Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Proving Fairness and Implementation Correctness of a Microkernel Scheduler

Matthias Daum · Jan Dörrenbächer · Burkhart Wolff

Received: date / Accepted: date

Abstract We report on the formal proof of a microkernel's key property, namely that its multi-priority process scheduler guarantees progress, i. e., strong fairness. The proof architecture links a layer of behavioral reasoning over system-trace sets with a concrete, fairly realistic implementation written in C.

Our microkernel provides an infrastructure for memory virtualization, for communication with hardware devices, for processes (represented as a sequence of assembly instructions, which are executed concurrently over an underlying, formally defined processor), and for inter-process communication (IPC) via synchronous message passing. The kernel establishes process switches according to IPCs and timer-events; the scheduling of process switches, however, follows a hierarchy of priorities, favoring, e. g., system processes over application processes over maintenance processes.

Besides the quite substantial models developed in Isabelle/HOL and the formal clarification of their relationship, we provide a detailed analysis what formal requirements a microkernel imposes on the key ingredients (hardware, timers, machine-dependent code) in order to establish the correct operation of the overall system. On the methodological side, we show how early modeling with foresight to the later verification has substantially helped our project.

Keywords Microkernel · Formal Verification · Interactive Theorem Proving · Isabelle/HOL

1 Introduction

"Real system code" has recently been recognized as a fruitful target for formal verification. By this term, we refer to code actually found in typical operating systems, which implement

Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

M. Daum · J. Dörrenbächer
Saarland University, Computer Science Dept.
E-mail: {md11,jandb}@wjpserver.cs.uni-sb.de

B. Wolff
Université Paris/Orsay, LRI
E-mail: wolff@lri.fr

preemptive multi-tasking, are written in a mix of C and assembly, are compiled with optimizing compilers, and run on modern processors with architecturally visible caches and incoherent memory. Since "real" C code characteristically relies on an architecture-dependent execution model, it has long been considered to be "dirty" and out of the reach of formal verification in academia. With the availability of realistic models for hardware processors [12, 10, 22, 37] in interactive theorem provers, and with the advent of more powerful automated theorem-proving techniques, this global picture has changed: Real system code verification is meanwhile at the brink of feasibility, albeit still representing a grand challenge. At the same time, verification efforts of system-level programs meet a growing demand by software-engineers, which are confronted by both, the rising complexity of computer architectures as well as higher reliability requirements of their code.

We call real system-code verification a grand challenge because, at present, all current approaches — including ours — compromise in one way or the other: the logical foundations are quite problematic, the computer architectures are simpler than industrial-strength processors, the code size is fairly small, or the underlying execution model of C and assembly makes severe simplifications (see Sect. 8.1).

The availability of a hardware-processor model also represents the foundational layer of our work. This model describes, at gate level, the transitions of the RISC processor VAMP. A small piece of software executed on the VAMP, called CVM, provides an abstract layer running concurrently a system process as well as a number of user processes, i. e., lists of assembly sequences assuming separate, linear memory. We implemented the microkernel VAMOS, which is executed as this system process. Our implementation uses the C fragment *C0*, which can be translated into assembly code by a verified compiler [32, 31]. VAMOS provides a process scheduler, an infrastructure for communication with hardware devices, and message passing between processes. All software layers are formally specified; refinement relations correlate the adjacent layers such that eventually, all specification layers can be mapped down to our hardware-processor model. The whole model stack is formalized in the theorem prover Isabelle/HOL [39].

In more detail, CVM and VAMOS together realize an abstract transition δ by an implementation function `kernel_step` as shown in Fig. 1. The transition function δ represents the execution of a non-empty portion of a user process including a possible process switch. Consequently, we need a refinement proof establishing that the implementation with its underlying state σ indeed realizes the high-level state transition δ over the corresponding abstract state s, where abstract and concrete states are linked via a formally defined abstraction relation.

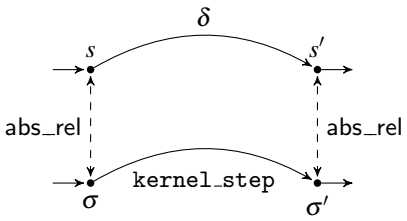

Fig. 1 Kernel step refinement (simplified)

On top of the VAMOS layer, we put a further theory layer for reasoning over the temporal behavior of the kernel executions. This layer arranges consecutive δ steps (see Fig. 2) and provides the necessary infrastructure for the proof of a fairness property of the scheduler. In particular, we define key-

Fig. 2 Kernel traces (simplified)

notions like the set of all possible traces of VAMOS and formulate an invariant of the states in a trace.

We formulate the fairness property of the scheduler in terms of liveness. More specifically, it is a strong fairness property as classified by Francez [23]. In principle: If a certain process is infinitely often ready to compute, it will always eventually compute. Technically, the matter is complicated by the fact that the scheduling policy supports static priorities. It is the nature of static-priority scheduling that it limits fairness to processes of *the same priority level*. Switches to processes of a *lower priority* occur only if all processes of the higher priorities are not ready to compute. As a consequence of this form of prioritization, a (group of) high-level processes might diverge and low-level processes will never be executed. Our notion of *prioritized fairness* reflects this phenomenon.

Based on prior work stemming from the Verisoft Project [26,3] (in particular VAMP [10], CVM [27,48,49], C0 [31,32] and the proof environment Isabelle/Simpl [45,44]), we have built our key contribution of this paper: It consists of the VAMOS layer, both specification as well as implementation, and a theory layer culminating in the fairness theorem of the VAMOS scheduler.[1] To our knowledge, such a theorem has been shown over a realistic kernel for the first time. One might object that VAMP, C0 and VAMOS are built with foresight for verification and rather academic components than industrial-strength processors or microkernel code. The VAMP, however, comprises all essential features of a RISC architecture including translation look-aside buffers and memory management units, and VAMOS, with its 3000 lines of C0 (excluding CVM), is not too far away from industrially developed microkernels (see discussion Sect. 8.1).

This paper proceeds essentially by following the proof architecture as outlined above: after a background section explaining the context of this work as well as logical and technical preliminaries related to the used verification technique, we present the model and the implementation of the kernel, the refinement proof linking these two, and the temporal theory over kernel execution traces leading to our final result: the proof of prioritized fairness for the process scheduler.

2 Background

2.1 The Verisoft Project

Although not inherently necessary, it is perhaps instructive to outline the context of this work: It is actually an extract of a stack of system models developed in the Verisoft project (funded by the German Federal Ministry of Education and Research, it was finished in 2007; currently, its successor Verisoft XT[2] is under way with a slightly different focus). The project adheres to the very idea of pervasive verification [9,36]: Each abstraction layer is justified by simulation theorems that permit transferring the results to the low-level models. All the development is mechanized in the uniform logical framework of the interactive theorem prover Isabelle/HOL and hence it is rigorously checked that all the results fit together.

Our microkernel belongs to Verisoft's so-called *Academic System* (as opposed to systems in subprojects with partners from industry), which is a distributed computer system for the exchange and management of signed and encrypted e-mails. While the system is able to receive external e-mails, each computer within the system uses identical hardware and the

[1] The theory files are soon available at http://www.verisoft.de/VerisoftRepository.html.

[2] The project's homepage is http://www.verisoftxt.de/.

same operating-system software. In Verisoft, we implemented and verified an e-mail client, an e-mail server, and a cryptography server that run on top of this operating system.

Fig. 3 depicts the hard- and software layers of a single computer in the system. The lowest software layer is called *communicating virtual machines* (CVM). This layer encapsulates all the hardware-specific low-level kernel functionality, which uses inlined assembly. Technically, this software constitutes the interrupt-service routine of the processor. CVM's major task, however, is memory virtualization and process separation. Hence, CVM includes a page-fault handler with a simple memory swapping facility [4]. The remaining functionality of our microkernel is implemented by the hardware-independent kernel part. CVM exports an interface of so-called *primitives* for the access to and manipulation of user processes to this kernel part. We have thereby established a solid framework for microkernel construction.

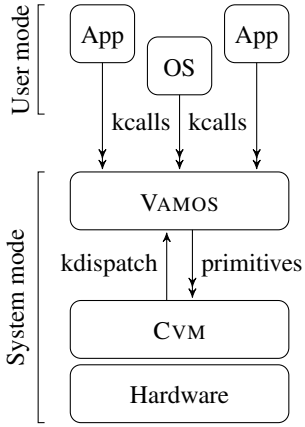

Fig. 3 System stack

Using this framework, we have implemented our microkernel VAMOS in the C variant *C0* without inlined assembly. When an interrupt occurs, CVM preserves the old processor context, establishes a suitable C0 environment and calls the function `kdispatch` of VAMOS. For the manipulation of the user memory or registers, VAMOS may call the primitives of CVM. The return value of `kdispatch` determines which process resumes when the kernel execution finishes.

While CVM and VAMOS run in the privileged system mode of the processor, processes run in the unprivileged user mode. In the figure, we labeled one process "OS" for the operating system and the others "App" as abbreviation for application (e. g., e-mail client or server). The OS process constitutes the highest layer of our operating system. It features an advanced rights management with different users, implements a sophisticated access control to kernel services like process creation and provides further services like file-system and network access. All processes interact with the kernel via *kernel calls*. The special instruction `trap` causes an exception, which is handled in VAMOS. VAMOS can examine and alter the state of the process using CVM primitives, thus identifying the process's specific request and storing the kernel's corresponding response.

