining quantum. This is necessary because the progress of task periods in a partition may no longer be aligned with its period in global time. More specifically, a partition's `quanta` array may not be updated even if $t \mod T_i = 0$ holds for a certain task τ_i.

The distinction between `local_propose` and `local_sched` simplifies the reasoning of this scheduler because it ensures that only the scheduled partition's state will be updated. Of course, there're functionally equivalent ways of implementing the same scheduling scheme, which update task arrivals for all partitions regardless of whether they will be scheduled or not. It is possible to prove that they are equivalent to the scheduler as shown in Fig. 6.5 and Fig. 6.6, so that all properties proved in this section also hold on those alternative implementations.

Refinement into an intermediate scheduler To connect the concrete scheduler with the canonical local time maps of tasks, we need to first prove that it is equivalent to an intermediate form that relies on local time instead of global time to schedule each partition.

```
int local_propose0_inter(){
  int pid = N;

  for(int i = 0; i < N; i++){
    if (quanta[i] > 0 ||
      index[i] != (lt+1) / (B_0/P)T_i){
      pid = i;
      break;
    }
  }

  return pid;
}



int local_sched0_inter(){
  lt++;

  for(int i = 0; i < N; i++){
    if (index[i] != lt / (B_0/P)T_i){
      index[i] = lt / (B_0/P)T_i;
      quanta[i] = C_i;
    }
  }

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

  return pid;
}
```

Figure 6.7: The intermediate version of the local scheduler for a partition

The global scheduler for partitions remains the same, as in Fig. 6.5. The local scheduler is illustrated in Fig. 6.7. Instead of relying on the global time to trigger new task arrivals, this intermediate version maintains a local time, `lt`, to schedule tasks within a partition. Further, it uses the shrunk set of task parameters to accommodate this distorted view of time.

The equivalence between the concrete scheduler implementation and its intermediate abstraction relies on the fact that the global time and the local time are always aligned when the local scheduler is invoked. Instead of proving an equivalence relation directly, we maintain local time without using it in Fig. 6.6. This is purely an abstract variable, incurring no runtime overhead. However, it helps us transform the equivalence proof into an invariant proof, which is local to a single function and easier to verify. That proof again relies on the remaining partition budget as a bridge. Through the rest of this section, we use t and lt to represent the global and local time, respectively. And we use b to abbreviate `parQuanta[i]`, representing the remaining partition budget. The full partition budget is B and its period is P.

Lemma 9 (The remaining partition budget correctly tracks the progress of partition local

time). *At any moment, the following equivalence relation hold.*

$$\begin{aligned} & b \neq 0 \Leftrightarrow (lt+1) \mod B + b = B \\ \wedge \quad & b = 0 \implies (lt+1) \mod B = 0 \\ \wedge \quad & (t+1) \mod P = 0 \implies b = 0 \end{aligned}$$

Proof. Since only the scheduler might update time and schedule-related data structures, the proof obligation rests on showing that invoking the scheduler (Fig.6.5 and Fig. 6.6) will preserve the above invariants.

Assuming that these invariants hold at time t. We prove that they also hold at time t + 1. We perform a case analysis on whether this is the end of a partition period or not.

1. When $(t+1) \mod P = 0$ holds, this is the end of a partition period.
   - If this partition is scheduled, we prove that its partition budget must be 1 at time t, which entails that

     $$lt \mod B = B - 1$$

     At its schedule for one time slot, we know the current budget $b = 0$, and that $(lt+1) \mod B = 0$. Thus, these invariants hold.
   - If this partition is not scheduled, we prove that it must have been finished, such that its partition budget and local time stays the same. Thus, these invariants hold.
2. When $(t+1) \mod P \neq 0$ holds, this is the beginning or middle of a partition period.
   - If this partition is scheduled, its partition budget must be decremented by 1 time slot, while its local time increments by 1. Thus, the relation that $(lt+1) \mod B + b = B$ is preserved.
   - If this partition is not scheduled, its local time and partition budget stay the same. Thus, these invariants hold.

This concludes the proof that a partition's remaining budget and its local time are always synchronized. □

On top of the above lemma, we prove that the partition local time and global time are always synchronized, as well. Notice that they may go out of sync when the partition has finished its current period while the global time has not. As a result, we carefully specify that this alignment only holds when the partition is still active.

Lemma 10 (The global time and partition local time are always aligned when the local scheduler is invoked)**.** *At any moment, the following invariants hold for each partition.*

$$\begin{aligned} & (t+1) \mod P \neq 0 \wedge b \neq 0 \implies (t+1)/P = (lt+1)/B \\ \wedge \quad & (t+1) \mod P \neq 0 \wedge b = 0 \implies (t+1)/P + 1 = (lt+1)/B \\ \wedge \quad & (t+1) \mod P = 0 \wedge b = 0 \implies (t+1)/P = (lt+1)/B \end{aligned}$$

Notice that we already prove the schedulability of each partition, such that $(t+1) \mod P = 0 \wedge b \neq 0$ *is not a valid case.*

Proof. Assuming that these invariants hold at time t. We prove that they also hold at time t + 1. We again perform case analysis on whether time instant t is the end of a partition period or not.

1. When $(t+1) \mod P = 0$ holds, this is the end of a partition period. Before the scheduler is invoked, we know that $(t+1)/P = (lt+1)/B$. We also prove from Lemma 9 that $(lt+1) \mod B = 0$. In other words, this is the end of the period both at the global timeline and at the partition's local timeline.

   After the invocation, the global time t must increment by 1, and the local time either stays the same or increments by 1, as well. In either case, we prove that

   $$(t+2)/P = (t+1)/P \wedge (t+2) \mod P \neq 0$$

   and

   $$(lt+2)/B = (lt+1)/B$$

   Thus, the invariants hold.

2. When $(t+1) \mod P \neq 0$ holds, this is the beginning or middle of a partition period.

- If $(t+2) \mod P \neq 0 \land b \neq 0$, we know that t is in the middle of a partition period. In this case, the local time either stays the same or increments by 1, depending on whether this partition is scheduled or not.
  - If lt stays the same, we prove that these invariants hold because $(t+2)/P = (t+1)/P$, while $(lt+1)/B$ stays the same.
  - Otherwise, we prove that $(lt+2)/B = (lt+1)/B$ since it will not finish the current period right away.
- If $(t+2) \mod P \neq 0 \land b = 0$, we know one of the following must hold.
  - This partition is already finished at time t, and cannot be scheduled until the next period. In this case, lt stays the same, and the invariants hold.
  - This partition has only 1 remaining time slot at time t, and it is scheduled and finishes at time t+1. In this case, we prove that $(lt+2)/B = (lt+1)/B+1$, such that the invariants hold.
- If $(t+2) \mod P = 0 \land b = 0$, we prove that $(t+2)/P = (t+1)/P+1$. We know that one of the following must hold regarding the execution of the partition.
  - This partition is already finished at time t, and cannot be scheduled until the next period. In this case, lt stays the same, and we prove that

    $$(lt+1)/B = (t+1)/P + 1 = (t+2)/P$$

  - This partition has only 1 remaining time slot at time t, and it is scheduled and finishes at time t+1. In this case, we prove that

    $$(lt+2)/B = (lt+1)/B + 1 = (t+1)/P + 1 = (t+2)/P$$

  In both cases, the invariants hold.

This concludes the proof that the global time and a partition's local time are always synchronized. □

On top of this, we prove that t/T_i is always consistent with $lt/\frac{B_0}{P}T_i$, such that the

concrete scheduler and its intermediate abstraction maintains equivalent `index` array and further equivalent `quanta` array. This helps us prove that the two schedulers are indeed equivalent to each other.

Refinement into a virtual-time-based scheduler The virtual-time-based local scheduler is the same as shown in Fig. 6.3. More specifically, Fig. 6.3 shows the virtual-time-based `local_sched`, while we only need to remove the update to the local time to obtain `local_propose`. The intuition is that if a partition indeed enjoys isolation from other partitions, its local scheduling of tasks does not depend on the top-level scheduling algorithm. In other words, the isolation property allows us to disentangle a partition's local schedule from the system's top-level scheduling of partitions. And we observe that the main obligation rests in proving that the concrete scheduler implementation is equivalent to an intermediate version that is oblivious to global time. As a comparison, the refinement proof between this intermediate version and the virtual-time-based scheduler is straightforward: it follows the same approach as described in Sec. 4.3.

6.3 A Perspective on the Interface for a Real-Time Partition

Previous sections discuss the implementation of various partitioned-scheduling schemes and how to connect them with the virtual timeline abstraction. This section provides the author's perspective on a different issue: the user interface. This section is largely orthogonal to the real-time reasoning in this work, because it only matters for the initialization of the system, while our reasoning always assumes that the system starts from a valid initial state.

First of all, the user needs to specify a common period for all partitions in the system. This is necessary for both the TDMA scheduling and the top-level scheduler as shown in Fig. 6.5.

To spawn a partition, the user specifies both the partition budget and the desired local scheduling policy. The policy can be fixed-priority scheduling, earliest-deadline-first scheduling, or round-robin scheduling (for batch tasks only). The OS kernel implements the set of `local_propose` and `local_sched` to accommodate all these choices, such that

the local scheduler used by each partition is still verified, despite the flexibility regarding scheduling policies.

Finally, when adding a task into a partition, the user needs to specify the period and budget requirement for this task. These parameters can be ignored if that partition adopts round-robin scheduling.

This is a general perspective on how different local scheduling policies could reside on the same platform, and how they are exposed to the user. Again, they only matter for the configuration of the system and are orthogonal to the reasoning of the temporal properties of components.

Chapter 7

Generalization with Dynamic Priority Assignment

Previous chapters demonstrate the successful application of the virtual timeline in the verification of fixed-priority scheduling and partitioned scheduling. Under these schemes, the composition of multiple components onto one system either imposes constraints on the component parameter (in partitioned scheduling, it is common to design partition periods to be aligned with each other) or requires sophisticated schedulability test (in fixed-priority scheduling, the priority, period and budget of each task has to be known a priori).

These are suitable for closed systems, where workloads are fixed and predictability is preferred. However, in open systems with dynamic workloads, where tasks arrive on the fly and the underlying OS kernel needs to decide whether to admit or reject them based on the current capacity of the system, a more flexible scheduling scheme is needed from both performance and verification perspectives.

This chapter discusses the generalization of the virtual timeline with a dynamic priority assignment. In particular, we adopt the earliest-deadline-first scheduling algorithm and show that the virtual timeline abstraction is powerful enough to address this more complex algorithm.

7.1 Earliest-Deadline-First and Compositionality of Workloads

In systems with dynamic workloads, the task set is not fixed thus not known a priori. The temporal parameters of real-time tasks, such as the budget and period, become known only when they arrive. Hence, a run-time *admission control* algorithm is employed to decide whether a new request can be granted without jeopardizing the schedulability of existing workloads.

The admission control mechanism should be able to utilize the CPU resource as much as possible while guaranteeing real-time tasks' timing constraints. Furthermore, the algorithmic complexity of its test should be low and implementable in an efficient way so that frequent invocations of the algorithm do not incur too much overhead at the runtime. Based on these criteria, fixed-priority scheduling is not suitable for dynamic workloads because (i) its exact test relies on iterative computation and is hard to scale and (ii) the utilization-bound test may lead to under-utilization of the CPU.

On the contrary, as briefly mentioned above, the earliest-deadline-first (EDF) scheduling permits a simple and exact schedulability test. In particular, Liu and Layland [22] showed that any task set whose total utilization (i.e., the sum of a task's budget over its period) does not exceed 100% is schedulable under EDF. Thus, it suffices to implement an admission controller that simply records the total utilization of exiting tasks and tests whether the addition of a new task would cause the total utilization to exceed 100%.

This also benefits the reasoning side from the compositionality perspective. When multiple components reside on the same platform, the resource provisioning for one component inevitably relies on the resource bound on others. However, the granularity of such bound matters. In fixed-priority scheduling, the schedulability of one component is sensitive to the exact period and budget of all higher-priority tasks. This is a restrictive setting because an update to one component's parameter may affect the schedulability of all lower-priority components. As a comparison, in EDF scheduling, a component is only sensitive to the total utilization of all others. This entails more flexibility since one or multiple components are free to update their parameters as long as they satisfy a total utilization bound. This is

useful in the design phase of a system, where CPU resource is divided based on utilization and distributed to components developed by different vendors. Each vendor can select its component parameters independently as long as they satisfy the utilization bound.

An EDF scheduler works by scheduling the ready task whose deadline is the nearest, or informally, most urgent. This is called dynamic priority assignment since a task's relative priority varies, depending on the current workload (i.e., the exact ordering of deadlines of all admitted tasks). In other words, a task's priority may range from the highest to the lowest. This leads to intricate interferences among tasks, making the formal verification of the scheduler challenging.

7.2 Generalization: Virtual Timeline with Dynamic Priority Assignment

This section discusses how to extend the virtual timeline abstraction to accommodate dynamic priority assignment.

As explained in Sec. 2.3, the virtual time available to a task includes time available to itself and all lower-priority tasks. This is suitable for fixed-priority scheduling, which implicitly forms a hierarchy among tasks according to their priorities and allows a static construction of the time map one task after another (Def. 1).

However, the formalization of the time map is not obvious with a dynamic priority assignment. The scheduling no longer enjoys a static hierarchy which allows us to consider the time maps in the priority order. For example, with EDF scheduling, even if task τ_i has the highest-priority at time 0, it is difficult to compute its schedule for all time since its priority varies.

The challenge described above calls for a closer look into the original virtual timeline abstraction. In particular, we consider its relevance by looking into a concrete scheduler with a dynamic priority assignment and compare it with a fixed-priority scheduler.