2.2 Isabelle/HOL

HOL [13,5] is a classical logic based on the typed λ-calculus, where types can be built by *type variables* α, β, γ, etc., type constructors like bool, int, α set, α list, and $\alpha \Rightarrow \beta$, where the latter is used to denote total functions. Further, there is the universal equality $\_ = \_$ of type $\alpha \Rightarrow \alpha \Rightarrow$ bool.

The logic HOL has been implemented as instance in the generic proof-assistant Isabelle [39]; nowadays, Isabelle/HOL is that instance of Isabelle, which is mostly used and developed. A few axioms describe the logical core system based on the logical type bool with the logical connectives $\neg\_$, $\_ \wedge \_$, $\_ \vee \_$ and $\_ \longrightarrow \_$, which are constants of type

bool⇒bool or [bool, bool]⇒bool. Moreover, there are the quantifiers $\forall x.\,P\,x$ and $\exists x.\,P\,x$ which can range over arbitrary types in HOL. In the literature on Isabelle, it is common to distinguish the built-in ("meta-level") implication $\_ \Longrightarrow \_$ and equality $\_ \equiv \_$ from their HOL-counterparts $\_ \longrightarrow \_$ and $\_ = \_$; throughout this paper, these two operators can be considered equivalent. We also use the notation $[\![A_1;\ldots;A_n]\!] \Longrightarrow A$ as abbreviation for $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$.

This core language can be extended via axiomatic ("conservative") definitions to large libraries comprising Cartesian product types $\_ \times \_$ with the usual projections fst and snd. The set type α set can be introduced isomorphic to the function space α⇒bool, i. e., to characteristic functions, and a typed set theory is introduced with the usual operators, e. g., $\_ \in \_$, $\_ \cup \_$, $\_ \cap \_$.

The HOL type constructor $\tau_\perp$ assigns to each type τ a type *lifted* by $\perp$. The function $\lfloor\_\rfloor : \alpha \Rightarrow \alpha_\perp$ denotes the injection, the function $\lceil\_\rceil : \alpha_\perp \Rightarrow \alpha$ its inverse for defined values; for undefined values the function is left unspecified (though it is technically defined). Partial functions $\alpha \rightharpoonup \beta$ are just functions $\alpha \Rightarrow \beta_\perp$, over which the usual concepts like partial function update $f(a \mapsto b)$ are defined.

Isabelle/HOL supports record notation, which we use extensively. Record types are denoted, for example, by:

$$\textbf{record}\ \mathsf{T} = \mathsf{a} :: \mathsf{T1} \quad \mathsf{b} :: \mathsf{T2}$$

which implicitly introduces the record constructor $(\!|\mathsf{a} = e_1, \mathsf{b} = e_2|\!)$ as well as the update of record r in field a, written as $r(\!|\mathsf{a} := x|\!)$.

2.3 The Verification Environment Isabelle/Simpl for C0

For the verification of C0, we use a general program-verification framework for sequential imperative programming languages: Isabelle/Simpl [44,45]. It is build as a conservative extension on top of Isabelle/HOL. The key feature of Isabelle/Simpl is the notion of a Hoare-Triple:

$\Gamma \vdash P\ c\ Q$

In a procedure environment Γ, this statement claims that under the assertion P on the original program state, the assertion Q will hold after the execution of the code c given in the *Simpl* language. The assertions P and Q are simply sets of states. In principle, Isabelle/Simpl is polymorphic over the state space; we use records but hide the details by an Isabelle syntax, such that $\{\sigma.\ ^\sigma\mathsf{var} = 5\}$ denotes the assertion that the value of program variable var in state σ is five.

Expressions in Simpl are HOL expressions. In addition to the HOL operations, we have defined bitwise conjunction $\_ \wedge_\mathsf{u} \_$ and disjunction $\_ \vee_\mathsf{u} \_$ over natural numbers. In contrast to expressions, statements are represented by an abstract datatype. The statement syntax is highly abstract, e. g., Basic f represents a state update using function f. In order to present programs in conventional terms, we employ Isabelle's powerful syntax translation machinery and denote a program variable by $'\mathsf{var}$, an assignment by $'\mathsf{var} :== 5$, a conditional by **IF** b **THEN** s_1 **ELSE** s_2 **FI**, a procedure call by $'\mathsf{var} :==$ **CALL** update(5), etc. The index g is used to instruct the parser to generate guards that protect against runtime faults like overflows.

The procedure environment Γ is a partial function from procedure names to statements. These statements constitute the procedure body and are defined by the Isabelle/Simpl command **procedures**. The following command, for instance, defines the procedure update:

procedures update(var | res_nat) =
  'res_nat :=$_g$ 'var

It has one formal parameter called var and the result to return to the calling function is held in variable res_nat, i. e., the bar separates input and output parameters. When formally specifying the functionality of the procedure, we write 'res_nat :== **PROC** update('var) as shorthand for the code of procedure `update`:

$\Gamma\vdash \{\sigma.\ {}^{\sigma}\mathsf{var} = x\}$
  'res_nat :== **PROC** update('var)
  $\{\tau.\ {}^{\tau}\mathsf{res\_nat} = x\}$

states that procedure `update` assigns the value of its argument to the return variable.

The framework includes a big-step semantics, a Hoare logic for partial as well as total correctness and an automated verification-condition generator for Simpl. Within this sequential core language, assembly fragments as well as the C fragment C0 are embedded — thus, Isabelle/Simpl plays a similar role as *Boogie* for the Spec#/VCC environment [6] or the WHY-tool for Caduceus [20]. The embedding is based on a compiler converting C0-constructs in terms of operations provided by the small-step semantics of the VAMP machine. A correctness proof for this compiler, that links the small-step semantics to the Simpl big-step semantics, is also provided [2]. This correctness theorem about the embedding of C0 into Simpl allows for mapping low-level properties to more abstract ones formulated on the big-step semantics of C0. Throughout this paper, we will present all algorithms in Simpl (so that we can rely on a uniform Isabelle/HOL foundation); note, however, that this Simpl code is the result of an automatic translation from the C0 code that is actually compiled and runs on the machine. Alkassar *et al.* have dedicated a separate article [3] on the semantics stack in Verisoft.

C0 is a well-typed language comprising compound values, i.e., structures and arrays. We employ the HOL type system to model C0 programming language types. Isabelle's type inference then takes care of typing constraints that would otherwise have to be explicitly maintained in the assertions. Propositions, however, that explicitly refer to the memory layout or to hardware device registers cannot be proven on the C0-Hoare-Logic level; in these situations, the verification necessarily descends down to the VAMP level. Our approach is to abstract the effect of those low-level computations into atomic *XCalls* (extended calls) in all our semantic layers. In particular, the state-space of C0 is augmented with an additional component that represents the state of the external component, e. g., a device. An XCall is a procedure call that performs a transition on this external state and communicates with C0 via parameter passing and return values. With this model, it is straightforward to integrate XCalls into the semantics and into Hoare logic reasoning. XCalls are typically implemented in assembly.

2.4 Code Verification in Isabelle/Simpl

The introduction presents a bird's eye view on the refinement as depicted in Fig. 1. In principle, we can reformulate the depicted claim in Isabelle/Simpl as follows:

$\Gamma\vdash \{\sigma.\ \mathsf{abs\_rel}\ \sigma\ s\}$
  PROC kernel_step()
  $\{\tau.\ \mathsf{abs\_rel}\ \tau\ (\delta\ s)\}$

Assuming an abstraction relation abs_rel that holds for a concrete state σ and an abstract state s, the relation is preserved by the transitions `kernel_step` on the concrete level and δ

on the abstract level. Just like the figure, this statement is an over-simplification. We postpone the details to Sect. 6.

Our approach to code verification combines refinement and code correctness, i. e., contracts are specified in terms of the abstract states. As a simple example, we assume a library function `append`, which appends an element to a singly-linked list. In the implementation, the list is represented as a pointer structure described by the variables $'\mathsf{head}$ and $'\mathsf{next}$. In order to prove that a specific pointer structure in the state τ after a function invocation indeed denotes a list, it is useful to know that the original pointer structure in state σ already denoted a list. We encapsulate the list property in a predicate $\mathsf{inv}_{\mathtt{list}}$ over the pointer variables and formulate correctness of $\mathtt{append}('\mathsf{head}, '\mathsf{elem})$ as follows:

$$\begin{array}{l}\Gamma\vdash \{\sigma.\ \mathsf{inv}_{\mathtt{list}}\ ^{\sigma}\mathsf{head}\ ^{\sigma}\mathsf{next} \wedge \mathsf{abs\_rel}_{\mathtt{list}}\ ^{\sigma}\mathsf{head}\ ^{\sigma}\mathsf{next}\ xs\}\\ \quad \textbf{PROC}\ \mathsf{append}('\mathsf{head},'\mathsf{elem})\\ \quad \{\tau.\ \mathsf{inv}_{\mathtt{list}}\ ^{\tau}\mathsf{head}\ ^{\tau}\mathsf{next} \wedge \mathsf{abs\_rel}_{\mathtt{list}}\ ^{\tau}\mathsf{head}\ ^{\tau}\mathsf{next}\ (xs\ @\ [^{\sigma}\mathsf{elem}])\}\end{array}$$

i. e., we express the effect of the function in terms of its abstract representation, which is a concatenation of lists.

Note that our approach to abstract representations of memory coincides with the idea of "ghost fields" attached to C structures in Spec#/VCC [6]. In the pragmatics for Spec#/VCC, one would provide a ghost field to each list-node attaching to it what the node and the pointer structure it points to represent.

3 Microkernel Design

The *kernel* of an operating system is the code that runs in the privileged mode of a processor, i. e., this and only this code has unrestricted access to all hardware resources. Traditionally, kernels provided an abstraction from the hardware processor and the external devices. When kernels grew in size over the years because of a rising variety of hardware, and especially external devices, the traditional systems were called *monolithic* and the idea of smaller *microkernels* was born.

The motivation for microkernels, however, is manifold. Microkernels became a popular research topic in the late 1980s together with the idea of multi-personality operating systems, which demanded a more general hardware abstraction than the traditional approach. This first generation of microkernels such as Mach [41] or IBM's Workplace OS [21] suffered from a poor performance. When Jochen Liedtke analyzed these systems [35], he pinned the problem down to a feature-overloaded mechanism for inter-process communication (IPC). Together with a light-weight, flexible IPC mechanism [33], he proposed the minimality of hardware abstractions [34] in the kernel and suggested that servers implement the traditional services of operating systems. Engler *et al.* [18] took the idea of minimality a step further and banned (nearly) all abstractions from the kernel. Instead of resource management, the kernel was restricted to resource protection. Abstractions were implemented in operating-system libraries and directly linked to the user processes.

Our microkernel design is not as minimal as Engler or Liedtke proposed. We host all functionality in the kernel that would be hard to verify if implemented in user processes. Obeying this principle, the memory management, support for finite IPC timeouts, and the scheduler live in our microkernel. Device drivers, which constitute the largest part of today's monolithic kernels, are implemented outside our microkernel. In the remainder of this section, we describe the basic functionality of our kernel in more detail. Moreover, we regard the design aspect of correctness, and finally, we explain our scheduling policy.

Functionality. Our microkernel VAMOS performs the following tasks: (a) enforcement of a minimal access control, (b) process management, (c) memory management, (d) priority-based round-robin scheduling, (e) support for user-mode device drivers, and (f) inter-process communication (IPC). Processes can control these tasks via the kernel's application binary interface (ABI). Table 1 lists the kernel calls that constitute the ABI.

A minimal access-control mechanism reserves most kernel calls for so-called *privileged processes*. Thus, only a privileged process can bring up new processes or kill existing ones, alter the memory consumption of processes, change their scheduling parameters, or control the registration of device drivers. Any process, however, might use the IPC mechanism. We presume that the privileged processes constitute the user-mode parts of the operating system and implement a more sophisticated access-control mechanism. Non-privileged processes may then communicate with the privileged processes in order to request kernel services on their behalf.

When VAMOS boots, it launches one single process, the *init process*. This process is privileged and has to set up the required servers of the operating system, start and register the device drivers, and possibly run initial applications.

A *device driver* is a user process, which is designated for the communication with certain devices. Only if a process is registered as a driver for a particular device, it may place read or write requests from or to that device, respectively. Moreover, the device driver is notified of interrupts from that device.

Design of a Correct Microkernel. Modern microkernels are strictly evaluated by their performance. While efficiency is clearly essential for the applicability of a microkernel, many implementations even sacrifice perdurability for performance and declare, e. g., a finite range of numbers as "sufficiently large" such that overflows *should* not occur during the expected lifetime of the kernel. An old but even ABI-visible example is Liedtke's [33] *generation*

Table 1 Application binary interface of the VAMOS kernel

Kernel Call	**Description**
Access Control	
SET_PRIVILEGED^p	add a process to the set of privileged processes
Process Management	
PROCESS_CREATE^p	create a new process from a memory image
PROCESS_CLONE^p	copy an already existing process
PROCESS_KILL^p	kill a process
Memory Management	
MEMORY_ADD^p	increase the amount of virtual memory for a process
MEMORY_FREE^p	decrease the amount of virtual memory for a process
Scheduling Mechanism	
CHG_SCHED_PARAMS^p	change scheduling parameters
Device Driver Support	
CHANGE_DRIVER^p	(un)register a process as a driver for a set of devices
ENABLE_INTERRUPTS^d	re-enable a set of interrupts after their successful handling
DEV_READ^d / DEV_WRITE^d	communicate with a certain device
Inter-Process Communication	
IPC_SEND / IPC_RECEIVE	unidirectionally communicate with another process
IPC_REQUEST	send a message and immediately wait for a reply
CHANGE_RIGHTS	manipulate IPC rights
READ_KERNEL_INFO	receive information from the kernel

^{p} call is reserved for privileged processes ^{d} call is reserved for device drivers

counter: In order to ensure a thread's identity, the fixed internal format of thread identifiers reserves some bits for a counter, which is incremented on reincarnation. A possible overflow, however, is just neglected. Although such an approach might be accepted for non-critical systems, it does certainly not suit a critical, verified system; moreover, a restriction on the lifetime contradicts the formal notion of liveness.

As a consequence, our kernel has no such "expiration date". For this to work, we maintain a solely relative notion of time within the kernel. Furthermore, a capability-like management of process identifiers allows for a conceptually infinite name space for identifiers.

Another design decision is related to provable correctness as well: the absence of a so-called *idle process*. In some microkernels, this distinguished process is scheduled with lowest priority and should implement a busy wait. We circumvent assumptions on user-level processes if possible and have thus implemented the idle loop (which solely waits for incoming interrupts) directly in the kernel.

Process Scheduling. The basic policy underlying the scheduler in VAMOS is round-robin process selection. This basic policy, however, can be adjusted by two regulators: priorities and timeslices. Our scheduler supports three different priority levels. Only processes in the highest, non-empty priority class will be scheduled. Processes in a lower class wait until no processes are ready to compute in any higher priority class. Within one priority class, the timeslice determines how long a certain process may compute until it is preempted in favor of another process of the same priority class. Thus, timeslices determine the relative weight of process runtimes while priorities lead to the preemption of lower process classes.

4 The Implementation Layer

This section is devoted to the implementation of our microkernel. Recall that VAMOS is implemented to run as CVM's kernel machine. Certainly, we have to represent this framework in our programming model. Hence, we describe in the first subsection, how we model the effects of CVM in Simpl as seen by a VAMOS programmer. The second subsection briefly introduces the top-level function `kdispatch` of VAMOS, which is called by CVM. Finally, we present the code of the timer-interrupt handler, which constitutes the core component of the VAMOS scheduler.

4.1 Representation of the Framework CVM

Though the CVM layer itself is not implemented in C, we can express its effects in terms of Simpl code. Two CVM components are visible in our programming model: (a) a processor abstraction containing the user processes with separate registers and linear, virtual memory and (b) a device subsystem, which communicates with the processor via memory-mapped I/O. Both components are combined in the global variable $'cvmX$.

The processor abstraction $cvm\_ups\ 'cvmX$ comprises not only the actual user machines $userprocesses$, but additionally a current-process identifier $currentp$ and a status register $statusreg$ storing the currently enabled interrupts. The function $exceptvec\ p$ computes the vector of the exceptions that occur during the next step of a process p. Some exceptions write an exception-data register; its content is computed by $edata\_nat\ p$. We call the device component $cvm\_devs\ 'cvmX$ and define the function $intvec$, which computes the interrupt vector from a state of the device subsystem.

```
procedures kernel_step() =
  'up :==_g cvm_ups 'cvmX;
  IF_g currentp 'up = ⊥ THEN     (* CVM is idle *)
    'eca :==_g statusreg 'up ∧_u intvec (cvm_devs 'cvmX);
    'edata :==_g 0
  ELSE
    'proc :==_g (userprocesses 'up) ⌈currentp 'up⌉;
    'eca :==_g statusreg 'up ∧_u (exceptvec 'proc ∨_u intvec (cvm_devs 'cvmX));
    'edata :==_g edata_nat 'proc;
    'cvmX :==_g ('up (|userprocesses := δ_up (userprocesses 'up) ⌈currentp 'up⌉|),
                 cvm_devs 'cvmX)
  FI;
  IF_g 'eca > 0 THEN     (* has an enabled interrupt been raised? -- Then call kdispatch *)
    'cp :== CALL_g kdispatch ('eca, 'edata );
    'cvmX :==_g ( (cvm_ups 'cvmX)(|currentp := if 'cp ∈ procnumT
                                               then ⌊Abs_procnumT 'cp⌋
                                               else ⊥|),
                  cvm_devs 'cvmX )
  FI
```

Fig. 4 Simpl function `kernel_step`, which represents a combined step of CVM and VAMOS

The pseudo-code of a CVM transition is shown in Fig. 4. This transition consists of up to two phases: First, the current process executes one (assembly) step if existing.[3] Second, the CVM layer computes the vector of enabled interrupts and invokes the kernel's `kdispatch` function if an enabled interrupt has been raised. There are two possible sources of interrupts: The current process may cause an exception, and external devices may raise their interrupt line. Interrupts are ignored when not enabled in the status register. The return value of `kdispatch` describes the process to be run next: If the value is a valid process number, this process is elected, otherwise, the system idles until the next device interrupt occurs.

4.2 The top-level function of VAMOS

Upon kernel entry, the CVM routine calls the function `kdispatch` of VAMOS. Fig. 5 shows the implementation of the function `kdispatch` of our microkernel. This function handles the incoming interrupts that could not be taken care of by CVM. The function takes two parameters: The exception cause `eca`, which is a bit vector of occurred interrupts, and the exception data `edata`, which contains the trap number if a trap has occurred.

As the name `kdispatch` suggests, the function is characterized by a number of case distinctions. We distinguish:

Initialization. After power-up, the processor generates a *reset* interrupt. In this case, the CVM framework sets up its internal data structures (not shown in `kernel_step`) and then passes the interrupt on to the `kdispatch` function. If called with the reset-interrupt bit set, `kdispatch` calls the function `vamos_init`, which initializes the data structures of VAMOS. The remaining interrupt vector is ignored.