Fig. 7.1(a) demonstrates a concrete scheduler with a dynamic priority assignment. It is parameterized over a priority queue PriQ, while $PriQ_t(p)$ represents the exact task at priority level p at time instant t. Notice that this value may vary depending on the current

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

  int tid = N;
  for (p = 0; p < N; p++) {
    // The dynamic priority queue
    int i = PriQ_t(p);
    if (tid == N){
      if (quanta[i] > 0){
        quanta[i]--;
        tid = i;
      }
    }else{
      continue;
    }
  }
  return tid;
}
```

(a) C code of the concrete scheduler with dynamic priority assignment

```c
int dynamic_vtsched(){
  t++;

  int tid = N;
  for (p = 0; p < N; p++) {
  // The dynamic priority queue
  int i = PriQ_t(p);
    if (tid == N){
      σ_i[t+1] = σ_i[t] + 1;
      if (VTA(i, t)){
        tid = i;
      }
    }else{
      σ_i[t+1] = σ_i[t];
    }
  }
  return tid;
}
```

(b) The iterative computation of time maps embedded in a virtual-time-based scheduler

Figure 7.1: The concrete and virtual-time-based scheduler

time t, thus is called dynamic priority. This is a generalized form of dynamic priority scheduling. In fact, the fixed-priority scheduler in Fig. 2.2 can be viewed as one of its special cases:

$$\forall t, p, PriQ_t(p) = p$$

We make two interesting observations as follows.

1. At each moment, this scheduler can still be viewed as a fixed-priority scheduler: the highest-priority ready task (has remaining budget) gets scheduled. This leads to the same motivation when the virtual timeline is first introduced for fixed-priority scheduling: encapsulate interferences from higher-priority tasks such that the schedule of a task can be decided by looking at its virtual timeline alone. In particular, the use of `VTA` (Fig. 4.5) still applies in this setting.

2. This scheduler also uses a `quanta` array to enforce budget constraints for each task. If a

task indeed has exclusive access to its virtual timeline, i.e. is scheduled immediately on this timeline whenever it has remaining budget, the same virtual-time-based scheduler (Fig. 4.4) can be applied and proven to be equivalent with the concrete scheduler implementation.

As discussed above, the point of employing the virtual timeline is to hide time slots that are occupied by higher-priority tasks, thus creating an illusion of exclusive access. Once achieved, this enables an equivalent scheduler entirely based on virtual timelines. Though it is impossible to adopt a static construction of time maps as shown in Def. 1, we define it iteratively with the advancement of time as shown in Fig. 7.1(b). We highlight its difference with Fig. 4.4 by attaching a comment before the dynamically computed priority queue.

Here, the virtual time map σ is a ghost variable only residing in the Coq abstraction and is instantiated iteratively. At time t, all time maps ($\{\sigma_0, \sigma_1, ..., \sigma_{N-1}\}$) are valid up to t. Then the scheduler increments the time counter to $t+1$ and selects the highest-priority ready task to schedule. For this task and all higher-priority ones, their virtual time is incremented by 1 because this time slot is available to them. For lower-priority tasks, their virtual time stagnates because they are preempted.

This enables the same reasoning as in Sec. 4.3, that a virtual-time-based scheduler connects the virtual timeline with the concrete scheduler implementation, which relies on the following two steps.

Equivalence between VTA and the virtual-time-based scheduler. A task is scheduled if and only if it has remaining budget and that its virtual time indeed increments (`VTA(i, t)` $= 1$). The justification is straightforward.

- Necessity. If task τ_i is scheduled by `dynamic_vtsched` at time t, $\sigma_i(t+1) = \sigma_i(t) + 1$ must hold and the time consumption of τ_i in this period must be smaller than C_i. Thus, `VTA(i, t)` $= 1$ holds.
- Sufficiency. If `VTA(i, t)` $= 1$ holds, $\sigma_i(t+1) = \sigma_i(t) + 1$ must also be true, thus all higher priority tasks must have used up their budget. Otherwise, task τ_i will be preempted. As a result, τ_i has the highest priority among ready tasks and is therefore

scheduled.

The contextual refinement between the virtual-time-based scheduler and the concrete scheduler implemented in C. In fixed-priority scheduling, the virtual-time-based scheduler is equivalent with the concrete scheduler because `VTA` faithfully describes a task's behavior: if a task's budget is depleted judged on its virtual timeline, its remaining budget value (`quanta[i]`) maintained by `sched` (Fig. 2.2) also reaches 0, and vice versa.

This reasoning applies to the case with a dynamic priority assignment. We observe that the virtual-time-based scheduler with a dynamic priority assignment (Fig. 7.1(b)) only differs from that in fixed-priority scheduling (Fig. 4.4) in the way it assigns priority to tasks. Similarly, the concrete scheduler with a dynamic priority assignment differs from that in fixed-priority scheduling in that it relies on a priority queue to calculate the exact task occupying certain priority levels. Other than these differences, each time a task τ_i is scheduled, `quanta`[i] decrements while σ_i increments at the same pace, allowing us to establish the equivalence between these two schedulers under the setting of a dynamic priority assignment.

Chpt. 8 discusses in more detail how to connect the virtual timeline with an EDF scheduler implementation.

7.3 Earliest-Deadline-First: the Schedulability Proof

This section discusses the schedulability proof for the EDF scheduling algorithm. We assume that the set of tasks is fixed, that all tasks arrive in a strictly periodic fashion and always use up their budgets in each period.

7.3.1 The Proof Sketch: Schedulability Through Transformation

This section explains the proof sketch for the EDF scheduling algorithm. Unlike fixed-prority scheduling, there is no hierarchy among tasks since their relative priorities vary depending on their deadlines at any particular moment. Thus, the proof by induction from higher-priority tasks to lower-priority ones no longer makes sense under this setting.

Instead, we reflect on the schedulability test for EDF which only considers the utilization of tasks. An interesting implication is that we are free to change a task's parameters (period and budget) without affecting the schedulability of the system as long as its utilization remains the same. This makes the schedulability test intuitive: for any task set, we can change tasks' periods to be identical without affecting their schedulability. Given this new task set, it's trivial to see that it is schedulable if the total utilization of tasks does not exceed 100%. Thus, the proof obligation rests in showing that tweaking a task's parameters preserves the schedulability of the system.

This is not a novel insight. Wilding's proof [30] on the EDF scheduling also relies on its optimality, which states that any schedulable task set is schedulable under EDF, enabling the adjustment of task parameters. However, its proof involves a complex reordering of the schedule and it is not obvious how that can be connected with the concrete implementation of an EDF scheduler.

Consider a task set Π that contains the following tasks: $\tau_0 = (1, 4)$, $\tau_1 = (2, 5)$ and $\tau_2 = (3, 10)$. We use $\Omega^{EDF}(\Pi)$ to denote that the task set Π is schedulable under EDF scheduling. Then a schedulability proof for the task set Π is obtained as follows.

Example 1. *The schedulability proof for* $\Pi = \{\tau_0, \tau_1, \tau_2\}$

$$\begin{aligned}
& \Omega^{EDF}(\{(5,20),(8,20),(6,20)\}) \\
\Longrightarrow\quad & \Omega^{EDF}(\{(20,80),(8,20),(6,20)\}) \\
\Longrightarrow\quad & \Omega^{EDF}(\{(20,80),(40,100),(6,20)\}) \\
\Longrightarrow\quad & \Omega^{EDF}(\{(20,80),(40,100),(60,200)\}) \\
\Longrightarrow\quad & \Omega^{EDF}(\{\tau_0,\tau_1,\tau_2\})
\end{aligned}$$

The first step is to *enlarge* all tasks to the same period, which is 20 (i.e., the least common multiple of the task periods) in the example. Here, enlarging a task by a factor of k means multiplying both its budget and period by k such that its utilization is preserved. In the following steps, we enlarge the first, second and third task by a factor of 4, 5 and 10 respectively. In the end, we *shrink* (the exact opposite of enlarge) all tasks by a factor of 20 to transform the task set back to its original parameters.

It's easy to see that all tasks are schedulable if they have a common period and the total

```
Fixpoint enlarge_to_hyper (H: Z) (N: nat) (conf: PrioConfigPool) :=
  match N with
  | O ⇒ conf
  | S n ⇒ match (ZMap.get (Z.of_nat n) conf) with
          | mkPrioConfigValid T C ⇒
            ZMap.set (Z.of_nat n) (mkPrioConfigValid H (H / T * C))
                     (enlarge_to_hyper H n conf)
          end
  end.
```

Figure 7.2: Enlarging all tasks to a common period

```
Inductive enlargeParam: Type :=
| EParam: Z (* task ID *) → Z (* factor *) → enlargeParam.

Fixpoint get_enlarge_params_aux (Ni: nat) (confi: PrioConfigPool) : list enlargeParam :=
  match Ni with
  | O ⇒ nil
  | S n ⇒ match (ZMap.get (Z.of_nat n) confi) with
          | mkPrioConfigValid T C ⇒
            EParam (Z.of_nat n) T :: get_enlarge_params_aux n confi
          end
  end.

Definition enlarge_sequence: list enlargeParam :=
  eparamsort.sort (get_enlarge_params_aux (Z.to_nat N) conf).
```

Figure 7.3: The sequence of enlarging operations to be applied to a task set

utilization does not exceed 100%. Subsequent steps in the above outline also rely on the proof that both enlarging a task and shrinking the whole task set preserve the schedulability of the system, which we detail in Sec. 7.3.3 and Sec. 7.3.4.

The rest of this section demonstrates the Coq formalization of the above outline.

Fig. 7.2 shows how we take a task set `conf` and adjust all its tasks to a common period. Here, `H` represents the hyper period (i.e. the least common multiple of the task periods) of this task set. `N` represents the number of tasks in this task set. We denote `enlarge_to_hyper H N conf` as `conf0`. Relevant data structures, such as `PrioConfigPool`, are explained in Fig. 4.1.

Fig. 7.3 shows how we prepare the sequence of enlarging operations to a task set. Here,

```
1  Function do_enlarge (id k: Z) conf: PrioConfigPool :=
2    match (ZMap.get id conf) with
3    | mkPrioConfigValid T C ⇒
4      ZMap.set id (mkPrioConfigValid (k * T) (k * C)) conf
5    end.
6
7  Fixpoint do_enlarge_sequence (confi: PrioConfigPool) (l: list enlargeParam) :=
8    match l with
9    | nil ⇒  confi
10   | EParam id k :: tl ⇒
11     do_enlarge_sequence (do_enlarge id k confi) tl
12   end.
```

Figure 7.4: Apply a sequence of enlarging operations to a task set

an enlarge operation, `enlargeParam` is defined with a task ID and the exact factor of this enlarging. Function `get_enlarge_params_aux` computes the enlarging parameters for all tasks. For each of the tasks, its factor of enlarging equals its original period, as shown in Example. 1, which guarantees that each task shrinks to its original parameters after the final shrinking operation. In the end, `enlarge_sequence` orders all operations in non-decreasing order of the enlarging factor to obtain the sequence of operations to be applied. This ordering is important for reducing some of the subsequent proof burdens. It actually imposes one more constraint to the task set: after each step of enlarging, the enlarged task has the longest period in the new task set. Sec. 7.3.3 shows how this additional constraint helps eliminates some proof burden.

Fig. 7.4 demonstrates the application of the enlarging sequence to a task set. Based on this, we prove the schedulability of a task set as follows.

Theorem 2. *A task set* `conf` *is schedulable under EDF if the total utilization of its tasks does not exceed 100%.*

Proof. The schedulability criterion that the total utilization must not exceed 100% is defined as the sum of task budgets of `enlarge_to_hyper H N conf`, also denoted as `conf0`, is less than or equal to `H`. In `conf0`, all tasks have a common period of `H`, and the schedulability of the system is trivial.

Then we prove that invoking `do_enlarge_sequence` leads to a schedulable task set.

This is done by proving that at any step, the schedulability of a task set is preserved by `do_enlarge`, which also relies on the ordering of `enlarge_sequence`. We denote the result of applying all operations as `conf1`.

In the end, we prove that every task τ_i' in `conf1` is the result of enlarging τ_i by a factor of `H` in `conf` and that the schedulability of `conf1` entails the schedulability of `conf`. □

The above proof follows the same outline as in Example 1, where `conf` represents $\{(1,4),(2,5),(3,10)\}$, `conf0` represents $\{(5,20),(8,20),(6,20)\}$, and `conf1` represents $\{(20,80),(40,100),(60,200)\}$. Proof obligations involved in individual steps are explained in the following sections.

7.3.2 The Virtual Time Map and Interference

The dynamic computation of virtual time maps As shown in Fig. 7.1(b), EDF scheduling no longer permits the static computation of time maps in decreasing order of the task priority. Instead, it can only be defined by recursion on the time tick: time maps for all tasks up to time $t+1$ can be computed based on time maps up to time t.

The computation relies on a concrete definition of the priority queue $PriQ_t$ at any arbitrary moment t. This is achieved by computing all task's current deadline and then sort them in increasing order. If multiple tasks have identical deadlines, we use the task ID as a tie-breaker. In this way, we instantiate the time map computation in Fig. 7.1(b) with a concrete dynamic priority assignment policy.

The accumulative interference The iterative computation of time maps discussed above makes it difficult to reason about the schedulability of a task in a straightforward way, which relies on examining the accumulative increments of the virtual time within any physical period.

To bridge this gap, we introduce another ghost variable, $I_i\binom{t}{t+T_i}$, to represent the amount of *temporal interference* task τ_i receives in a period spanning $[t, t+T_i)$. In particular, a task is said to experience temporal interference from others during time slot [t, t+1) if this time slot is occupied by a higher-priority task. We demonstrate the accounting of the amount of interference in Fig. 7.5 and prove its connection with the virtual time map as follows.