Process Exceptions. The current process may cause a number of exceptions during its execution. From the kernel's perspective, there are only two alternatives: a *fatal exception* like an illegal page fault or a *trap*. In the former case, there is no reasonable recover procedure, and thus, the process is simply killed by VAMOS. The semantics of a fatal

[3] Recall that we do not rely on the existence of an idle process. Hence, there might be no current process.

```
procedures kdispatch (eca, edata | res_nat) =
  IF_g ('eca ∧_u 1) ≠ 0 THEN
    'dummy_i :== CALL_g vamos_init()
  ELSE
    'old_cup :=_g 'current_process;
    IF_g ('eca ∧_u UEXCEPT_MASK) ≠ 0 THEN
      'dummy_i :== CALL_g process_kill(HANDLE_SELF)
    ELSE
      IF_g ('eca ∧_u EXCEPT_TRAP) ≠ 0 THEN
        'dummy_i :== CALL_g handle_trap('edata)
      FI
    FI;
    IF_g ('eca ∧_u DEVICE_TIMER_BIT) ≠ 0 THEN
      'dummy_i :== CALL_g handle_timer('old_cup)
    FI;
    IF_g ('eca ∧_u UEXT_INT_MASK) ≠ 0 THEN
      'dummy_i :== CALL_g int_delivery('eca)
    FI
  FI;
  IF_g 'current_process ≠ Null THEN
    'res_nat :=_g 'current_process → 'pid
  ELSE
    'res_nat :=_g 0
  FI
```

Fig. 5 The kernel-dispatcher function of VAMOS

exception is exactly the same as if the process had requested to be killed via the kernel call `process_kill`. Hence, we reuse this function. In case of a trap, the function `trap_handler` is called. This function is essentially a huge case distinction over trap numbers.

Timer Interrupt. Independently from the process exceptions, we check for the timer interrupt, which is passed on to the function `handle_timer`. This function implements the scheduler (see Sect. 4.3).

Device Interrupts. If external devices have raised interrupts, the function `int_delivery` is invoked in order to disable the interrupts and then either immediately deliver the interrupts, if the corresponding device-driver processes are waiting, or schedule the interrupts for later delivery.

The function `kdispatch` determines its return value using the global variable 'current_process, which is a pointer into the process information block (PIB) of the current process. If there is no current process, the pointer is `NULL` and we return 0 (meaning "wait for interrupts"). Otherwise, we retrieve the process identifier (PID) of the current process from the PIB. Once, the function `kdispatch` returns, the CVM framework transfers the CPU to the process identified by the return value.

4.3 The Timer-Interrupt Handler

When a timer interrupt occurs, `kdispatch` calls the function `handle_timer` (see Fig. 6). This function constitutes the heart of our scheduler and performs the following tasks:

- acknowledge the timer interrupt by reading a word from the timer device such that the latter may lower its interrupt line.

```
procedures handle_timer(old_cup | res_int) =
  (* acknowledge the timer interrupt *)
  'dummy :== CALL_g cvm_in_word(DEVICE_TIMER, 0);

  (* detect and handle elapsed IPC timeouts *)
  IF_g 'next_timeout ≠ INFINITE_TIMEOUT THEN
    'current_time :=_g 'current_time + 1
  FI;
  IF_g 'current_time ≥ 'next_timeout THEN
    'dummy_i :== CALL_g check_elapsed_timeouts()
  FI;

  (* charge the process that computed in the last step *)
  IF_g 'old_cup ≠ Null ∧ 'old_cup → 'state = READY_STATE THEN
    IF_g 'old_cup→'consumed_time ≥ 'old_cup→'timeslice THEN
      'ready_lists ! ('old_cup→'priority) :==
            CALL_g queueDelete('ready_lists ! ('old_cup→'priority), 'old_cup);
      'ready_lists ! ('old_cup→'priority) :==
            CALL_g queueAppend('ready_lists ! ('old_cup→'priority), 'old_cup);
      'old_cup→'consumed_time :=_g 0
    ELSE
      'old_cup→'consumed_time :=_g 'old_cup→'consumed_time + 1
    FI
  FI;

  (* select the process that runs in the next step *)
  'dummy_i :== CALL_g search_next_process();
  'res_int :=_g 0
```

Fig. 6 Function `handle_timer`

- increase the current time and check for elapsed timeouts. This check is encapsulated in function `check_elapsed_timeouts`, which traverses the wait queue, wakes up all processes with elapsed timeouts, and subtracts the current time from all other timeouts. Finally, the current time is set to zero. This approach avoids the overflow of the 'current_time variable at relatively low cost.[4]
- charge the process 'old_cup that computed in the last step if it is still ready. If the so far consumed time of 'old_cup is greater than or equal to its timeslice, the process is moved to the end of its ready queue and the consumed time is set to zero. Otherwise, the consumed time is increased by one.
- elect the process that runs in the next step by invoking function `search_next_process`, which returns the first element in the ready queue with the current maximum priority.

5 The Abstract System Layer

At the abstract layer, we describe our computer system by a number of communicating automata. Fig. 7 depicts the situation: Automaton $\mathscr{A}_{V+D}$ specifies the overall system and is composed of the device automaton $\mathscr{A}_D$ and the VAMOS automaton $\mathscr{A}_V$. The former describes

[4] We have compared the described timeout adjustment with an overflow-aware implementation. The latter does not prevent overflows but the check for elapsed timeouts takes overflows into account. To our surprise, the overflow-aware implementation comprises 122 instructions *more* than the timeout adjustment. This overhead is caused by the more involved comparison between the current time and the timeouts. Apparently, the expression evaluation for lazy operators is comparatively costly.

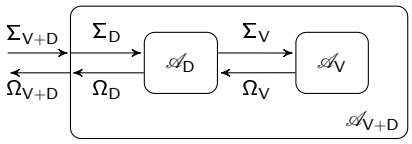

Fig. 7 The input/output automata of the abstract system layer and their relationship

the behavior of the external devices like the keyboard, the timer or the network card while the latter formalizes the behavior of the processor with VAMOS running on it. The automata $\mathscr{A}_D$ and $\mathscr{A}_V$ communicate using the alphabet Σ_V (from the device subsystem to the processor) and the alphabet Ω_V (from the processor to the devices). Our kernel uses memory-mapped I/O for device communication. Hence, the output alphabet Ω_V comprises read and write accesses to device addresses whereas the input alphabet Σ_V consists of interrupt lines and optionally incoming data.

In addition, the devices may interact by an *external interface* with the environment in order to receive keystrokes, for example, or send network packages. We define this interface by the alphabets Σ_D and Ω_D. This external device interface is exposed by the overall system (alphabets Σ_{V+D} and Ω_{V+D}).

We specify VAMOS running on the VAMP by $\mathscr{A}_V = (\mathscr{S}_V, \mathscr{S}_V^0, \Sigma_V, \Omega_V, \omega_V, \delta_V)$ with the state space $\mathscr{S}_V$, the set of initial states $\mathscr{S}_V^0 \subset \mathscr{S}_V$, the input alphabet Σ_V, the output alphabet Ω_V, the output function ω_V, and the transition function δ_V. Likewise, we define $\mathscr{A}_D$ and $\mathscr{A}_{V+D}$. In this article, we focus on $\mathscr{A}_V$ and $\mathscr{A}_{V+D}$. Alkassar and Hillebrand [1] describe the device model in detail.

The states $\mathscr{S}_{V+D}$ of the overall system are pairs of a VAMOS state $s_V \in \mathscr{S}_V$ and a device-system state $s_D \in \mathscr{S}_D$. The input alphabet Σ_{V+D} is used to determine which subsystem takes the next step. It extends Σ_D (indicating a device step) by the value $\perp$ for a processor step. The output alphabets are the same, i. e., $\Omega_{V+D} = \Omega_D$. Note that the output of the device subsystem depends on the transition, i. e., δ_D returns pairs (s_D, w) of a successor state s_D and an output w. Consequently, there is no separate output function for $\mathscr{A}_{V+D}$.

Transitions δ_{V+D} of the overall system inspect the input and distinguish three cases:

1. An *external device transition* is performed, if the input $i \in \Sigma_{V+D}$ contains an input for the device subsystem. The VAMOS component remains constant in this case.
2. A *local kernel transition* is performed if the input is $\perp$ and the VAMOS output $\omega_V\ s_V$ is $\mathsf{idle}_{\Omega V}$.
3. A *kernel-device transition* is performed if it is the processor's turn (overall input $\perp$) and the kernel requests a device communication, i. e., $\omega_V\ s_V = \mathsf{read}_{\Omega V}$ *device port count* or $\omega_V\ s_V = \mathsf{write}_{\Omega V}$ *device port data*. In this case, δ_{V+D} passes the VAMOS output with the transition function δ_D to the device subsystem. In response, the device subsystem delivers an output w, which contains the read data if requested. Using this data, δ_{V+D} finally performs the VAMOS transition δ_V.

In the remainder, we describe the VAMOS automaton in more detail. This automaton specifies the interaction between the kernel and the user processes. User processes are modeled as assembly machines interacting with the kernel via a well-defined interface, the kernel ABI. In analogy to $\mathscr{A}_V$, we specify a process by a self-contained input/output automaton.

The following tuple represents a user process:

$\mathscr{A}_{\mathsf{asm}} = (\mathscr{S}_{\mathsf{asm}}, \mathsf{init}_{\mathsf{asm}}, \Sigma_{\mathsf{asm}}, \Omega_{\mathsf{asm}}, \omega_{\mathsf{asm}}, \delta_{\mathsf{asm}})$

with the state space $\mathscr{S}_{\mathsf{asm}}$, the initialization function $\mathsf{init}_{\mathsf{asm}}$, the input alphabet Σ_{asm}, the output alphabet Ω_{asm}, the output function ω_{asm}, and the transition function δ_{asm}.

$[\![\mathsf{current\_instr}\ s_{\mathsf{asm}} = \text{TRAP } 13;\ \neg\ \mathsf{fatal\_excpt}\ s_{\mathsf{asm}}]\!]$
$\Longrightarrow \omega_{\mathsf{asm}}\ s_{\mathsf{asm}} =$
DEV_READ $(\mathsf{reg2devnum}\ (s_{\mathsf{asm}}.\mathsf{gprs}\ !\ 11))\ (\mathsf{reg2port}\ (s_{\mathsf{asm}}.\mathsf{gprs}\ !\ 12))\ (\mathsf{regs2buffer}\ s_{\mathsf{asm}}\ 13\ 14)$

δ_{asm} ERR_UNPRIVILEGED $s_{\mathsf{asm}} = s_{\mathsf{asm}}$
$(\!|\mathsf{gprs} := s_{\mathsf{asm}}.\mathsf{gprs}[22 := -4],\ \mathsf{dpc} := s_{\mathsf{asm}}.\mathsf{pcp},\ \mathsf{pcp} := (s_{\mathsf{asm}}.\mathsf{pcp} + 4)\ \mathsf{mod}\ 2^{32}|\!)$

Fig. 8 Formal definition of output and transition function of assembly processes for the call DEV_READ

A state $s_{\mathsf{asm}} \in \mathscr{S}_{\mathsf{asm}}$ contains a normal and a delayed program counter (implementing the delayed branch mechanism), a file of general-purpose registers, and a byte-addressable linear memory. The initialization function $\mathsf{init}_{\mathsf{asm}}$ takes a binary program as parameter and determines the according initial process state. The output alphabet Ω_{asm} enumerates all possible kernel calls, a few error conditions, and the output ε_Ω denoting the intention to perform a local computation. The input alphabet Σ_{asm} contains all responses to kernel calls and error conditions as well as the input ε_Σ for a process-local transition.

Fig. 8 depicts the formal definition of the output function ω_{asm} and the transition function δ_{asm}, for a DEV_READ call. We assume that s_{asm} is the state of an assembly process. If the current instruction encodes a TRAP 13 and no fatal exception (e. g., an illegal page fault) occurred, the output function returns DEV_READ with the specified device number (register 11), port number (register 12), and the buffer (registers 12 and 13).

Let us now assume that the kernel recognizes this output but the process is not privileged. In this case, the kernel informs the process of this error by passing ERR_UNPRIVILEGED on to the transition function δ_{asm} of the process. For this input, the transition function updates the result register 22 with -4. Furthermore, the delayed program counter dpc is set to the former value of the normal program counter pcp and pcp is increased by 4 modulo 2^{32}.

After this brief introduction of the user processes, we describe the components of $\mathscr{K}_{\mathsf{V}}$ in more detail. Finally, we formally specify the VAMOS scheduler.

The State Space. A state of the VAMOS automaton is an abstraction of the implementation state. While we avoid redundancy, we resemble the information that is required for the observable kernel behavior.

A state $s_{\mathsf{V}} \in \mathscr{S}_{\mathsf{V}}$ comprises the following components:

$s_{\mathsf{V}}.\mathsf{procs}$ is a partial function mapping process identifiers (PIDs) to their assembly states $s_{\mathsf{asm}} \in \mathscr{S}_{\mathsf{asm}}$.
$s_{\mathsf{V}}.\mathsf{priodb}$ is a partial function mapping PIDs to their priorities.
$s_{\mathsf{V}}.\mathsf{schedds}$ contains the scheduling data structures like wait and ready queues as well as process-specific accounting information like the length of the timeslice.
$s_{\mathsf{V}}.\mathsf{rightsdb}$ contains information for the IPC rights management and the set of privileged processes.
$s_{\mathsf{V}}.\mathsf{sndstatdb}$ are the remains of the process status in the implementation. Usually, the membership in a ready or wait queue determines the current process status—except for one case: An IPC_REQUEST call has a send and a receive phase, which might both require waiting. The send status database $s_{\mathsf{V}}.\mathsf{sndstatdb}$ keeps track of the phase.
$s_{\mathsf{V}}.\mathsf{devds}$ contains data for device communication.

Note, that the partial functions are only defined for the PIDs of the currently active processes.

$$\begin{array}{l}[\![\omega_{\mathsf{asm}}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil\rceil = \text{DEV\_READ}\ \lfloor \mathit{devid}\rfloor\ \lfloor \mathit{port}\rfloor\ \mathit{buffer};\\ \quad \mathit{buffer} \neq \mathsf{BufUndefined};\\ \quad \mathit{buffer} \neq \mathsf{BufUnavailable};\\ \quad \lceil s_{\mathsf{V}}.\mathsf{devds.driver}\ \mathit{devid}\rceil = \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil]\!]\\ \Longrightarrow \omega_{\mathsf{V}}\ s_{\mathsf{V}} = \mathsf{read}_{\Omega\mathsf{V}}\ \mathit{devid}\ \mathit{port}\ (\mathsf{bufLength}\ \mathit{buffer})\end{array}$$

Fig. 9 Formal definition of the output function for the call DEV_READ

The Output Function. The function ω_{V} determines the output in a certain VAMOS state and is used in the function $\delta_{\mathsf{V+D}}$ to figure out whether a device interaction is desired. VAMOS interacts with the device subsystem by read or write requests. A device interaction is initiated, if the output of the currently running process $\omega_{\mathsf{asm}}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil\rceil$ indicates a DEV_READ or DEV_WRITE call, the arguments are valid, and the process is the driver for the specified device. Fig. 9 depicts the formal definition of the output function ω_{V} for the call DEV_READ.

The Transition Function. The transition function δ_{V} takes two parameters: (a) an input d from the device subsystem, which contains data read from a device, and (b) a VAMOS state s_{V}. It returns a successor state s'_{V}. Within a transition, we distinguish up to three phases:

1. If the current process $cp = \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil$ is defined, we consult its output $\omega_{\mathsf{asm}}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ cp\rceil \in \Omega_{\mathsf{asm}}$ and compute the response according to the current VAMOS state. For instance, if a process calls DEV_READ, we check for sufficient privileges and choose the corresponding response $\mathit{res} \in \Sigma_{\mathsf{asm}}$ for success or failure. With this response, we advance the current process by calling its transition function:

   $$s_{\mathsf{V}}(\!|\mathsf{procs} := s_{\mathsf{V}}.\mathsf{procs}(cp \mapsto \delta_{\mathsf{asm}}\ \mathit{res}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ cp\rceil)|\!)$$

   On success, *res* comprises the device data d, and otherwise an error value.
2. If the timer-interrupt line is raised, the timer-interrupt handler is invoked (see Sect. 4.3).
3. Finally, VAMOS delivers the occurred interrupts to the according drivers (assuming that the latter are waiting; otherwise, the interrupts are saved in $s_{\mathsf{V}}.\mathsf{devds}$ for a later delivery).

Each phase is encapsulated in its own specification function. For space restrictions, we cannot present all of them but content ourselves with the specification of the timer-interrupt handler handleTimer. We formally define this function in the following section.

5.1 The Scheduler

On the abstract level, the scheduler is defined by the component schedds of the VAMOS state space and the VAMOS transition function. Below, we describe the data structures. The transition function, however, is far too complex for a full presentation in this article. Hence, we confine ourselves to the core of the scheduler: the timer-interrupt handler.

Data Structures. The component $s_{\mathsf{V}}.\mathsf{schedds}$ is divided into sub components. The current time $\mathsf{time} \in \mathbb{N}$ is a counter for clock ticks. Process-specific scheduling information for active processes is collected in the partial function procdb that maps PIDs to a record of (a) the timeslice tsl, (b) the amount of consumed time ctsl, and (c) the absolute timeout timeout. If a process is found to be computing when a timer interrupt raises, the component ctsl is increased until the process has finally run for tsl ticks. In this case, another process is

```
checkTimeouts s_V ≡ s_V
  (|procs := λx. if expiredTimeout s_V.schedds x
                 then if is_process_sending s_V.procs s_V.sndstatdb x
                      then ⌊δ_asm ERR_SND_TIMEOUT ⌈s_V.procs x⌉⌋
                      else ⌊δ_asm ERR_RCV_TIMEOUT ⌈s_V.procs x⌉⌋
                 else s_V.procs x,
     schedds := s_V.schedds
       (|ready := λp. s_V.schedds.ready p @
                      [x∈s_V.schedds.wait . expiredTimeout s_V.schedds x ∧ ⌈s_V.priodb x⌉ = p],
         wait := [x∈s_V.schedds.wait . ¬ expiredTimeout s_V.schedds x],
         procdb := λx. if expiredTimeout s_V.schedds x then ⌊⌈s_V.schedds.procdb x⌉(|ctsl := 0|)⌋
                       else s_V.schedds.procdb x|),
     sndstatdb := λx. if expiredTimeout s_V.schedds x then ⌊False⌋ else s_V.sndstatdb x|)
```

Fig. 10 Specification of checkTimeouts

scheduled. If a process calls the kernel for IPC and no partner is ready for communication, the absolute timeout timeout is computed from the current time and the relative timeout that has been specified with the call.

Moreover, the scheduler maintains different queues for scheduling. They are represented as finite sequences in the VAMOS specification. Namely, there is a ready queue s_V.schedds.ready *prio* of schedulable processes for each priority *prio*. Additionally, all processes currently not ready to be scheduled (because they are waiting for an IPC partner) are held in the queue s_V.schedds.wait. Inactive ones are stored in s_V.schedds.inactive.