```
1 int instrumented_dynamic_vtsched
    (){
2   t++;
3
4   int tid = N;
5   for (p = 0; p < N; p++) {
6     int i = PriQ_t(p);
7     if (tid == N){
8       ......
9     }else{
10      σ_i[t+1] = σ_i[t];
11      I_i(t/T_i*T_i, t/T_i*T_i+T_i)++;
12    }
13  }
14  return tid;
15 }
```

Figure 7.5: The computation of temporal interferences embedded in a virtual-time-based scheduler

Lemma 11 (The amount of interference decides the increments of virtual time)**.**

$$\forall i, k, \sigma_i((k+1)*T_i) - \sigma_i(k*T_i) = T_i - I_i\binom{k*T_i}{k*T_i+T_i}$$

Proof. The virtual time for an arbitrary task τ_i stagnates if and only if the temporal interference it experiences increments. Thus, at any time tick, either the virtual time or the interference increments by 1, and their total reaches T_i at the end of this current period. □

Recall that τ_i arrives at $k*T_i$ and the corresponding (implicit) deadline is $(k+1)*T_i$ for $k \geq 0$. Hence, $I_i\binom{k*T_i}{k*T_i+T_i}$ represents the amount of temporal interference that τ_i experiences from its arrival to the deadline. Following Lemma 11, the schedulability proof is reduced to showing that enlarging and shrinking a task set does not lead to more interference for each of its tasks.

A breakdown on the interference The above definition provides us a measurement to reason about the amount of interference a task experiences. However, to prove that the schedulability is preserved by the enlarging/shrinking operation, it's important to compare the interference incurred under two different task sets. In other words, the schedulability of a system is preserved if the amount of interference on each task does not increase. Thus, we need to further break down the interference and account for contributions from individual tasks.

Fig. 7.6 illustrates the temporal interference that τ_i can experience due to another task τ_j. Here, we only consider the periods of τ_j which have higher priority over τ_i, since the relative priority between tasks is time-dependent. In particular, t_s and t_e denote the start

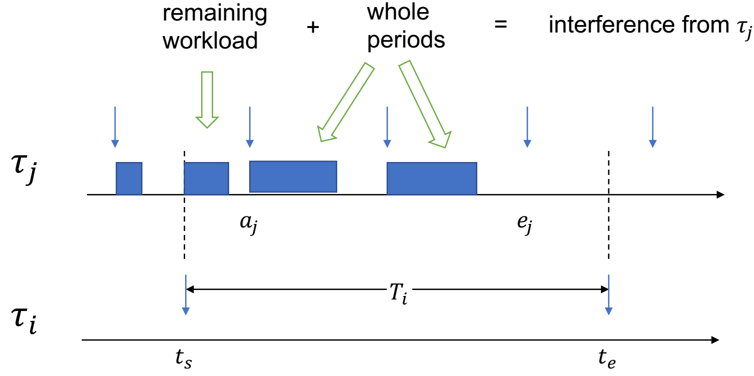


Figure 7.6: The amount of interference τ_j incurs on τ_i: a breakdown

and end of the current period for τ_i, a_j and e_j denote the first and last arrival of τ_j within the time interval $[t_s, t_s + T_i)$. As can be seen from the figure, there are two main categories of interference from τ_j as follows.

- Remaining workloads $wr_i(\tau_j, \sigma_j, t_s, t_e)$. It characterizes the case when t_s is in the middle of a period of τ_j, such that there might still be remaining quanta to be consumed by τ_j from t_s to the end of its current period. The value of such interference depends on the exact schedule of τ_j. Notice that if e_j is larger than t_e, or in other words, τ_j has a lower priority than τ_i within $[t_s, t_e)$, the amount of remaining workloads would be 0 since τ_j can no longer preempt τ_i.

- Whole periods of τ_j $ww_i(\tau_j, t_s, t_e)$. It represents the arrivals of τ_j that must finish no later than t_e, which also has a higher priority and can preempt τ_i. This is true for arrivals whose deadlines are earlier than t_e, and also for the arrival whose deadline equals t_e but the task ID j is less than i. Such interference only depends on task parameters, not the actual schedule of tasks.

The above two kinds of workloads are defined as follows.

Definition 12 (Remaining workloads $wr_i(\tau_j, \sigma_j, t_s, t_e)$)**.**

$$wr_i(\tau_j, \sigma_j, t_s, t_e) = \begin{cases} C_j - v_j(t_s) & v_j(t_s) < C_j \wedge (t_s \mod T_j \neq 0) \wedge (\lfloor \frac{t_s}{T_j} \rfloor T_j + T_j, j) < (t_e, i) \\ 0 & otherwise \end{cases}$$

where $v_j(t) = \sigma_j(t) - \sigma_j(\lfloor \frac{t_s}{T_j} \rfloor T_j)$

Definition 13 (Whole-period interference $ww_i(\tau_j, t_s, t_e)$)**.**

$$\begin{aligned} a_j = & \quad (t_s \mod T_j = 0) \ ? \ t_s : \lfloor \frac{t_s}{T_j} \rfloor T_j + T_j \\ e_j = & \quad ((t_e \mod T_j = 0) \wedge i < j) \ ? \ t_e - T_j : \lfloor \frac{t_e}{T_j} \rfloor T_j \end{aligned}$$

And

$$ww_i(\tau_j, t_s, t_e) = \begin{cases} \frac{e_j - a_j}{T_j} * C_j & t_s \le a_j < e_j \le t_e \\ 0 & otherwise \end{cases}$$

Note that τ_j can never execute after e_j as its priority becomes lower than τ_i (due to the deadlines, $t_e \le e_j + T_j$.) We use $w_i(\tau_j, \sigma_j, t_s, t_e)$ to represent the total workloads incurred by one task, i.e., $wr_i(\tau_j, \sigma_j, t_s, t_e) + ww_i(\tau_j, t_s, t_e)$.

We also use $WR_i(\Pi, t_s, t_e)$ and $WW_i(\Pi, t_s, t_e)$ to represent the total remaining workloads and whole period interferences from all other tasks in Π to τ_i. The sum of the two is denoted as $W_i(\Pi, t_s, t_e)$. We prove that it upper-bounds the amount of actual interference τ_i receives from other tasks.

Lemma 12 (Validity of the accumulative interference)**.** *For any task* $\tau_i \in \Pi$*, and any of its period* $[t_s, t_s + T_i)$

$$I_i \binom{t_s}{t_s + T_i} \le W_i(\Pi, t_s, t_s + T_i)$$

Proof. We consider an arbitrary moment t within the window of $[t_s, t_s + T_i)$. We prove that if there is interference to τ_i during $[t, t+1)$, the corresponding workload must also increase. In other words,

$$I_i \binom{t}{t+1} + W_i(\Pi, t+1, t_s + T_i) \le W_i(\Pi, t, t_s + T_i)$$

This is true because the task scheduled at time t, denoted τ_k, must have a higher priority than τ_i. This task must still have remaining quantum at t and this quantum value is decremented by 1 at $t + 1$. Thus, $1 + w_i(\tau_j, \sigma_j, t + 1, t_e) \le w_i(\tau_j, \sigma_j, t, t_e)$ must hold. Since workloads for all other tasks do not change, the above statement holds.

Given the above statement, we perform induction on the value of t and prove that

$$I_i\binom{t_s}{t_s+T_i} + W_i(\Pi, t_s+T_i, t_s+T_i) \leq W_i(\Pi, t_s, t_s+T_i)$$

Since there can be no interference within an empty window, this overall lemma is proved.

□

Notice that the above lemma uses an inequality to characterize the relationship between the amount of workloads and the amount of interference. This is because the accounting of interference is precise: the computation decides whether there is interference by looking at the exact schedule. However, the calculation of workloads only takes into account the accumulative values. When the schedulability of a task set is not known, one can only prove that the amount of workloads is an upper bound on the actual interference. This mismatch appears in the proof when $t+1$ represents the beginning of a new period for task τ_k. The exact interference at $[t, t+1)$ can at most be 1 time slot, yet $W_i(\Pi, t, t_s+T_i) - W_i(\Pi, t+1, t_s+T_i)$ can be larger then 1 if its schedulability is not known.

7.3.3 Enlarging a Task

This section details the reasoning that enlarging a task does not lead to more interference on itself or on other tasks. Without loss of generality, we assume that task τ_i is enlarged by a factor of k, while other tasks remain the same. We denote $\Pi' = \{\tau_i'\} \cup \{\tau_0, ..., \tau_{i-1}, \tau_{i+1}, ..., \tau_{N-1}\}$, where $\tau_i' = (kC_i, kT_i)$.

Characterization of the actual schedule. To account for the overlapping of workloads across periods in a comprehensive way, we further categorize the schedule of a time slot as follows.

```
Inductive sched_type: Type :=
| Sched_self
| Sched_interference
| Sched_leftover
| Sched_overlap
| Sched_longoverlap
| Sched_low
| Sched_idle.
```

Notice that the above categorization characterizes the nature of the schedule of one time slot from a task τ_i's perspective. Further, it is specified with regard to its current and next period, denoted as $[t_s, t_e)$ and $[t_e, t_e')$ respectively.

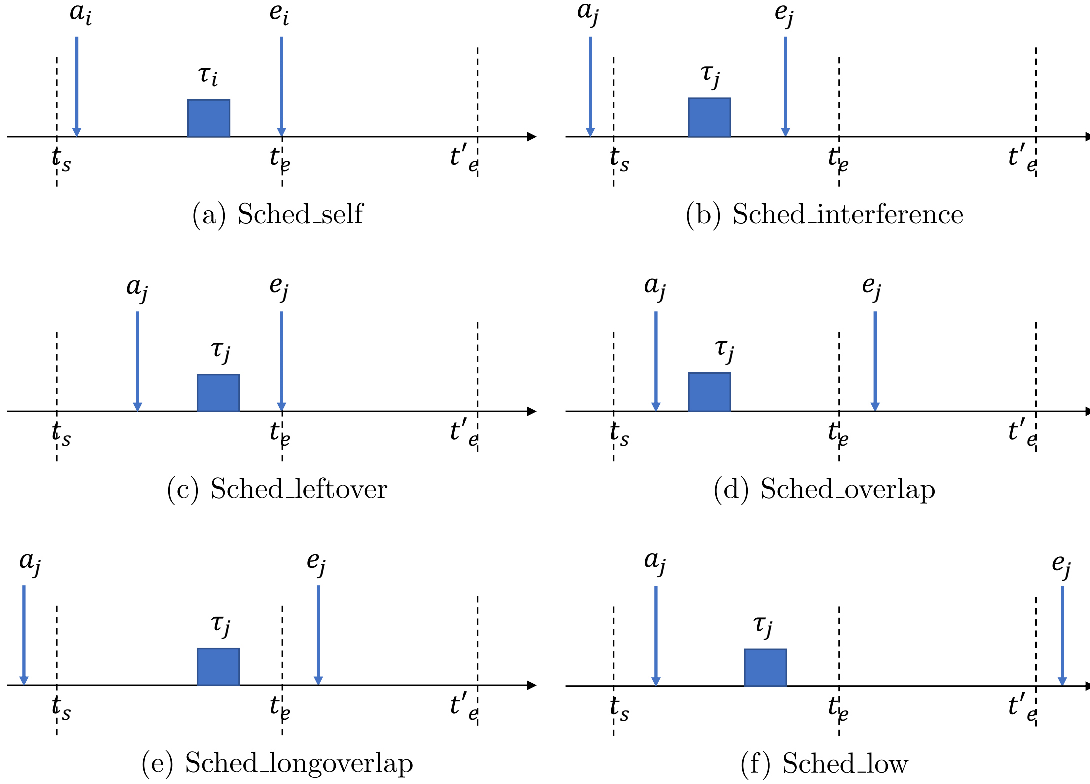
- Fig. 7.7(a) illustrates `Sched_self`. The shaded box labeled τ_i represents the schedule of τ_i for one time slot. The down arrows labeled a_i and e_i represents the beginning and end of the current period for τ_i. A schedule of τ_i is categorized as `Sched_self` if its period $[a_i, e_i)$ is enclosed by $[t_s, t_e)$.

- Fig. 7.7(b) illustrates `Sched_interference`. Here, $\tau_j \neq \tau_i$, and τ_j has higher priority over τ_i, either because $e_j < t_e$, or because $e_j = t_e \wedge j < i$. There is no constraint on the relation between a_j and t_s.

- Fig. 7.7(c) illustrates `Sched_leftover`. This is a special case when $e_j = t_e \wedge i < j$, such that τ_j has a lower priority than τ_i within $[a_j, t_e)$, but has higher priority than tasks whose deadlines are later than t_e, in particular, those labeled as `Sched_overlap`.

- Fig. 7.7(d) illustrates `Sched_overlap`. It characterizes the period of τ_j that starts within $[t_s, t_e)$ and ends within $[t_e, t_e')$. This case is particularly interesting because it influences the amount of interference "leaked" to the next period: more schedules before t_e means fewer remaining workloads for $[t_e, t_e')$. And the overall reasoning largely depends on reasoning about the allocated workloads of `Sched_overlap` before t_e, which is in turn influenced by others such as `Sched_self`, `Sched_interference`, and `Sched_leftover`.

- Fig. 7.7(e) illustrates `Sched_longoverlap`. This is the case when the current period of τ_j starts before t_s and must end within $[t_e, t_e')$.



(a) Sched_self (b) Sched_interference

(c) Sched_leftover (d) Sched_overlap

(e) Sched_longoverlap (f) Sched_low

Figure 7.7: Categorization of types of the schedule

- Fig. 7.7(f) illustrates `Sched_low`. This includes all tasks that have a lower priority than τ_i within the window $[t_e, t'_e)$. This is true when the deadline of τ_j is later than t'_e.

- `Sched_idle` represents the schedule of the idle task.

We formalize in Coq the computation of the type of a schedule and implement a function that counts the number of occurrences of each type within a time window. In particular, $\phi_i(\tau_j, ty, c_s, c_e, t_s, t_e, t'_e)$ counts the occurrences of τ_j within $[c_s, c_e)$, where it has a type of `ty` from the perspective of τ_i and the window specified by t_s, t_e, t'_e.

Definition 14 (Counting the schedule of τ_j that is of a given type ty)**.**

$$\phi_i(\tau_j, ty, c_s, c_e, t_s, t_e, t'_e) = |\{t \in [c_s, c_e) | \texttt{get_sched_type}(\tau_i, \tau_j, t, t_s, t_e, t'_e) = \texttt{ty}\}|$$

We also formalize in Coq a function that counts the occurrences of all tasks having a particular type.

Definition 15 (Counting the schedule of all tasks that is of a given type ty)**.**

$$\Phi_i(ty, c_s, c_e, t_s, t_e, t'_e) = \sum_j \phi_i(\tau_j, ty, c_s, c_e, t_s, t_e, t'_e)$$

Lemma 13 (This categorization is comprehensive)**.**

$$\forall c_s, c_e, t_s, t_e, t'_e, c_s \leq c_e \implies \sum_{ty} \Phi_i(ty, c_s, c_e, t_s, t_e, t'_e) = c_e - c_s$$

Proof. By construction, at any time instant, if there is no real-time task scheduled, the type would be `Sched_idle`. Otherwise, the type of the scheduled task must fall into one of the above categories in Fig. 7.7. □

The workloads and the actual schedule. We further define two additional measurements as follows.

Definition 16 (Overlapping workloads count all other task periods that overlap with τ_i)**.**

$$is_overlap_i(\tau_j, t_s, t_e, t'_e) = \begin{cases} 1 & t_s \leq a_j < t_e < e_j \land (e_j, j) < (t'_e, i) \\ 0 & otherwise \end{cases}$$

where $a_j = \lfloor \frac{t_e}{T_j} \rfloor T_j \land e_j = a_j + T_j$

Considering all other tasks, we have

$$WO_i(\Pi, t_s, t_e, t'_e) = \sum_{\tau_j \in \Pi} is_overlap_i(\tau_j, t_s, t_e, t'_e) * C_j$$

Definition 17 (Leftover workloads count all other tasks whose schedule must be of type Sched_leftover)**.**

$$is_leftover_i(\tau_j, t_s, t_e) = \begin{cases} 1 & t_s \leq t_e - T_j \land t_e \mod T_j = 0 \land i < j \\ 0 & otherwise \end{cases}$$

And

$$WL_i(\Pi, t_s, t_e) = \sum_{\tau_j \in \Pi} is_leftover_i(\tau_j, t_s, t_e) * C_j$$

We prove in Coq the following relationships between the value of various workloads and the counting of the real schedule.

Lemma 14 (The leftover workloads is an upperbound on the real schedule)**.**

$$\forall i, t_s, t_e, t'_e, \Phi_i(\texttt{Sched_leftover}, t_s, t_e, t_s, t_e, t'_e) \leq WL_i(\Pi, t_s, t_e)$$

Proof. The proof is achieved by showing that each task τ_j satisfies the following constraint.

$$\phi_i(\tau_j, \texttt{Sched_leftover}, t_s, t_e, t_s, t_e, t'_e) \leq is_leftover_i(\tau_j, t_s, t_e) * C_j$$

1. If $is_leftover_i(\tau_j, t_s, t_e) = 0$, we prove that there is no schedule of τ_j satisfying type `Sched_leftover`, since the condition as depicted in Fig. 7.7(c) does not hold.

2. If $is_leftover_i(\tau_j, t_s, t_e) = 1$, we prove that for any time instant $t \in [t_e - T_j, t_e)$, the following is true.

$$\phi_i(\tau_j, \texttt{Sched_leftover}, t_s, t, t_s, t_e, t'_e) = min(C_j, \sigma_j(t) - \sigma_j(t_s))$$

   This is because if τ_j is scheduled at t, both its virtual time and the accounting increments by 1. Otherwise, neither of the two increments. We also prove that the schedule of τ_j is subject to its budget constraint, such that the number of actual schedules cannot exceed C_j while the amount of available virtual time could be larger. When $t = t_e$, we prove the above property.

Combining the contribution of all tasks together, we prove this lemma. Notice that equality holds when the task set is proved to be schedulable. □

The proof of the above lemma demonstrates how we connect the accumulative value of a particular kind of workload to the actual schedule under EDF. It shows that the virtual timeline abstraction faithfully tracks the actual schedule of a task, makes it easy to carry out

the budget constraint in the reasoning, but more importantly, connects the schedulability with the concrete schedule in a straightforward way. It's easy to prove that task τ_j receives its full budget if this task is schedulable (Def. 2). This proof also provides some insight into the seemingly unusual inequality introduced in Lemma 12.

Similarly, we prove the following lemmas regarding the various workloads and the real schedule. Most of their proofs follow the same approach as in Lemma 14, and proof details are omitted in this section.

Lemma 15 (The schedule of a task is bounded by its budget constraint)**.**

$$\forall i, t, t'_e, t \mod T_i = 0 \implies \Phi_i(\texttt{Sched_self}, t, t+T_i, t, t+T_i, t'_e) \leq C_i$$

Lemma 16 (The actual amount of the interference is bounded by the remaining and whole-period workloads)**.**

$$\forall i, t, \Phi_i(\texttt{Sched_interference}, t, t+T_i, t, t+T_i, t+2T_i) \leq W_i(\Pi, t, t+T_i)$$

Lemma 17 (The overlapping workloads leak to the next period and becomes remaining workloads)**.**

$$\forall t_s, t_e, t'_e, WO_i(\Pi, t_s, t_e, t'_e) - WR_i(t_e, t'_e) = \Phi_i(\texttt{Sched_overlap}, t_s, t_e, t_s, t_e, t'_e)$$

Interference on τ_i itself Now we examine temporal interferences incurred by other tasks on τ'_i. Since its parameters are enlarged, a period now spans from t_s to $t_s + kT_i$, as shown in Fig. 7.8. In this case, the total whole-period interference experienced by τ'_i during $[t_s, t_s + kT_i)$ equals that experienced by τ_i (i.e., before the enlargement) during the same interval $[t_s, t_s + kT_i)$, illustrated as shaded rectangles in the figure. This is because those tasks that could preempt τ_i during $[t_s, t_s + kT_i)$ will still have higher-priorities than τ'_i after the period is enlarged. Thus, we focus on the remaining workloads from others.

We reason about the amount of workloads by induction on the index of the period. We assume that workloads in period k are properly bounded, and prove that this leaves enough time slots to accommodate workloads that overlap with the next period so that the

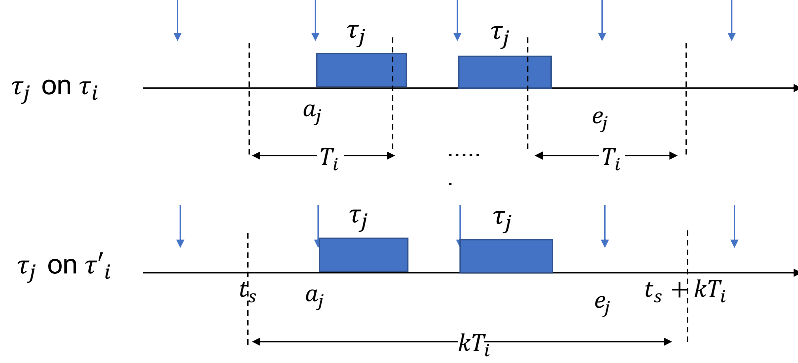


Figure 7.8: Whole-period interference from τ_j to τ_i before and after the enlargement

interference incurred on the next period is also sufficiently bounded.

Lemma 18 (Enlarging τ_i does not increase the remaining workloads incured on it)**.**

$$\forall m \geq 0, WR_i(\Pi, m * kT_i, (m+1) * kT_i) \geq$$
$$WR_i(\Pi', m * kT_i, (m+1) * kT_i)$$

Proof. As discussed above, we prove this lemma by induction on the period. The base case corresponds to $[0, kT_i)$, which is trivial because the remaining workloads are 0 in the schedule of both Π and Π'.

For the induction case, we assume this lemma holds for an arbitrary m, and we prove it for $m + 1$. We define $t_s = mkT_i$, $t_e = t_s + kT_i$ and $t'_e = t_e + kT_i = t_s + 2kT_i$. Notice that one period of τ'_i (e.g. $[t_s, t_e)$) spans k periods of τ_i.

We perform case analysis on whether there are idle time slots in the schedule of Π' within $[t_s, t_e)$.

1. Assume the schedule is busy through the whole interval $[t_s, t_e)$. As shown in Fig. 7.9, the top half depicts the schedule of Π and the bottom half depicts that of Π'. We observe that the overlapping workloads are equal in both cases since it only depends on task parameters instead of the actual schedule. Then the question is how much of its execution is in $[t_s, t_e)$ and how much is left over to $[t_e, t'_e)$ and contributes to the remaining interference for the $(m+1)^{th}$ period.

   Intuitively, since τ'_i suffers less total interference in $[t_s, t_e)$, there should be more

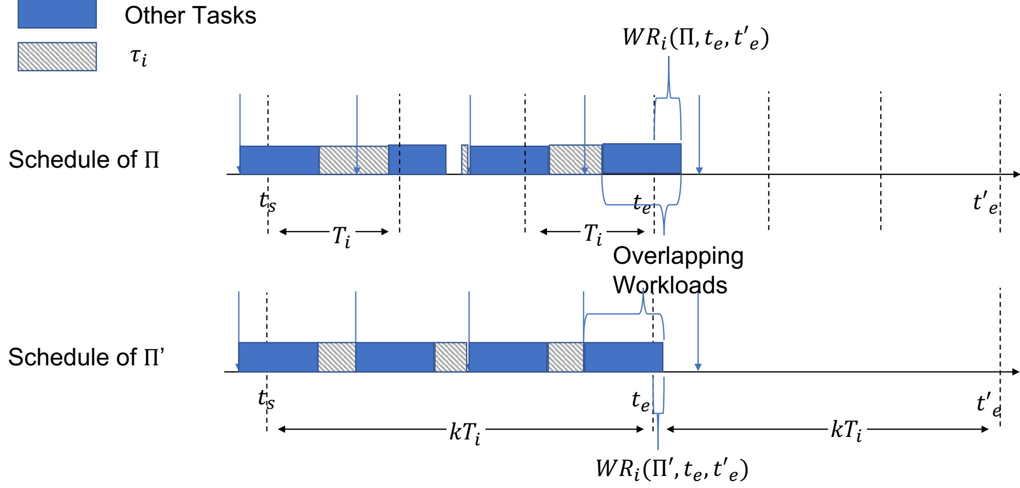


Figure 7.9: When the schedule of Π' is busy, it leaks less remaining workload to the next period compared to the schedule of Π

time slots devoted to these overlapping workloads, and hence lead to less remaining workload for the next period.

- For the schedule of Π', we prove that

$$WO_i(\Pi', t_s, t_e, t'_e) - WR_i(\Pi', t_e, t'_e) \geq kT_i - kC_i - W_i(\Pi', t_s, t_e) - WL_i(\Pi', t_s, t_e)$$

This is done by proving that since kT_i is the longest period in Π', there can be no schedule of type `Sched_longoverlap` or `Sched_low`. Thus, the whole period of length kT_i is occupied by `Sched_self`, `Sched_interference`, `Sched_leftover`, and `Sched_overlap`. By applying Lemma 14, 15, 16 and 17, the above property is proved. The intuition is that the first three types of schedules are all bounded by their corresponding workloads, thus the last one is guaranteed a lower bound of time slots.

- For the schedule of Π, since it is known to be schedulable, we prove that

$$WO_i(\Pi, t_s, t_e, t'_e) - WR_i(\Pi, t_e, t'_e) \leq kT_i - kC_i - W_i(\Pi, t_s, t_e) - WL_i(\Pi, t_s, t_e)$$

This is true because the amount of workloads equals the accounting on the actual schedule when this task set is schedulable. However, recall that there might be

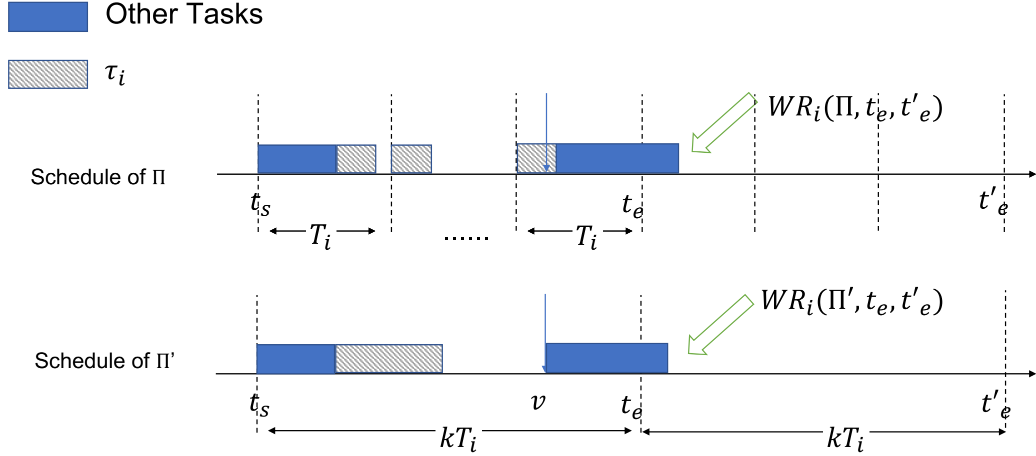


Figure 7.10: When the schedule of Π' is not busy, it leaks less remaining workload to the next period compared to the schedule of Π

`Sched_idle` in this case, thus the total count of `Sched_self`, `Sched_interference`, `Sched_leftover`, and `Sched_overlap` is less than or equal to kT_i.

We also prove that the overlapping and leftover workloads only depend on task parameters, such that $WO_i(\Pi, t_s, t_e, t'_e) = WO_i(\Pi', t_s, t_e, t'_e)$ and $WL_i(\Pi, t_s, t_e) = WL_i(\Pi', t_s, t_e)$. Combining everything together with the induction hypothesis that $W_i(\Pi, t_s, t_e) \geq W_i(\Pi', t_s, t_e)$, we prove $WR_i(\Pi, t_e, t'_e) \geq WR_i(\Pi', t_e, t'_e)$.

2. Assume there is at least one idle slot in the schedule of Π', and there exists $v \in (t_s, t_e]$ such that $[v-1, v)$ is the last idle slot. In this case, the schedule within $[v, t_e)$ is busy as shown in Fig. 7.10.

   Intuitively, the amount of workloads leaking to the next period is determined by the overlapping workloads within $[v, t_e)$. Compared to the schedule of Π, the schedule of Π' allocates as many time slots as possible to accommodate those overlapping workloads and should result in the least amount of remaining workload for the next period. We prove it in Coq as follows.

   - For the schedule of Π', we prove that

$$WO_i(\Pi', t_s, t_e, t'_e) - WR_i(\Pi', t_e, t'_e) \geq t_e - v - WW_i(\Pi', v, t_e) - WL_i(\Pi', v, t_e)$$

This is true because the last busy window, $[v, t_e)$, only contains `Sched_interference`, `Sched_leftover`, and `Sched_overlap`. While the first two types are bounded by their workload (Lemma 16 and 14), the last one equals its workload (Lemma 17).

- For the schedule of Π, we prove that

$$WO_i(\Pi, t_s, t_e, t'_e) - WR_i(\Pi, t_e, t'_e) \leq t_e - v - WW_i(\Pi, v, t_e) - WR_i(\Pi, v, t_e) - WL_i(\Pi, v, t_e)$$

  The proof follows the same reasoning as in the case when the entire window is busy. Since there might be `Sched_idle` and extra schedules of τ_i within $[v, t_e)$, time slots allocated to `Sched_overlap` are subject to an upper bound.

Since the overlapping and leftover workload stays the same, and also the whole-period workload is identical in both cases, we prove $WR_i(\Pi', t_e, t'_e) \leq WR_i(\Pi, t_e, t'_e)$