In VAMOS, the current process is the first process in the highest, non-empty ready queue. If all ready queues are empty, the current process is undefined. Or more technically, we concatenate the ready queues from the highest to the lowest and specify the first process in this list as current:

```
v_cup schedds ≡
  case concat (map schedds.ready [high_prio, med_prio, low_prio]) of [] ⇒ ⊥ | x · xs ⇒ ⌊x⌋
```

Handling Timer-Interrupts. The abstract function handleTimer specifies the semantics of the timer-interrupt handling in VAMOS. It increases the current time whereby timeouts of waiting processes might elapse. All processes with elapsed timeouts are woken up and notified about the timeout. Furthermore, the handler charges the current process in order to ensure prioritized fairness in our scheduling algorithm: If the current process has consumed its whole timeslice, it is moved to the end of its ready queue; otherwise, the value of the consumed timeslice is increased.

Following the implementation, we specify handleTimer as composite of two functions checkTimeouts and reschedule, i. e.,

```
handleTimer s_V cp ≡
reschedule (checkTimeouts (s_V(|schedds := s_V.schedds(|time := s_V.schedds.time + 1|)|))) cp
```

The former describes the handling of elapsed timeouts, i. e., it awakes waiting processes with an elapsed timeout and notices the affected processes. The latter balances the timeslice account of the current process.

The formal definition of checkTimeouts is given in Fig. 10. We encapsulate the check for an expired timeout of a waiting process in the predicate expiredTimeout. If the timeout of a user process expires, the process is noticed by an error value. With the error value, we distinguish whether it has been waiting in the send phase (predicate is_process_sending holds) or in the receive phase.

```
reschedule s_V cp ≡
  let procdb = s_V.schedds.procdb ⌈cp⌉; ready = s_V.schedds.ready ⌈s_V.priodb ⌈cp⌉⌉
  in s_V(|schedds := if cp = ⊥ ∨ ⌈cp⌉ ∉ set ready then s_V.schedds
                    else if ⌈procdb⌉.tsl ≤ ⌈procdb⌉.ctsl
                        then s_V.schedds
                             (|ready := s_V.schedds.ready
                                (⌈s_V.priodb ⌈cp⌉⌉ := [x∈ready . x ≠ ⌈cp⌉] @ [⌈cp⌉]),
                                procdb := s_V.schedds.procdb(⌈cp⌉ ↦ ⌈procdb⌉(|ctsl := 0|))|)
                        else s_V.schedds
                             (|procdb := s_V.schedds.procdb(⌈cp⌉ ↦ ⌈procdb⌉
                                (|ctsl := ⌈procdb⌉.ctsl + 1|))|)|)
```

Fig. 11 Specification of reschedule

In addition, we wake up each notified process *pid*: At first, the process is removed from the wait queue and appended to the ready queue according to their priority. At second, it retrieves its full timeslice, i. e., we reset its consumed time $\lceil s_V.\mathsf{schedds.procdb}\ pid\rceil.\mathsf{ctsl}$ to zero. At third, we reset its send status $s_V.\mathsf{sndstatdb}\ pid$ to False: Recall that this value determines whether the process is waiting in the receive phase of an IPC request call after a successful send phase. With the canceled IPC call, this condition cannot hold any longer.

The function reschedule takes two parameters: The resulting state after checkTimeouts, at the one hand, and the identifier *cp* of the current process at kernel entry, at the other hand. Note that the latter is necessary because the current process might change while handling kernel calls (phase 1 of the transition function δ_V) and the VAMOS state does not hold the current process as a distinguished value but rather generates it from the current state of the ready queues. Nevertheless, it is important to charge that process *cp* that actually computed immediately before the kernel entry.

The formal definition of reschedule is given in Fig. 11. Three situations are determined while updating the scheduling data structures:

In the first one, there has been no current process at kernel entry, i. e., $cp = \bot$ or this process *cp* has been rescheduled in the meantime. In this case, the current process has "paid" in the sense that it is not computing anymore. Thus, there is nothing to do but to increase the current time.

In the second situation, the current process *cp* at kernel entry is still ready to compute. In order to preserve the fairness of the scheduling, we check whether *cp* consumed all of its assigned timeslice. If so, we shift *cp* to the end of its ready queue and ensure that the process is not scheduled again until all other processes in the queue had a chance to compute. Technically, we first remove it from the according queue and then append it again. For the next computation phase, *cp* should again get its full timeslice. Hence, we set its consumed time to 0.

In the third situation, the current process *cp* at kernel entry is as well ready to compute but its timeslice is not yet fully consumed. In this case, the consumed time of process *cp* is increased by 1 but it is allowed to proceed with its computation.

6 Refinement

In the last sections, we have introduced the VAMOS implementation and the abstract system model. In this section, we formally relate both layers by a proof of functional correctness. As mentioned in the introduction, the function `kernel_step` of the CVM layer implements together with the VAMOS code a transition δ_{V+D} of the abstract system model. For the

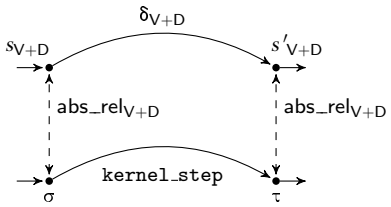

Fig. 12 Kernel step refinement

refinement proof, we link the abstract states s_{V+D}, s'_{V+D} and the implementation states σ, τ via the formally defined abstraction relation $\mathsf{abs\_rel}_{V+D}$.

Fig. 12 visualizes the kernel step refinement whereas the following theorem gives a formal statement:

Theorem 1 (Kernel Correctness) *If the invariant* inv_i *holds for an implementation state* σ*, the abstraction relation* $\mathsf{abs\_rel}_{V+D}$ *correlates* σ *and the abstract state* s_{V+D}*, and the function* `kernel_step` *is executed, then* $\mathsf{abs\_rel}_{V+D}$ *again correlates the resulting implementation state* τ *and the abstract state obtained by* $\delta_{V+D}\ \bot\ s_{V+D}$*, and the invariant* inv_i *holds for* τ*.*

$$\begin{aligned}&\Gamma\vdash \{\sigma.\ \mathsf{abs\_rel}_{V+D}\ \sigma\ s_{V+D} \wedge \mathsf{inv}_i\ \sigma\}\\ &\quad \textbf{PROC}\ \mathsf{kernel\_step}()\\ &\quad \{\tau.\ \mathsf{abs\_rel}_{V+D}\ \tau\ (\mathsf{fst}\ (\delta_{V+D}\ \bot\ s_{V+D})) \wedge \mathsf{inv}_i\ \tau\}\end{aligned}$$

Most notably, the abstraction relation $\mathsf{abs\_rel}_{V+D}$ is the conjunction of two independent abstraction relations for the kernel data structures, on the one hand, and the device subsystem, on the other hand. We introduce $\mathsf{abs\_rel}_{V+D}$ and the implementation invariant inv_i in more detail in the following subsections.

Recall that the transition function δ_{V+D} uses its input parameter to determine whether the kernel or solely the device subsystem (e. g., by receiving a network package) advance during a transition of the overall system. If this input is not $\bot$, the kernel remains constant. Knowing that the abstraction relation of the kernel data structures and the one of the device subsystem are independent, we may consequently disregard changes that are limited to the device subsystem in the consideration of the kernel refinement. While we are interested in the kernel-device interaction, we disregard any external device input from the environment and the according output to it. As a consequence, this input is fixed at $\bot$ in the theorem above. Recall that the transition function δ_{V+D} returns a pair of the updated state and the external output. We are, however, only interested in the updated state, and thus, take the first component of this pair, i. e., $s'_{V+D} = \mathsf{fst}\ (\delta_{V+D}\ \bot\ s_{V+D})$.

Proof (Idea) The correctness of this statement is based on the correctness of all functions applied in the scope of `kernel_step`. Thus, for all functions we have to define and prove lemmata stating their correctness. Eventually, we connect these lemmata in order to prove the correctness of the overall system.

The proof of Theorem 1 mainly relies on the correctness of the function `kdispatch` representing the main function of the VAMOS implementation. The correctness of `kdispatch`, in turn, depends on the correctness of its four different parts: the initialization, the handling of process exceptions, the timer-interrupt handler and the delivery of device interrupts (see Sect. 4.2). □

Due to complexity, we cannot provide deep insights on the correctness of all these parts. We rather confine ourselves in the remaining section to the timer-interrupt handler as an example. Stating the correctness of the function `handle_timer` relies on the abstraction relation and the implementation invariant. Thus, we first sketch both before we deal with the actual correctness of the handler.

6.1 Abstraction Relation

In this section we are concerned with the abstraction relation $\mathsf{abs\_rel}_{\mathsf{V+D}}$. Together with the state invariant inv_i, which we present in Sect. 6.2, it constitutes a crucial part throughout the complete functional verification. A wrong or insufficient formulation would question our overall correctness result as stated in Theorem 1.

The abstraction relation $\mathsf{abs\_rel}_{\mathsf{V+D}}$ relates an implementation state σ with an according abstract state $s_{\mathsf{V+D}}$, where $s_{\mathsf{V+D}}$ is a pair consisting of a VAMOS state s_V and a state s_D of the device system. Formally, the relation is defined as follows:

$$\mathsf{abs\_rel}_{\mathsf{V+D}}\ \sigma\ (s_\mathsf{V},\ s_\mathsf{D}) \equiv \mathsf{abs\_rel}_\mathsf{v}\ \sigma\ s_\mathsf{V} \wedge s_\mathsf{D} = \mathsf{cvm\_devs}\ {}^{\sigma}\mathsf{cvmX}$$

The right-hand side of the conjunction is an equality for the device subsystem states because both, the implementation and the specification, share the same device representation.