This concludes the proof by induction. □

Given that the whole-period interference does not change, we know the total interference experienced by τ_i does not increase after it is enlarged. Thus, τ'_i must also be schedulable.

Interference on other tasks Now we consider the interferences incurred on an arbitrary task τ_j, where $\tau_j \neq \tau_i$. Intuitively, τ_i's priority is lowered as its deadline is extended. Starting from a burst of a busy schedule, i.e. there is no remaining workload, interferences incurred on τ_j cannot increase.

Before diving into the proof details, we define an extended version of the whole period workload, so that all whole periods within a window is counted regardless of the relative priority.

Definition 18 (Extended whole-period interference $wew_i(\tau_j, t_s, t_e)$)**.**

$$\begin{aligned} a_j = & \quad (t_s \mod T_j = 0) \ ? \ t_s : \lfloor \frac{t_s}{T_j} \rfloor T_j + T_j \\ e_j = & \quad \lfloor \frac{t_e}{T_j} \rfloor T_j \end{aligned}$$

And

$$wew_i(\tau_j, t_s, t_e) = \begin{cases} \dfrac{e_j - a_j}{T_j} * C_j & t_s \le a_j < e_j \le t_e \\ 0 & otherwise \end{cases}$$

And we define

$$WEW_i(\Pi, t_s, t_e) = \sum_{\tau_j \in \Pi} wew_i(\tau_j, t_s, t_e)$$

Lemma 19 (Enlarging τ_i does not lead to more temporal interference on other tasks)**.**

$$\forall m \ge 0, W_j(\Pi, mT_j, mT_j + T_j) \ge$$
$$W_j(\Pi', mT_j, mT_j + T_j)$$

Proof. Consider an arbitrary period of τ_j: $[mT_j, mT_j+T_j)$. We denote $t_s = mT_j, t_e = t_s+T_j$. We define a function `start_of_burst(ts, id, deadline)`, which searches for the longest window $[v, ts)$ such that any schedule within this window has a priority higher than or equal to $(deadline, id)$. This can be viewed as a generalized version for searching a busy window since lower-priority slots are also ruled out in addition to idle slots. The result of this function must be within the range $[0, ts]$. The return value is ts if The slot at $[ts-1, ts)$ is idle or accommodates a task whose priority is lower.

We apply this function on the schedule of Π' and get $v =$ `start_of_burst`(t_s, t_e, j). By definition, all time slots within $[v, t_s)$ are busy and have a priority higher than (t_e, j). On the other hand, The slot at $[v-1, v)$ is either idle or has a lower priority.

- For the schedule of Π', we prove that

  $$WO_j(\Pi', v, t_s, t_e) - WR_j(\Pi', t_s, t_e) + WEW_j(\Pi', v, t_s) \ge t_s - v$$

  This is because, in such a busy schedule, $WO_j(\Pi', v, t_s, t_e) - WR_j(\Pi', t_s, t_e)$ equals the count of `Sched_overlap`, while $WEW_j(\Pi', v, t_s)$ is no less than the count of other types that are possible within $[v, t_s)$ (i.e. `Sched_interference` and `Sched_leftover`).

- For the schedule of Π, we prove that

  $$WO_j(\Pi, v, t_s, t_e) - WR_j(\Pi, t_s, t_e) + WEW_j(\Pi, v, t_s) \le t_s - v$$

The task set Π is schedulable and its schedule within $[v, t_s)$ may contain tasks that have lower priority. Thus, schedules corresponding to the above workload must be less than or equal to $t_s - v$.

- Comparing these two schedules, we prove that

$$WO_j(\Pi', v, t_s, t_e) + WEW_j(\Pi', v, t_s) + WW_j(\Pi', t_s, t_e) \leq$$
$$WO_j(\Pi, v, t_s, t_e) + WEW_j(\Pi, v, t_s) + WW_j(\Pi, t_s, t_e)$$

This actually compares the overall whole-period workload within $[v, t_e)$, while $WEW_j(\Pi', v, t_s)$ counts the portion entirely within $[v, t_s)$, $WW_j(\Pi', t_s, t_e)$ counts those entirely within $[t_s, t_e)$, and $WO_j(\Pi', v, t_s, t_e)$ counts those spanning across these two windows. Since enlarging a task can only result in less or equal whole-period workload (it might cause a period to fall outside the window), the above property holds.

Combining everything together,

$$WW_j(\Pi, t_s, t_e) + WR_j(\Pi, t_s, t_e) - WW_j(\Pi', t_s, t_e) - WR_j(\Pi', t_s, t_e) \geq 0$$

Thus, this lemma holds. □

This concludes our reasoning for enlarging a task. Since the task itself and all other tasks cannot experience more workload, the enlarged task set must also be schedulable.

7.3.4 Shrinking a Task Set

This section details the reasoning that shrinking a task set preserves its schedulability. In particular, we define the following predicate.

Definition 19 (The task set Π is the result of shrinking Π' by k)**.**

$$\texttt{taskset_k_times}\ k\ \Pi\ \Pi' \equiv \forall \tau_i \in \Pi, \tau'_i \in \Pi', T'_i = kT_i \land C'_i = kC_i$$

Throughout this section, we use σ_i to denote the time map of task τ_i in the schedule of Π. Similarly, we use σ'_i to denote the time map of τ'_i in the schedule of Π'.

Intuitively, we observe that the schedule of Π' must consist of smaller chunks of size k. In other words, within any window $[pk, pk+k)$, all time slots schedule a common task since all tasks' budgets and periods are a multiple of k.

We define the following predicate.

Definition 20 (All task parameters are a multiple of k)**.**

$$\texttt{taskset_k_divisor}\ k\ \Pi' \equiv \forall\, \tau'_i \in \Pi', T'_i \mod k = 0 \wedge C'_i \mod k = 0$$

We then prove the following property about the schedule of Π'.

Lemma 20 (The schedule of Π' consists of smaller chunks of size k)**.** *Assume that* `taskset_k_divisor` $k\ \Pi'$ *holds. At any time t, for any task τ'_i, one of the following must be true. (i) τ'_i is blocked by another task at time $\lfloor\frac{t}{k}\rfloor k$, and $\sigma'_i(t) = \sigma'_i(\lfloor\frac{t}{k}\rfloor k)$. (ii) τ'_i is not blocked by any task at time $\lfloor\frac{t}{k}\rfloor k$, and $\sigma'_i(t) = \sigma'_i(\lfloor\frac{t}{k}\rfloor k) + t \mod k$. Notice that case (ii) does not mean τ'_i must be the running task. It is possible that the virtual time of τ'_i increases while the task itself has exhausted its budget in the current period.*

Proof. We strengthen the induction hypothesis with another proof goal:

$$\forall t, t \mod k = 0 \implies \sigma'_i(t) \mod k = 0$$

We prove by induction on the time t.

In the base case, t = 0, $\sigma'_i(0) = 0$. It's easy to see the above properties hold.

Assume that the above properties hold for t. Consider the case with t + 1.

- If $(t+1) \mod k = 0$, we prove that the value of virtual time must also be a multiple of k. This requires an inspection of $\sigma'_i(t)$.
  - If τ'_i is blocked at $\lfloor\frac{t}{k}\rfloor k = t+1-k$, it will also be blocked at time t since the ordering in the priority queue does not change. Thus,

    $$\sigma'_i(t+1) = \sigma'_i(t) = \sigma'_i(t+1-k)$$

    which is indeed a multiple of k.

- Otherwise, if τ'_i is not blocked at $t+1-k$, it is not blocked at t, either. Thus,

$$\begin{aligned}\sigma'_i(t+1) &= \sigma'_i(t)+1\\ &= \sigma'_i(t+1-k)+(t \mod k)+1\\ &= \sigma'_i(t+1-k)+(k-1)+1\\ &= \sigma'_i(t+1-k)+k\end{aligned}$$

which is also a multiple of k.

- If $(t+1) \mod k \neq 0$, we prove that the value of virtual time at this moment is valid. Here, we know that $t \mod k \neq 0$ also holds, and $\lfloor\frac{t+1}{k}\rfloor k = \lfloor\frac{t}{k}\rfloor k$.

  We do case analysis on whether τ'_i is blocked at time $\lfloor\frac{t}{k}\rfloor k$.

  - If τ'_i is blocked at $\lfloor\frac{t}{k}\rfloor k$, it will also be blocked at time t. Thus,

$$\begin{aligned}\sigma'_i(t+1) &= \sigma'_i(t)\\ &= \sigma'_i(\lfloor\frac{t}{k}\rfloor k)\end{aligned}$$

  - Otherwise, if τ'_i is not blocked at $\lfloor\frac{t}{k}\rfloor k$, it is not blocked at t, either. Thus,

$$\begin{aligned}\sigma'_i(t+1) &= \sigma'_i(t)+1\\ &= \sigma'_i(\lfloor\frac{t}{k}\rfloor k)+(t \mod k)+1\\ &= \sigma'_i(\lfloor\frac{t}{k}\rfloor k)+((t+1) \mod k)\\ &= \sigma'_i(\lfloor\frac{t+1}{k}\rfloor k)+((t+1) \mod k)\end{aligned}$$

  The value of $\sigma'_i(t+1)$ is valid in both cases.

Thus, this lemma holds for any task at any time. □

The intuition of Lemma 20 is that the schedule of Π' must be in units of k consecutive time slots. Since all periods and budgets are multiples of k, and that we already impose the task ID as a tie-breaker, the schedule within any window of $[pk, pk+k)$ must be consistent. As a corollary, the value of time map at the boundary of any window must be a multiple of k, which is exactly the strengthened goal in the above proof.

We then connect the schedule of Π' with that of Π. Intuitively, they follow the same schedule but with different granularity. Every k consecutive time slots in the schedule of Π' correspond to one time slot in that of Π.

Lemma 21 (The schedule of Π' maps to that of Π)**.** *Consider two task sets satisfying* `taskset_k_times` $k\ \Pi\ \Pi'$*. For any arbitrary task τ_i and time t, the following holds*

$$\sigma_i(\lfloor\frac{t}{k}\rfloor) = \lfloor\frac{\sigma_i'(t)}{k}\rfloor$$

Proof. We prove by induction on the time t. In the base case, t = 0, $\sigma_i'(0) = 0 \wedge \sigma_i(0) = 0$. It's easy to see this lemma holds.

Assume that the above properties hold for t. Consider the case with t+1. We inspect the virtual time of an arbitrary task τ_i.

- If $(t+1) \mod k = 0$, we know $\lfloor\frac{t+1}{k}\rfloor = \lfloor\frac{t}{k}\rfloor + 1$, which corresponds to one step in the schedule of Π.
  - if τ_i is blocked at time $\lfloor\frac{t}{k}\rfloor$, we prove that τ_i' is also blocked through the k consecutive slots starting from $\lfloor\frac{t}{k}\rfloor k$. Thus,

$$\begin{aligned}\sigma_i(\lfloor\frac{t+1}{k}\rfloor) &= \sigma_i(\lfloor\frac{t}{k}\rfloor)\\ &= \lfloor\frac{\sigma_i'(t)}{k}\rfloor\\ &= \lfloor\frac{\sigma_i'(t+1)}{k}\rfloor\end{aligned}$$

  - if τ_i is not blocked at time $\lfloor\frac{t}{k}\rfloor$, we prove that τ_i' is not blocked either through

the k consecutive slots starting from $\lfloor \frac{t}{k} \rfloor k$. Thus,

$$\begin{aligned}
\sigma_i(\lfloor \frac{t+1}{k} \rfloor) &= \sigma_i(\lfloor \frac{t}{k} \rfloor) + 1 \\
&= \lfloor \frac{\sigma'_i(t)}{k} \rfloor + 1 \\
&= \lfloor \frac{\sigma'_i(t+1-k) + (t \mod k)}{k} \rfloor + 1 \\
&= \lfloor \frac{\sigma'_i(t+1-k) + k}{k} \rfloor \\
&= \lfloor \frac{\sigma'_i(t+1)}{k} \rfloor
\end{aligned}$$

- If $(t+1) \mod k \neq 0$, we know that $t \mod k \neq 0$ also holds, and $\lfloor \frac{t+1}{k} \rfloor = \lfloor \frac{t}{k} \rfloor$. We further prove that whether τ'_i is blocked or not at time $\lfloor \frac{t}{k} \rfloor k$, the following holds.

$$0 \leq \sigma'_i(t+1) - \sigma'_i(\lfloor \frac{t}{k} \rfloor k) \leq (t+1) \mod k < k$$
$$0 \leq \sigma'_i(t) - \sigma'_i(\lfloor \frac{t}{k} \rfloor k) \leq t \mod k < k$$

  Thus,

$$\begin{aligned}
\sigma_i(\lfloor \frac{t+1}{k} \rfloor) &= \sigma_i(\lfloor \frac{t}{k} \rfloor) \\
&= \lfloor \frac{\sigma'_i(t)}{k} \rfloor \\
&= \lfloor \frac{\sigma'_i(\lfloor \frac{t}{k} \rfloor k)}{k} \rfloor \\
&= \lfloor \frac{\sigma'_i(t+1)}{k} \rfloor
\end{aligned}$$

This lemma holds. □

Finally, we prove in Coq that shrinking a task set preserves its schedulability.

Lemma 22 (Shrinking a task set preserves its schedulability)**.** *For two task sets* Π *and* Π'*, if* `taskset_k_times` k Π Π' *holds, and* Π' *is schedulable, then* Π *must also be schedulable.*

Proof. Without loss of generality, we examine an arbitrary task $\tau_i \in \Pi$. We pick an arbitrary

period $[pT_i, pT_i + T_i)$, and denote $t_s = pT_i$, $t_e = pT_i + T_i$. The schedulability proof of τ_i in this period requires a comparison between $\sigma_i(t_e) - \sigma_i(t_s)$ and C_i.

Firstly, we apply Lemma 21 and prove that

$$\sigma_i(t_e) - \sigma_i(t_s) = \sigma_i'(kt_e)/k - \sigma_i'(kt_s)/k$$

Since $\sigma_i'(kt_e) \mod k = 0$ (Lemma 20), we know

$$\sigma_i'(kt_e)/k - \sigma_i'(kt_s)/k = (\sigma_i'(kt_e) - \sigma_i'(kt_s))/k$$

And the schedulability of Π' implies that

$$\sigma_i'(kt_e) - \sigma_i'(kt_s) \geq kC_i$$

Thus,

$$\sigma_i(t_e) - \sigma_i(t_s) \geq C_i$$

Π must also be schedulable. □

To this end, we have proved that both the enlarging and shrinking operations preserve the schedulability of the system. They complete the proof in Thm. 2, such that a task set is schedulable if its total utilization does not exceed 100%.

Chapter 8

Case Study: CertiKOS with Earliest-Deadline-First Scheduling

In this chapter, we discuss how we implement and verify an EDF scheduler and connects it with the virtual timeline abstraction.

8.1 The Concrete Scheduler Implementation

Fig. 8.1 depicts the concrete implementation of an EDF scheduler. It first refills budgets for each task, then iterates over all tasks and records the one that has the nearest deadline and also remaining budget. In the end, if such a task is found, the scheduler will deduct one time slot from its remaining budget.

Notice that this implementation relies on the following two implicit assumptions.

1. Task arrivals exhibit regular periods. In this way, the scheduler tracks the start of a new period and calculates the deadline of a task in a straightforward way.

2. As discussed in Sec. 7.3.2, we use the task ID as a tie-breaker when two tasks have a common deadline. The scheduler in Fig. 8.1 iterates over all tasks in increasing order of the task ID, thus implicitly carrying out this precedence rule.

Also notice that this linear-time search can be improved by adopting a more advanced data structure, such as a balanced binary search tree, that enables a more efficient lookup.

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

  int pid = N;
  int dmin = 0;
  for(int i = 0; i < N; i++){
    if (quanta[i] > 0){
      int d = t/T_i*T_i + T_i;
      if (pid==N || d<dmin){
        dmin = d;
        pid = i;
      }
    }
  }

  if (pid < N){
    quanta[pid]--;
  }
  return pid;
}
```

Figure 8.1: The C implementation of an EDF scheduler

However, incorporating such an improvement only relies on a proof of algorithmic equivalence, which is not the focus of this thesis. Further, safety-critical real-time systems usually only accommodate a moderate number of tasks per single processor, rendering the performance optimization less significant.

8.2 Refinement with the Virtual-Time-Based Scheduler

To reason about the concrete scheduler, we need to prove its contextual equivalence with an intermediate abstraction, whose structure is closer to that of the virtual-time-based scheduler. In particular, this intermediate abstraction iterates over a priority queue instead of over all task IDs consecutively.

Formalization of the priority queue Fig. 8.2 depicts the Coq formalization of the priority queue at instant t. In particular, it is a list of `priEntry`, where each element contains the current deadline and ID for a task. Function `real_priqueue_aux` initializes such a list containing all tasks in the system, but in the order of task IDs.

The module `prisort` is an instantiation of Coq Sort with the element type `priEntry` and comparison function `priEntry_le`. Thus, `prisort.sort` sorts the aforementioned list such that all elements are in ascending order with respect to `priEntry_le`.

An intermediate abstraction The intermediate abstraction follows the structure of Fig. 7.1(a). Similar to Fig. 8.1, it first refills budgets for each task, then selects the ready

```
Inductive priEntry: Type :=
| PriEntry: Z (* deadline *) → Z (* task ID *) → priEntry.

Function priEntry_le e1 e2 :=
  match e1, e2 with
  | PriEntry d1 id1, PriEntry d2 id2 ⇒
    if (zlt d1 d2) then true
    else (if zeq d1 d2 then
            (if zle id1 id2 then true else false) else false)
  end.

(* A list of entries, from high to low priority  *)
Definition priQueue := list priEntry.

Fixpoint real_priqueue_aux (N: nat) (t: Z) (conf: PrioConfigPool):=
  match N with
  | O ⇒ nil
  | S n ⇒
    match (ZMap.get (Z.of_nat n) conf) with
    | mkPrioConfigValid T C ⇒
      PriEntry (t / T * T + T) (Z.of_nat n) :: (real_priqueue_aux n t conf)
    end
  end.

Definition real_priqueue (N: nat) (t: Z) (conf: PrioConfigPool) :=
  prisort.sort (real_priqueue_aux N t conf).
```

Figure 8.2: Coq formalization of the priority queue

task with the nearest deadline. However, unlike the C implementation, it iterates over a priority queue. The formalization of budget refills is the same as the function `tick_quanta` explained in Fig. 4.2. Thus, we omit details in this section.

Figure. 8.3 depicts the iteration used in the intermediate abstraction. It is defined as a recursive function, which searches over the priority queue and returns the first task whose remaining budget is positive. At each invocation, the parameter `queue` is ordered by $(deadline, id)$ for each task, where the deadline for τ_i is defined as $\lfloor \frac{t}{T_i} \rfloor T_i + T_i$.

We prove that invoking `highest_pos_quantum` is contextually equivalent to the iteration over task IDs as defined in Fig. 8.1. The intuition is as follows.

- If `highest_pos_quantum` returns `idle`, all tasks' quantum value must be 0. In this

```
1  Fixpoint highest_pos_quantum_queue_aux (queue: priQueue) (idle: Z)
2         (quantums: QuantumPool) :=
3    match queue with
4    | nil ⇒  idle
5    | (PriEntry deadline id) :: l ⇒
6      if (zlt 0 (ZMap.get id quantums)) then id else
7        highest_pos_quantum_queue_aux l idle quantums
8    end.
9
10 Definition highest_pos_quantum (queue: priQueue) (quantums: QuantumPool) :=
11   highest_pos_quantum_queue_aux queue N quantums.
```

Figure 8.3: The iteration over a priority queue, used in the intermediate abstraction

case, `sched` also schedules the idle task.

- If `highest_pos_quantum` schedules task τ_p, all other tasks either has 0 remaining budget or is ready but has a lower priority than p. In other words, τ_p has the highest priority among ready tasks at this moment. In this case, `sched` also schedules task τ_p because `pid` would eventually be assigned the value of `p` and never updates again.

Proof details are omitted in this section since they mainly involve algorithmic equivalence.

Refinement into the virutal-time-based scheduler Finally, we prove the equivalence between the intermediate abstraction and the virtual-time-based scheduler (similar to Fig. 7.1(b)). For simplicity, we use `abs_sched` to denote the intermediate abstraction and use `vt_sched` to denote the virtual-time-based scheduler.

We follow the same approach as in Sec. 4.3 and prove the equivalence relation as in Def. 4. Then the equivalence between the two schedulers is straightforward. Since they iterate over the same priority queue, at each step, they either skip this task or schedule the same task. In fact, this largely resembles the existing proof in Sec. 4.3, with the exception that the fixed-priority scheduling and earliest-deadline-first scheduling iterates over all tasks in different orders. This concludes the connection between the concrete scheduler implementation with the virtual timeline abstraction under the EDF scheduling.

The connection in this chapter largely follows the same approach as in Chpt. 4, other than the gap between the concrete scheduler and its intermediate abstraction. This again

demonstrates the generality of the virtual timeline abstraction: the formalization of virtual time is a common structure among preemptive schedulers. It faithfully captures the preemption relation between tasks (even in the dynamic setting), making the connection with the concrete scheduler straightforward.

Chapter 9

Related Work

9.1 Mechanized Schedulability Analysis

The first mechanized proof about schedulability is achieved by Wilding et al. [30]. They use the Nqthm theorem prover to formalize the EDF scheduling algorithm and prove its optimality. They motivate the need for a machine-checked proof by pointing out that the original informal proof by Liu and Layland [22] contains errors, even though the proof goal is correct. Dutertre [31] 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. [32, 33] formalize the priority inheritance protocol using Isabelle/HOL and prove that it guarantees a finite bound on the duration of priority inversion. They also leverage the insight obtained during the proof development to guide the implementation of such a protocol. The PROSA [34] project demonstrates readable mechanized proofs of various schedulability analyses written in Coq. It defines an abstract model of real-time tasks and real-time scheduling algorithms and formalizes and proves correct a response time analysis for tasks. All of the above work is conducted on an abstract model of a real-time system.

Guo et al. [35] connect PROSA with the real-time extension of CertiKOS described in Sec. 4.2, and obtain 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 earliest-deadline-first scheduling and partitioned scheduling, which involves more subtle isolation issues.

9.2 Verification of OS kernels

Both Klein [36] and Zhao et al. [37] have conducted a survey and provided a full overview of the verification efforts about operating systems. Among all the OS verification works, we place our emphasis mainly on those done at the C or assembly code level.

SeL4 [2] 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 [38]. 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 [39] 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 where regular kernel code still runs sequentially and the effect of interrupts is only visible when the device driver code is invoked. However, it does not support preemption, i.e. interrupts do not affect the scheduling of tasks. Concurrent CertiKOS (known as mC2) [40, 41] is the first verified kernel that can support multicore concurrency with fine-grained locking but its scheduler is still not preemptive.

Nemati et al. [42] verify the design of a hypervisor for the ARMv7 platform. In particular, they extend a formal model of the ARMv7 hardware with MMU, then formalize and prove that the hypervisor indeed guarantees spatial isolation of an untrusted guest OS: the guest's access to memory resources is indeed confined by the hypervisor. Unlike us, however, their modeling of the system only resembles its C implementation, instead of

being formally connected to it.

Nelson et al. [43] report the push-button verification of a kernel using the z3 SMT solver. In particular, they translate the implementation, specification, and their equivalence relation into SMT formulas, and check whether they are indeed equivalent to each other. Similarly, they translate invariants of the system as SMT formulas and check whether they are preserved by the specification. To achieve that though, they must make compromises on the interfaces to make them amenable to automated verification by avoiding complex loops or linked data structures. None of the above work addresses real-time features of an OS kernel.

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 [44]. Nevertheless, this proof only applies to a model of the kernel, and has no formal link with its implementation.

The eChronos OS [13] is a uniprocessor real-time operating system . 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 [45] and proves that the scheduler always schedules the highest-priority process. This work only models the behavior of different components and hardware interrupts without an end-to-end proof.

Xu et al. [14] verify μC/OS-II [46], 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 formalize and prove a notion of priority inversion freedom, which specifies the desired behavior in the presence of shared resources between tasks. However, their verification work stops at the policy level: they do not address more sophisticated high-level properties such as schedulability or temporal isolation.

9.3 Enforcement of Algorithmic Level Isolation Properties

Verified software systems with isolation Murray et al. [4] 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. [5, 47] prove the 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 [48], 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 (by yielding) 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 supported in our work.

Vassena et al. [49] redesign the runtime system of GHC to close the timing channel between concurrently running threads, in terms of reduction steps. On each single-processor, they explicitly partition time resource, i.e. number of reduction steps, among different threads to enforce a static schedule. For parallel execution on multiple cores, they design the top-level scheduler in a way such that all threads execute in lock-step: they synchronize after each step. In this way, they achieve deterministic parallelism.

Unverified temporal-constraint-aware systems Lyons et al. [50] tackle the issue of specifying and enforcing temporal constraints on a user-level task. They propose to in-

tegrate sporadic-server-based scheduling into an OS kernel with capability-based resource management. In particular, they provide to the user an interface for reserving a portion of CPU time for a specific critical task and use a classical server-based scheduling scheme to ensure an average-case resource guarantee. 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 [50]. However, the same reasoning framework applies in both cases.

9.4 Intransitive Non-Interference

The non-interference proof in this work is based on an observation function, proposed by Costanzo et al. [3]. However, there is also work relying on intransitive non-interference, such as nickel [5]. In particular, it allows information flow from domain A to B, and also from B to C, but can still guarantee that there is no information flow from A to C, i.e. the behavior of C is not influenced by A.

The main motivation of intransitive noninterference is to allow declassification or downgrading in a system. Assume that A is maintaining a file containing both a public and a secret portion. B is a filter that is supposed to erase the secret portion then release the file to the public. And C is a task reading from that file if permitted.

The original noninterference definition [51] compares two traces of actions. In particular, it states that starting from the same initial state, a task's behavior is not influenced by others unless there is explicit information flow between them. For example, consider two traces $\alpha_1 = A.write(public, 1) :: B.downgrade :: C.read$ and $\alpha_2 = B.downgrade :: C.read$. By the original definition, they should produce the same result since only B can influence the behavior of C. However, it is apparent that the released file contains different public sections and is distinguishable by C, rendering the original definition too restrictive for this scenario. This is where intransitive noninterference comes into play. It keeps all actions

that might affect C transitively, thus tolerating the different behavior between α_1 and α_2. In particular, the two traces are incomparable under this new definition, such that this downgrading system satisfies the noninterference requirement. This notion is proposed by Rushby [51] and is adopted by nickel [5].

However, it seems pointless to define an intransitive information flow policy when A actually influences C through an intermediary B. As shown in Fig. 4 in [5], the definition retains the transitive closure of all actions that might affect the behavior of the observer, implying that the influence is indeed transitive. Further, it is not strong enough to comprehensively specify the desired constraint on a system. For example, assume the implementation of B is buggy and it does not clean up the file before releasing it. Consider the following two traces:

$$\alpha_3 = A.write(secret, 1) :: A.write(public, 1) :: B.downgrade :: C.read$$
$$\alpha_4 = A.write(secret, 2) :: A.write(public, 1) :: B.downgrade :: C.read$$

The very purpose of defining noninterference is to specify that the two traces must lead to the same output from C's perspective and to capture insecure implementations by showing that they violate this definition. However, the definition used in [5] tolerates different results produced by these two traces, thus is not able to capture bugs in the downgrader.