Much more effort is necessary to relate the kernel implementation and the abstract VAMOS state s_V. The definition of $\mathsf{abs\_rel}_\mathsf{v}$ comprises several conjunctions that reflect the structure of a VAMOS state. Each conjunction is dedicated to a certain component of s_V relating relevant implementation variables with their abstract counterparts:

$$\begin{aligned}
&\mathsf{abs_rel}_\mathsf{v}\ \sigma\ s_\mathsf{V} \equiv \\
&\quad \mathsf{rel_procs}_\mathsf{v}\ \sigma\ s_\mathsf{V}.\mathsf{schedds.inactive}\ s_\mathsf{V}.\mathsf{schedds.wait}\ s_\mathsf{V}.\mathsf{procs} \wedge \\
&\quad \mathsf{rel_schedds}_\mathsf{v}\ \sigma\ s_\mathsf{V}.\mathsf{schedds} \wedge \\
&\quad \mathsf{rel_priodb}_\mathsf{v}\ \sigma\ s_\mathsf{V}.\mathsf{schedds.inactive}\ s_\mathsf{V}.\mathsf{priodb} \wedge \\
&\quad \mathsf{rel_sndstatdb}_\mathsf{v}\ \sigma\ s_\mathsf{V}.\mathsf{procs}\ s_\mathsf{V}.\mathsf{sndstatdb} \wedge \\
&\quad \mathsf{rel_rightsdb}_\mathsf{v}\ \sigma\ s_\mathsf{V}.\mathsf{schedds.inactive}\ s_\mathsf{V}.\mathsf{rightsdb} \wedge \mathsf{rel_devds}_\mathsf{v}\ \sigma\ s_\mathsf{V}.\mathsf{devds}
\end{aligned}$$

The relations for the following components actually hide functional abstractions: the device data structures, the send status database, the rights database, and the priority database. For space restrictions we cannot provide the formal specification for all of these abstractions but take $\mathsf{rel\_priodb}_\mathsf{v}$ as an example.

For this relation, however, we use some basic abstractions that bridge the implementation and the specification regarding the process identifiers and the priorities. The implementation represents process identifiers and priorities as unsigned 32-bit integers, while the specification uses finite subtypes of natural numbers, namely $\mathsf{procnumT} \equiv \{1..{<}\mathsf{PID\_MAX}\}$ and $\mathsf{prioT} \equiv \{0..{<}3\}$. The function $\mathsf{Abs\_procnumT}$ takes an unsigned 32-bit integer value and computes the corresponding abstract process number. The inverse function is called $\mathsf{Rep\_procnumT}$. For priorities, we have the functions $\mathsf{Abs\_prioT}$ and $\mathsf{Rep\_prioT}$, respectively.

With these basic abstractions at hand, we formally specify:

$$\begin{aligned}
&\mathsf{rel_priodb}_\mathsf{v}\ \sigma\ s_\mathsf{V}.\mathsf{schedds.inactive}\ s_\mathsf{V}.\mathsf{priodb} \equiv \\
&\quad s_\mathsf{V}.\mathsf{priodb} = \\
&\quad (\lambda p.\ \textbf{if}\ p \in \mathsf{set}\ s_\mathsf{V}.\mathsf{schedds.inactive}\ \textbf{then}\ \bot \\
&\qquad \textbf{else}\ \lfloor \mathsf{Abs_prioT}\ ({}^{\sigma}\mathsf{priority}\ ({}^{\sigma}\mathsf{pib}\ !\ \mathsf{Rep_procnumT}\ p)) \rfloor)
\end{aligned}$$

The component $s_\mathsf{V}.\mathsf{priodb}$ defines a partial mapping between process identifiers and priorities. Inactive processes are not assigned with any priority but with $\bot$. Processes are inactive

iff they are in the inactive queue s_V.schedds.inactive. For active processes we have to explore the priority as assigned in the implementation. Thereby, the crucial point is to obtain the pointer corresponding to process p. Pointers to the process information blocks of all user processes are collected in the list $^\sigma$pib. By design we know, that the corresponding pointer to process p is stored at the index, which we get by converting the process number p to a natural number. Based on the pointer, the heap function $^\sigma$priority delivers the priority, which finally is abstracted by Abs_prioT.

The relation rel_priodb$_v$ relies on the inactive queue, which is obtained by abstracting the inactive list in the implementation. The implementation describes a list by a pointer *head* to the first element of the list. A list can be traversed by means of the previous and next pointers provided by the functions *prev* and *next*. In contrast, the model solely talks about lists of process numbers. We base our abstraction of these process queues on predicate Queue:

Queue *head next prev pid Ls* ≡
 ∃*Rs lst*.
  (Path *head next* Null *Rs* ∧ Path *lst prev* Null (rev *Rs*)) ∧
  map (Abs_procnumT ∘ *pid*) *Rs* = *Ls*

where Path *h n e l* tests whether the list *l* can be constructed starting with element *h* and collecting further elements by recursively applying function *n* until element *e* is reached. The formal definition of Path is:

Path *x h y* [] = (*x* = *y*)
Path *x h y* (*p* · *ps*) = (*x* = *p* ∧ *x* ≠ Null ∧ Path (*h x*) *h y ps*)

The first two conjunctions of Queue specify a doubly-linked list *Rs* of references using predicate Path. The last conjunction states that the list *Ls* of process numbers can be constructed from list *Rs* by applying the function Abs_procnumT ∘ *pid* to each element. Using this predicate, we abstract the inactive list from the implementation variables $^\sigma$inactive_list, $^\sigma$queue_next, $^\sigma$queue_prev, and $^\sigma$pid. By design we know, that the variables specify a doubly-linked list of references representing the list of waiting processes. Thus, in conjunction with the abstraction relation we can formulate the following statement:

⟦rel_schedds$_v$ σ s_V.schedds; Path $^\sigma$inactive_list $^\sigma$queue_next Null *Ps*;
 Path *lst* $^\sigma$queue_prev Null (rev *Ps*)⟧
⟹ Queue $^\sigma$inactive_list $^\sigma$queue_next $^\sigma$queue_prev $^\sigma$pid s_V.schedds.inactive

In an analogous way, we can abstract the other lists, which cover big parts of the scheduling data structures. Nevertheless, abstracting the remaining parts – the time and the process-specific scheduling data – is slightly more interesting, caused by the diverse notion of time.

On the abstract level we specify points in time as unbounded natural numbers while the implementation uses unsigned 32-bit integers to represent those times.

Thus, the specification reflects the abstract idea of a monotonic increasing time while the implementation employs the fact that only a finite range of time points is relevant at a certain execution step. Consequently, there is a growing offset between the implementation time and the abstract time. We formulate this fact in the following statement:

rel_schedds$_v$ σ s_V.schedds ⟹
∃*n*. $^\sigma$current_time + *n* = s_V.schedds.time ∧
 (∀*p*. *p* ∉ set s_V.schedds.inactive ∧ *p* ∈ set s_V.schedds.wait ⟶
  abs_timeout ($^\sigma$timeout ($^\sigma$pib ! Rep_procnumT *p*)) += *n* =
  ⌈s_V.schedds.procdb *p*⌉.timeout)

A natural number *n* defines the offset of the implementation variable $^\sigma$current_time and the component time in the scheduling data structures of the model. This same number is the offset between the (absolute) timeouts of the non-inactive waiting processes.

The implementation stores timeouts as unsigned 32-bit integers, where the maximal value represents an infinite timeout. In the specification, we use the predefined Isabelle/HOL type inat that extends the natural numbers by the value ∞. We have defined the function abs_timeout to convert the implementation value into the abstract one. For abstract timeouts, we defined the operation $+=$ in order to increment a timeout by a natural number.

The processes are also connected via an abstraction relation. We remember, that the implemented system stores the interrupt information in the exception-cause and exception-data registers and performs a step of the current process, by applying δ_{up}, before entering the kernel (cf. Sect. 4). In contrast, the model does not save the information regarding interrupts, but rather computes it indirectly via ω_{asm}, if needed. Thus, in order to get the proper output, the process in the model must be kept in the original state until (possible) interrupts are completely handled. If the occurred interrupts, e. g., a trap, can be handled within one kernel phase, the user step in the model is performed within δ_{V+D}, and when leaving the kernel, both machines are synchronous again. A different situation occurs, if processes have to wait while performing an IPC operation with no suited partner at hand. In this case, they wait until a partner appears or the timeout for the operation expires. Thus, while the process in the implementation already performed a step, the model keeps the original state and thereby has the possibility to explore the pending operation. Consequently, it might happen, that the states of waiting processes in the implementation and in the model differ over several kernel phases.

For clarification: We know about waiting processes, that they eventually have been the current process performing a trap instruction without fatal errors. Otherwise, no IPC operation would be possible.