Both Roscoe et al. [52] and Meyden [53] notice this issue and propose fixes on defining intransitive noninterference. Further, Meyden points out that the notion proposed in [52] is not strong enough in that it leaks the information of the interleaving of multiple critical components. In contrast, it draws inspiration from the definition of knowledge [54] and proposes alternative definitions to specify the desired property of an access control system. Yet, these fixes are ad-hoc and still lack a comprehensive description of the desired behavior of a task.

This is a challenging problem because there is indeed information flow from A to C, and the purpose of the isolation constraint is to control the amount of information transmitted, which is extremely hard to quantify. The fundamental problem with previous definitions is that they are specified on the trace of actions, instead of on the task's behavior directly. This indirection leads to the loose constraints as demonstrated above, which are not able

to capture buggy implementations.

As a comparison, our work advocates the observation-based noninterference proposed by Costanzo et al. [3]. For example, C's observation includes its virtual memory space, registers, and file f defined as below.

$$obs_C(f) = \begin{cases} \epsilon & f.released = false \\ f.public & f.released = true \end{cases}$$

Here, C's observation on file f is empty if it is not released yet, and equals the public portion of the file after its release. Further, instead of relying on an untrusted B, we implement the downgrading as a library function for A to invoke. Then noninterference is simple: starting from two observably equivalent states, if the released information from A is also equivalent, the final states of C must also be observably equivalent. The intuition is that we determinize C's behavior by parameterizing its execution with the release operation from A. On top of it, the execution of C is not influenced by others.

Now we show how this new definition specifies the isolation property. Firstly, it's straightforward to observe that a correct implementation satisfies this definition. Secondly, a buggy implementation, which does not clear the secret portion of the file, might result in different executions of C (reflected in the register state) and is thus caught by this definition. Thirdly, more subtle issues such as whether the downgrading operation can copy some contents from the secret section into the public section, can only be resolved by a manual inspection on the specification of downgrading. Fundamentally, this is part of the security policy specification and users should have the freedom to specify their own needs.

To conclude, the observation function approach gives us much flexibility in defining the isolation property in a straightforward way. As a comparison, specifications in the intransitive noninterference approach are indirect and error-prone.

9.5 Microarchitectural Level Isolation

There is rich literature tackling the microarchitectural aspect, i.e. caching effects, kernel overhead, etc. These low-level details might affect the actual execution time a critical

component may receive, and also affect the number of instructions it is able to execute within the same time window.

One line of work focuses on calculating an upper bound on the kernel overhead. Blackham et al. [11] conduct a static WCET analysis on the seL4 compiled binary code's interrupt response time. Based on a particular ARM processor, they overestimate the cycle count for different categories of instructions and use that to evaluate the cost of each execution path of the kernel code. They show that the WCET is sufficiently low without making the kernel fully preemptible. Sewell et al. [12] improve the trustworthiness of this analysis result by ensuring the soundness of the loop bound estimation and the infeasible path elimination phase. Both work relies on an overapproximation of the cycle count for instructions, in order to compensate for cache misses or other similar penalties. It is possible to incorporate such an analysis into our work by assigning each assembly instruction a worst-case cost. Nevertheless, the trustworthiness of the analysis relies heavily on this cost model, which is usually empirical and lacks a formal guarantee.

Another line of work aims to mitigate the caching effect by partitioning cache locations among different components so that they do not interfere with each other's rate of progress. Ge et al. [10] propose a series of countermeasures in designing an OS kernel to mitigate the microarchitectural level timing interference between components. In particular, they propose a few general requirements, such as to always flush cache upon domain switch, to partition OS kernel objects among domains, etc. These mechanisms reduce the sharing of microarchitectural level hardware resources and indeed mitigate these low-level timing channels. Heiser et al. [55] discuss the feasibility of proving such microarchitectural level temporal isolation. They suggest that it is possible to design an abstract cost model, such that the execution time of an instruction is deterministic w.r.t. a few microarchitectural level states, such as the related cache. They argue that this should be sufficient for proving that a well-partitioned system indeed enjoys temporal isolation between components. This approach is compatible with the assembly machine model used in our work, and its integration could help us prove the mitigation of caching effects.

Chapter 10

Limitations, Future Work, and Conclusion

10.1 Tasks with Dependencies

This work assumes independent tasks, i.e. a task's arrival and deadline only depend on its task parameters and a task is able to execute (though might be preempted) as soon as it arrives in the system. This is a common and practical setting for many real-time systems. However, there also exist systems that impose dependencies among tasks for various reasons. Such dependencies can be roughly categorized as precedence dependency, data dependency, and temporal dependency [18].

Precedence dependency There exists a precedence constraint between task τ_i and τ_j if τ_i has to finish before τ_j can start its execution. This kind of dependency is useful if deterministic communication is desired, i.e. τ_j's execution relies on data sent from τ_i [56]. In this way, communicating tasks follow the same rate and always synchronize with each other. Under this setting, precedence constraint is a partial order, and dependencies among all tasks form a directed graph.

One way of tackling this problem is to compute a schedule statically and let the scheduler follow it repeatedly. This is called offline scheduling and is discussed in Sec. 10.2. This section focuses on another approach called online scheduling, where the scheduler decides

the task schedule dynamically at runtime.

Given that all tasks follow a common period P, the precedence relation is not different from the preemption relation: a high-precedence task blocks a lower-precedence one in the same way a high-priority task blocks a lower-priority one. This gives rise to a modified EDF scheduling algorithm, which modifies the task arrival time and deadlines to make them consistent with the precedence graph and simply schedules them using an EDF scheduler [57]. Chpt. 8 discusses the verification of an EDF scheduler. Here, we discuss how to use the virtual timeline abstraction to express and reason about precedence constraints.

Recall that under this setting, each task τ_i is specified with a release time (r_i) and deadline (d_i), which repeats after every period of P. Since precedence constraints are usually specified using a task's start and end time, we define their computation as below.

Definition 21 (The start ($s_i(k)$) and end ($e_i(k)$) time of task τ_i in its k-th period)**.**

$$g_i(vt, k, t) = \begin{cases} t & \sigma_i(t) - \sigma_i(kP + r_i) = vt \\ g_i(vt, k, t+1) & otherwise \end{cases}$$

And

$$s_i(k) = g_i(1, k, kP + r_i) - 1 \wedge e_i(k) = g_i(C_i, k, kP + r_i)$$

Here, the function $g_i(vt, k, t)$ finds the first time instant after t such that the task's accumulative virtual time increments reach vt. On top of this, we can specify precedence constraints such as $\forall k, e_i(k) \leq s_j(k)$, which states that τ_j can only execute after τ_i is finished. And the framework guarantees that all properties proved on this abstraction carry down to the concrete scheduler implementation.

Data dependency Data dependency refers to asynchronous communication between tasks, e.g. through shared memory. Under this setting, communicating tasks do not need to synchronize with each other. We defer its discussion in Sec. 10.4.

Temporal dependency Temporal dependency refers to requirements that involve temporal distance between two consecutive executions of the same task. For example, a video

application may require that the difference between the display time for adjacent frames cannot exceed an upper bound such that the discreteness is not noticeable. Han et al. [58] define its task model as follows: a task τ_i arrives as soon as it finishes in the previous period, while the relative deadline, P_i, is fixed.

This is more generic and challenging than precedence dependency since it cannot be reduced to a priority assignment problem. Han et al. tackle this problem by transforming the task set into a periodic one, where periods of different tasks are harmonic (being multiples of each other) and satisfying $T_i \leq P_i$ for each task. The budget for each task does not change. Then they use a fixed-priority scheduler to schedule tasks following this new set of parameters, and they prove a utilization bound that guarantees the original temporal distance constraints.

We first show that the virtual timeline abstraction is able to formalize this kind of constraints. For simplicity, we assume that a task always uses up its budget within each period. Otherwise, we follow the same approach as in Sec. 3.2 to accommodate variable execution time. Since the constraint refers to the finishing time of each task, we define its computation as follows.

Definition 22 (The finishing time $fin_i(k)$ of τ_i in its k-th period)**.**

$$fin_i(k) = g_i(C_i, k, kT_i)$$

Here, g_i is defined in Def 21, and the definition of the finish time also follows the idea of $e_i(k)$. On top of it, we formalize the temporal constraint as below.

$$\forall k \geq 0, fin_i(k+1) - fin_i(k) \leq P_i$$

Since the system eventually uses a fixed-priority scheduler, the virtual time map for each task can be computed and then connected with this scheduler following Sec. 2.3 and Sec. 4.3. This transforms the reasoning of a scheduler into the abstract domain, and we can use the above definitions to prove that a system indeed satisfies these temporal distance constraints.

10.2 Constraint-Based Scheduling

The previous section discusses temporal dependencies between tasks and online scheduling algorithms for accommodating them. This section discusses the literature on constraint-based scheduling, which is a more generic way of scheduling tasks with temporal constraints.

In contrast to the periodic task model we have discussed throughout this work, constraint-based scheduling defines its task model as a set of one-shot tasks. Each task has a release time, deadline, and execution time. Depending on whether jobs can preempt each other, the actual schedule of a task is either specified with a start time or is specified with a set of active time instants [59, 60]. The validity of the schedule requires that the start and end time of a task must be within its release time and deadline, and that the number of time instants occupied must be equal to its execution time. On top of it, a system adds other temporal constraints and use a solver to compute a feasible schedule. It can also optimize the schedule based on a given metric.

This is called offline scheduling, where the set of tasks is known and the scheduler carries out a fixed schedule (e.g. specified with a table containing the time slot allocation at each time instant) instead of relying on a policy to decide the next task dynamically at runtime [18]. The virtual timeline framework can be extended to accommodate constraint-based scheduling and formally connect these abstract constraints with the concrete scheduler implementation.

Firstly, we formalize these temporal constraints using the virtual timeline abstraction. Under this setting, we define the virtual time map of a task in the same way as we define the time map for a partition in Sec. 5.2. In particular, a task's virtual time only includes time occupied by this task since there is no preemption relation among them. Assume that the release time and deadline for a task τ_i is r_i and d_i, respectively. The validity of a schedule is straightforward:

$$\begin{cases} \sigma_i(d_i) - \sigma_i(r_i) = C_i \\ \forall t, \sigma_i(t+1) = \sigma_i(t) \vee \sigma_i(t+1) = \sigma_i(t) + 1 \end{cases}$$

The first condition means that this task indeed receives its full execution time. The second

condition specifies that a task either occupies a time slot or idles at that time instant. We also define a virtual time for all tasks, $\Xi(t) = \sum \sigma_i(t)$, and we require that

$$\forall t, \Xi(t+1) = \Xi(t) \vee \Xi(t+1) = \Xi(t) + 1$$

This means at any moment, there could be at most one task scheduled.

Further, temporal constraints are specified using the start and end time of a task, which is computed as below.

Definition 23 (The start (s_i) and end (e_i) time of task τ_i)**.**

$$h_i(vt, t) = \begin{cases} t & \sigma_i(t) = vt \\ h_i(vt, t+1) & otherwise \end{cases}$$

And

$$s_i = h_i(1, 0) - 1 \wedge e_i = h_i(C_i, 0)$$

In this way, we are able to specify temporal constraints using tasks' virtual time maps and solving these constraints will produce feasible virtual timelines. In fact, we can view the virtual timeline as an alternative to the finite set variable used in [60] which denotes the schedule of a task. We can either implement these constraint programming techniques in Coq or simply use external tools to generate feasible time maps then validate them in Coq. In either way, we are able to prove that those constraints indeed hold on the virtual timeline abstraction.

Finally, we connect these generated virtual timelines with a concrete scheduler. A concrete scheduler relies on a time table that is consistent with the feasible schedule produced by the constraint solver. This allows us to construct it with the virtual timelines in a straightforward way. The virtual-time-based scheduler is the same as in Fig. 4.4, and we follow the same approach to prove that the concrete scheduler is contextually equivalent with this virtual-time-based scheduler, which is further consistent with individual virtual timelines. Thus, all properties proved on the virtual timeline also hold on the concrete scheduler implementation. This concludes the extension with constraint-based scheduling.

```
void instrumented_sched(){
  ......
  // refill budgets
  ......

  int used = 0;
  for(int i = 0; i < N; i++){
    if (used < m){
      vt[i][t+1] = vt[i][t] + 1;
      if (quanta[i] > 0){
        // schedule i on a core
        ......
        used++;
      }
    }else{
      vt[i][t+1] = vt[i][t];
    }
  }
}
```

Figure 10.1: Illustration of the dynamic computation of time maps for m identical cores

10.3 Multicore Scheduling

Our work focuses on the scheduling on a uniprocessor. A natural extension is to adapt it for multicore platforms, which are popular in the current world. There are two categories of multicore scheduling, the global and the partitioned approach, depending on whether task migration between cores is allowed or not. In the partitioned approach, the task set on each core is fixed, which allows us to reuse existing proofs on the uniprocessor scheduling. However, in the global scheduling scheme, tasks migrate between cores and the parallel execution of multiple tasks on different cores is possible.

Leung et al. [61] point out that neither of the above two approaches is strictly better than the other. This section focuses on how to extend the existing virtual timeline framework to support global scheduling on multicore platforms since the other approach is straightforward.

For example, Andersson et al. [62] propose a global rate-monotonic scheduling on m identical processors, and prove a utilization bound for schedulable task sets. Even though the scheduler still follows a fixed priority, the static construction of time maps as shown in Def. 1 no longer works because it is based on the assumption that there could be at most one task running at any moment. This requires us to extend the general principle described in Fig. 2.5 to accommodate the situation that a time slot is available to a task if no more than (m - 1) higher-priority tasks are running.

We demonstrate the extended principle in Fig. 10.1. It omits details about how to assign a core to a task and only focuses on the construction of time maps. Here, the virtual time of a task stagnates if and only if there are already m higher-priority tasks scheduled

for the coming time slot. In this way, if a task is schedulable in its virtual timeline, the scheduler must be able to find a core to accommodate this task, such that the virtual timeline abstraction is consistent with the concrete scheduler implementation.

Similarly, the virtual timeline framework can also be generalized with a dynamic priority assignment in the multicore setting, by following the approach described in Chpt. 7. Thus, it can accommodate dynamic algorithms such as the multiprocessor EDF scheduling [63] in the same way as discussed above.

10.4 Communicating Tasks

Sec. 1.3 discusses how communications can be supported in our current framework. This section provides more detailed discussion. In particular, we address issues related to the schedulability of the system and the isolation property between tasks.

Kernel objects for I/O operations As mentioned in Sec. 1.3, one common motivation for supporting communication is to allow tasks access to I/O operations. Under this setting, we choose to hide the complex interactions into one of the abstraction layers. For example, it is common to dedicate one processor for all I/O operations [9] to minimize its interference on regular applications. We can implement them as kernel objects supporting (buffered) read and write operations. On the I/O processor, we design the reader task to poll a device periodically, and the system call for reading always returns the most recent result. Similarly, the system call for writing also writes to a buffer, which is polled periodically by a writer task. In this way, the execution of I/O operations does not interfere with regular tasks.

I/O also breaks the noninterference proof between tasks because it usually leads to the nondeterministic behavior of a task. We can follow the same approach as discussed in Sec. 9.4 and treat the reading operation as declassification. In this way, we explicitly allow the information flow from the device, while also formally specifying what is the desired behavior of declassification.

General-purpose communication For general-purpose communication between tasks, their influences on each other are intentional. Under this setting, we enclose a group of

communicating tasks in a partition to ensure that components outside this partition are not affected by them. And the isolation proof follows the approach discussed in Sec. 5.2.

Here, we focus on the schedulability aspect. For example, one common issue with shared-memory communication is that access to the shared memory must be mutually exclusive to ensure the atomicity of these transactions. However, this could lead to situations where a low-priority task blocks a higher-priority one, which is called priority inversion [64] and may hurt the schedulability of the system. The literature recognizes this problem and proposes ways to reduce the corresponding blocking time incurred on a task. Among them are the priority inheritance protocol and the priority ceiling protocol [65]. Both protocols rely on an upper bound on the length of each task's critical section, and they boost a task's priority when it's holding the shared memory to achieve a bound on the total blocking time.

We can follow the general approach of the priority ceiling protocol since it achieves the lowest blocking time. We model shared memory as objects and limit users' access to them through pre-defined methods associated with each object. Further, an object is tied to a partition and is not accessible from outside this partition. We also ask tasks to declare which objects they intend to access upon creation. In this way, the length of the critical section (the execution time of the object's methods) can be analyzed and controlled in a disciplined manner, and the information about which tasks may access this object also enables us to define a priority for this object. In particular, the priority of this object is higher than any task that might invoke methods on it. In this way, mutual exclusion is automatically achieved and the blocking time is at most the WCET of one method call [65]. In this way, the scheduling of this object also falls within the virtual timeline framework since it is also scheduled according to a priority. The only difference is that it is not a periodic task but is triggered by others. However, we can approximate its invocation rate and treat it as a periodic task in the analysis, and further prove that the schedulability analysis in [65] indeed applies to this scenario.

10.5 Kernel Overhead

User-level preemption Similar to [5], 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 reduce the interrupt latency without introducing kernel preemption. This is possible if we adopt the bottom-halves mechanism (as used in μC/OS-III [66]) 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 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, as well as the scheduler themselves, also consume time. In both cases, these small overheads prevent a task from executing for its full budget.

We have discussed in Sec. 1.3 that the budget enforcement mechanism ensures the validity of the schedulability analysis since kernel overhead only eats into the available execution time for a task instead of devastating the whole system. 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 make this jitter explicit so that users are aware of this loss of time and are able to compensate for it when declaring budgets for their tasks.

Even though the WCET computation, in general, is a hard problem and relies heavily on microarchitectural-level details such as caches, TLBs, and so on, we can implement the OS kernel in a way that facilitates its computation. For example, as discussed in Sec. 1.3, our work exposes a limited set of system calls to real-time tasks, which only contain simple straight-line code without loops (we measure their average execution time to be $0.47\mu s$ on a 2.8 GHz machine). This removes the challenge of computing loop bounds and makes the WCET analysis on the assembly code level more tractable. Secondly, instead of relying on dynamic memory allocation, kernel modules in CertiKOS rely extensively on pre-allocated global memory for their data, resulting in more predictable caching behavior and simpler code. Last but not least, there is no page fault in the kernel mode. All these characteristics simplify the WCET analysis and make it more plausible even with an imprecise or imperfect cost model.

10.6 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 [5, 4]. 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. 1) 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.

10.7 Microarchitectural-Level Interference

We discuss upfront that this work focuses on the algorithmic instead of the microarchitectural level aspect. For task-level isolation, this means that whether or not a task can finish its execution within the number of time slots specified by its budget might depend on the behavior of others due to caching effects. Similarly, for partition-level isolation, the exact instruction location where one task is preempted by another may also depend on the behavior of other partitions, thus leading to non-deterministic instruction-level interleaving.

Both are critical issues if one wants to achieve microarchitectural-level temporal isolation. However, ultimately, the feasibility of closing such timing channels relies on a secure interface/contract provided by the underlying hardware, which is not available yet [10]. Nevertheless, even with commodity hardware today, there are ways of mitigating such channels by partitioning and flushing shared microarchitectural-level hardware resources [10, 9, 7, 8].

The integration of the above mitigation mechanisms does not require too much restructuring of existing code: they are local to the implementation of context switches or memory allocation, and they do not interfere with other functionalities of the OS kernel. The reasoning of its guarantees is also compatible with the current temporal reasoning framework: the propose of the above mechanisms is to ensure that a task exhibits deterministic behavior within every time slot, which entails microarchitectural-level isolation if combined with existing properties of the time slot allocation.

Finally, our current machine model is ready to incorporate such microarchitectural level details. As discussed in Sec. 4.2, we maintain two abstract functions, `intr_trigger` and `intr_handler` to model interrupt handling. It's straightforward to extend the definition of `intr_trigger` with a cost model, possibly in the form of an uninterpreted function over microarchitectural level states as discussed in [55], such that we are able to formally compare the execution time of a task over different runs.

10.8 Conclusions

This thesis 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 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. Combining everything together, we achieve both temporal and spatial isolation between different components.

Finally, we demonstrate that the virtual timeline abstraction is not restricted to fixed-priority scheduling. We use it to reason about earliest-deadline-first scheduling, which exhibits a dynamic priority assignment, and we connect it with the concrete implementation of an EDF scheduler. This indicates that the virtual timeline abstraction is commonly

applicable to verifying preemptive schedulers.

Bibliography

[1] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'15, pages 595–608, New York, NY, USA, 2015. ACM.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, New York, NY, USA, 2009. ACM.

[3] D. Costanzo, Z. Shao, and R. Gu. 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*, pages 648–664, New York, 2016. ACM.

[4] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. 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*, pages 415–429, Washington, DC, 2013. IEEE Computer Society.

[5] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for design and verification of information flow control sys-

tems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 287–305, Carlsbad, CA, 2018. USENIX Association.

[6] M. T. Higuera-Toledano and A. J. Wellings. *Distributed, Embedded and Real-Time Java Systems*. Springer Publishing Company, Incorporated, 2012.

[7] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, April 2013.

[8] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

[9] 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. Real-time computing on multicore processors. *Computer*, 49(9):69–77, Sept 2016.

[10] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser. Time protection: The missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys'19, pages 1:1–1:17, New York, NY, USA, 2019. ACM.

[11] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *2011 IEEE 32nd Real-Time Systems Symposium (RTSS'11)*, pages 339–348, Washington, DC, Nov 2011. IEEE Computer Society.

[12] T. Sewell, F. Kam, and G. Heiser. High-assurance timing analysis for a high-assurance real-time operating system. *Real-Time Systems*, 53(5):812–853, Sep 2017.

[13] J. Andronick, C. Lewis, D. Matichuk, C. Morgan, and C. Rizkallah. Proof of os scheduling behavior in the presence of interrupt-induced concurrency. In *Proceedings*

of 7th International Conference on Interactive Theorem Proving (ITP), pages 52–68, Nancy, France, 2016. Springer International Publishing.

[14] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive os kernels. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification: 28th International Conference (CAV'16), Toronto, ON, Canada, July 17-23, 2016, Proceedings*, pages 59–79, Berlin, Heidelberg, 2016. Springer International Publishing.

[15] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS'05, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[16] QNX. Neutrino rtos, 2019.

[17] J. Kim, T. Abdelzaher, and L. Sha. 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)*, pages 221–231, Washington, DC, April 2015. IEEE Computer Society.

[18] J. W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[19] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[20] M. Liu, L. Rieg, Z. Shao, R. Gu, D. Costanzo, J.-E. Kim, and M.-K. Yoon. Virtual timeline: A formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[21] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings. Real-Time Systems Symposium (RTSS'89)*, pages 166–171, Washington, DC, Dec 1989. IEEE Computer Society.

[22] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[23] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[24] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a c compiler front-end. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 460–475, Berlin, Heidelberg, 2006. Springer-Verlag.

[25] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model. In A. W. Appel, editor, *Program Logics for Certified Compilers*. Cambridge University Press, Cambridge, UK, April 2014.

[26] ARINC. *ARINC Specification 653 Part 1*. ARINC, Annapolis, MD, 2015.

[27] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[28] B. Sprunt, L. Sha, and J. Lehoczky. 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, 1989.

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

[30] M. Wilding. 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, pages 369–378, London, UK, UK, 1998. Springer-Verlag.

[31] B. Dutertre. Formal analysis of the priority ceiling protocol. In *Proceedings 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pages 151–160, Washington, DC, 2000. IEEE Computer Society.

[32] X. Zhang, C. Urban, and C. Wu. Priority inheritance protocol proved correct. In *Interactive Theorem Proving (ITP'12)*, pages 217–232, Berlin, Heidelberg, 2012. Springer.

[33] X. Zhang, C. Urban, and C. Wu. Priority inheritance protocol proved correct. *Journal of Automated Reasoning*, 64(1):73–95, 2020.

[34] F. Cerqueira, F. Stutz, and B. B. Brandenburg. Prosa: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS'16)*, pages 273–284, Germany, July 2016. Schloss Dagstuhl.

[35] X. Guo, M. Lesourd, M. Liu, L. Rieg, and Z. Shao. Integrating formal schedulability analysis into a verified os kernel. In *Computer Aided Verification - 31st International Conference (CAV'19), July 15-18, Proceedings*, pages 496–514, Berlin, Heidelberg, 2019. Springer.

[36] G. Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, Feb 2009.

[37] Y. Zhao, Z. Yang, and D. Ma. A survey on formal specification and verification of separation kernels. *Front. Comput. Sci.*, 11(4):585–607, August 2017.

[38] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 471–482, New York, NY, USA, 2013. ACM.

[39] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. 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*, pages 431–447, New York, 2016. ACM.

[40] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 653–669, GA, 2016. USENIX Association.

[41] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 646–661, New York, 2018. ACM.

[42] H. Nemati, R. Guanciale, and M. Dam. 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*, pages 578–589, Berlin, Heidelberg, 2015. Springer-Verlag.

[43] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. 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*, pages 252–269, New York, NY, USA, 2017. ACM.

[44] R. J. Richards. *Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel*, pages 301–322. Springer US, Boston, MA, 2010.

[45] J. Andronick, C. Lewis, and C. Morgan. 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)*, pages 10–24, Suva, Fiji, 2015. EPTCS.

[46] J. J. Labrosse. *Microc/OS-II.* Focal Press, New York, 2nd edition, 1998.

[47] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. A note on verifying information flow control systems with nickel. Technical Report UW-CSE-2019-10-01, School of Computer Science & Engineering, University of Washington, Oct 2019.

[48] A. C. Myers and B. Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, October 1997.

[49] M. Vassena, G. Soeller, P. Amidon, M. Chan, J. Renner, and D. Stefan. Foundations for parallel information flow control runtime systems. In F. Nielson and D. Sands, editors, *Principles of Security and Trust*, pages 1–28, Cham, 2019. Springer International Publishing.

[50] A. Lyons, K. McLeod, H. Almatary, and G. Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys'18, pages 26:1–26:16, New York, NY, USA, 2018. ACM.

[51] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI, dec 1992.

[52] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Workshop on Computer Security Foundations*, CSFW '99, page 228, USA, 1999. IEEE Computer Society.

[53] R. van der Meyden. What, indeed, is intransitive noninterference? In J. Biskup and J. López, editors, *Computer Security – ESORICS 2007*, pages 235–250, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[54] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge.* MIT Press, Cambridge, MA, USA, 2003.

[55] G. Heiser, G. Klein, and T. Murray. Can we prove time protection? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 23–29, New York, NY, USA, 2019. Association for Computing Machinery.

[56] M. Spuri and J. A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12):1407–1412, Dec 1994.

[57] K. Erciyes. *Uniprocessor-Dependent Task Scheduling*, pages 183–202. Springer International Publishing, Cham, 2019.

[58] Ching-Chih Han, Kwei-Jay Lin, and Chao-Ju Hou. Distance-constrained scheduling and its applications to real-time systems. *IEEE Transactions on Computers*, 45(7):814–826, July 1996.

[59] R. Barták. Constraint-based scheduling: An introduction for newcomers. *IFAC Proceedings Volumes*, 36(3):75 – 80, 2003. 7th IFAC Workshop on Intelligent Manufacturing Systems - IMS 2003 [7th IFAC Workshop Preprints], Budapest, Hungary, 6-8 April 2003.

[60] P. Baptiste, P. Laborie, C. L. Pape, and W. Nuijten. Chapter 22 - constraint-based scheduling and planning. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 761 – 799. Elsevier, 2006.

[61] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982.

[62] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, pages 193–202, Dec 2001.

[63] T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 120–129, Dec 2003.

[64] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, February 1980.

[65] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep. 1990.

[66] J. J. Labrosse. *Microc/OS-III.* Micrium Press, Austin, TX, 2011.