The semantics of δ_{up} in case of a trap instruction without fatal errors is given by solely increasing the program counters. Thus, the relation between waiting processes in the implementation and the ones in the model is defined as follows:

$$
\begin{aligned}
&[\![\mathsf{rel\_procs_v}\ \sigma\ s_V.\mathsf{schedds.inactive}\ s_V.\mathsf{schedds.wait}\ s_V.\mathsf{procs};\\
&\quad x \notin \mathsf{set}\ s_V.\mathsf{schedds.inactive};\\
&\quad x \in \mathsf{set}\ s_V.\mathsf{schedds.wait};\\
&\quad \exists i.\ \mathsf{current\_instr}\ \lceil s_V.\mathsf{procs}\ x\rceil = \textsc{trap}\ i]\!]\\
&\Longrightarrow \lceil s_V.\mathsf{procs}\ x\rceil(\!|\mathsf{dpc} := \lceil s_V.\mathsf{procs}\ x\rceil.\mathsf{pcp},\ \mathsf{pcp} := (\lceil s_V.\mathsf{procs}\ x\rceil.\mathsf{pcp} + 4)\ \mathsf{mod}\ 2^{32}|\!) =\\
&\quad (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ x
\end{aligned}
$$

Waiting processes x are performing an IPC operation which was triggered by a trap instruction of type $\textsc{trap}\ i$. In the model, the state of a non-inactive but waiting process x is that one, that, after increasing the program counters, again corresponds with the implementation state of process x.

States of waiting processes have to be abstracted in a relational manner because the increasing of the counters is not invertible.

In contrast, for non-waiting processes the functional abstraction is possible. Inactive ones are assigned with $\perp$ whereas all others are mapped one-to-one.

6.2 Implementation Invariant

This section deals with the second main ingredient of the correctness theorem: the state invariant.

Encapsulated in the predicate $\mathsf{inv_i}$, it establishes validity requirements on implementation states. These requirements reach from ordinary range restrictions of values to those affecting the concrete implementation design. Previous statements like “we know by de-

sign" must be discharged by the invariant. Thus, similar to $\mathsf{abs\_rel}_{V+D}$ a careful and well-considered formulation of inv_i is essential.

Formally, the invariant is defined as:

$$\mathsf{inv}_i\ \sigma \equiv \\ \quad \mathsf{validProcs}\ \sigma \wedge \mathsf{validRanges}\ \sigma \wedge \mathsf{validRights}\ \sigma \wedge \mathsf{validPIB}\ \sigma \wedge \mathsf{validLists}\ \sigma \wedge \mathsf{validDevds}\ \sigma$$

where σ represents an implementation state. With some exemplary excerpts, we communicate the flavor of the invariant. Space restrictions and complexity do not allow the complete presentation.

Predicate validProcs states the validity of the user processes. It ensures, for instance, that each process has 32 general purpose registers and that register values fit into 32 bits. Ranges of the kernel variables are fixed through validRanges. Thus, the process identifiers in the implementation are restricted to the range between 0 and PID_MAX. Furthermore, the length of the list keeping the pointers to the process information blocks equals the number of processes, i. e., PID_MAX, and the time value is smaller than MAX_TIME.

$$\mathsf{validRanges}\ \sigma \Longrightarrow \\ (\forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ 0 < {}^{\sigma}\mathsf{pid}\ p < \mathsf{PID\_MAX}) \wedge \\ \mathsf{length}\ {}^{\sigma}\mathsf{pib} = \mathsf{PID\_MAX} \wedge {}^{\sigma}\mathsf{current\_time} < \mathsf{MAX\_TIME}$$

Recall that in the abstraction relation $\mathsf{rel\_priodb}_v$, we knew by design, that the corresponding pointer to process p is stored at the index of ${}^{\sigma}\mathsf{pib}$, which we get by converting the process number p to a natural number. This fact can be derived from predicate validPIB and the properties derived above:

$$[\![\mathsf{validPIB}\ \sigma;\ \forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ 0 < {}^{\sigma}\mathsf{pid}\ p < \mathsf{PID\_MAX};\ \mathsf{length}\ {}^{\sigma}\mathsf{pib} = \mathsf{PID\_MAX}]\!] \\ \Longrightarrow \forall p.\ \exists x{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ {}^{\sigma}\mathsf{pib}\ !\ \mathsf{Rep\_procnumT}\ p = x$$

To be right with $\mathsf{rel\_priodb}_v$, let us consider the validity statements regarding the inactive list and lists in general. Statements regarding the lists are subsumed by the predicate validLists. Mainly, they are characterized through the relation between the state of a process and its list membership. Hence, for the inactive list this means, that processes are only in the inactive list iff their state is also set to inactive:

$$[\![\mathsf{validLists}\ \sigma;\ \mathsf{Path}\ {}^{\sigma}\mathsf{inactive\_list}\ {}^{\sigma}\mathsf{queue\_next}\ \mathsf{Null}\ \mathit{Inact}; \\ \mathsf{Path}\ \mathit{lst}\ {}^{\sigma}\mathsf{queue\_prev}\ \mathsf{Null}\ (\mathsf{rev}\ \mathit{Inact})]\!] \\ \Longrightarrow \forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ ({}^{\sigma}\mathsf{state}\ p = \mathsf{INACTIVE\_STATE}) = (p \in \mathsf{set}\ \mathit{Inact})$$

We can derive similar statements for the wait and ready queues.

The predicates validRights and validDevds tie down the design regarding the rights and device data structures. Both data structures are not relevant for this paper, thus, we leave out any further details.

6.3 Implementation Correctness of the Timer-Interrupt Handler

Stating the correctness of the timer-interrupt handler does not rely on the complete abstraction relation $\mathsf{abs\_rel}_{V+D}$, due to the fact that occurring device steps can be moved to the overall step function δ_{V+D}. To reflect the situation before calling the timer-interrupt handler, we even have to adjust $\mathsf{abs\_rel}_v$ regarding the user processes. If a timer interrupt occurs, the timer-interrupt handler is either called after the trap handler or directly after entering the kernel. Both situations have different impact on the user processes, especially regarding the one of the current process. To deal with this, we define a relation $\mathsf{rel\_procs}_T$ as a more fine-grained process abstraction than provided by $\mathsf{rel\_procs}_v$. The resulting abstraction relation $\mathsf{abs\_rel}_T$ is equal to $\mathsf{abs\_rel}_v$ with $\mathsf{rel\_procs}_T$ replacing $\mathsf{rel\_procs}_v$.

In order to show the functional correctness of the timer-interrupt handler, we start with the formal correctness of the function `check_elapsed_timeouts`.

Theorem 2 (Correctness of the `check_elapsed_timeouts` routine) *The semantic effects of the function* `check_elapsed_timeouts` *are described by the abstract function* checkTimeouts.

$$\begin{array}{l}\Gamma\vdash \{\sigma.\ \mathsf{abs\_rel_T}\ \sigma\ s \wedge \mathsf{inv_i}'\ \sigma \wedge \mathsf{pre}\ \sigma\}\\ \quad {}'\mathsf{res\_int} :== \textbf{PROC}\ \mathsf{check\_elapsed\_timeouts}()\\ \quad \{\tau.\ \mathsf{abs\_rel_T}\ \tau\ (\mathsf{checkTimeouts}\ s) \wedge \mathsf{inv_i}'\ \tau \wedge \mathsf{post}\ \sigma\}\end{array}$$

Note, the function `check_elapsed_timeouts` is applied after increasing the time in function `handle_timer`. This may lead to a situation where, e. g., the time is no longer smaller than MAX_TIME. Thus, in the precondition we can only assume a weaker state invariant $\mathsf{inv_i}'$, where among others, the time-related properties of $\mathsf{inv_i}$ are extracted. However, the predicate pre keeps track of such extracted and adjusted properties as well as of properties implicitly given by the implementation.

In the postcondition this part is taken over by the predicate post. For instance, $\mathsf{inv_i}'$ does not fulfill the complete validity requirements for ready queues. Actually, the head of the ready list of the highest priority determines the current process. Waking up processes, as it happens in `check_elapsed_timeouts`, could violate this property, because the highest priority might change. This violation is not resolved to the point where function `search_next_process` is called in `handle_timer`. In the meantime, predicate post keeps track on this situation.

Proof The main ingredient for this proof is the formulation of the loop-invariant for the iteration over the wait queue, which resembles the filter function over the abstract lists. The invariant itself is a rather technical detail, which we do not elaborate on in this paper. Once we found this invariant, the proof was straight forward. □

Using the correctness result of this routine, we can finally prove the correctness of the function `handle_timer`:

Theorem 3 (Correctness of the Timer-Interrupt Handler) *The semantic effects of the function* `handle_timer` *are described by the abstract function* handleTimer.

$$\begin{array}{l}\Gamma\vdash \{\sigma.\ \mathsf{abs\_rel_T}\ \sigma\ s \wedge \mathsf{inv_i}\ \sigma\}\\ \quad {}'\mathsf{res\_int} :== \textbf{PROC}\ \mathsf{handle\_timer}({}'\mathsf{old\_cup})\\ \quad \{\tau.\ \textbf{let}\ \mathit{pid} = \textbf{if}\ {}^{\sigma}\mathsf{old\_cup} = \mathsf{Null}\ \textbf{then}\ \bot\ \textbf{else}\ \lfloor\mathsf{Abs\_procnumT}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{old\_cup})\rfloor\\ \qquad \textbf{in}\ \mathsf{abs\_rel_T}\ \tau\ (\mathsf{handleTimer}\ s\ \mathit{pid}) \wedge \mathsf{inv_i}\ \tau\}\end{array}$$

Proof The proof of this function involves at the one hand, a distinction over the three cases distinguished in the implementation. Hence, we show that the scheduling policy is correctly implemented with regard to the specification function reschedule. At the other hand, we have to establish the precondition for `check_elapsed_timeouts`. With this prerequisite in place, we use its specification checkTimeouts to establish the claim of our theorem. Additionally, the state invariant $\mathsf{inv_i}$ is again recovered. □

Together with the remaining steps of `kdispatch` and δ_{V+D} we derive $\mathsf{abs\_rel_{V+D}}$ and finally prove Theorem 1.

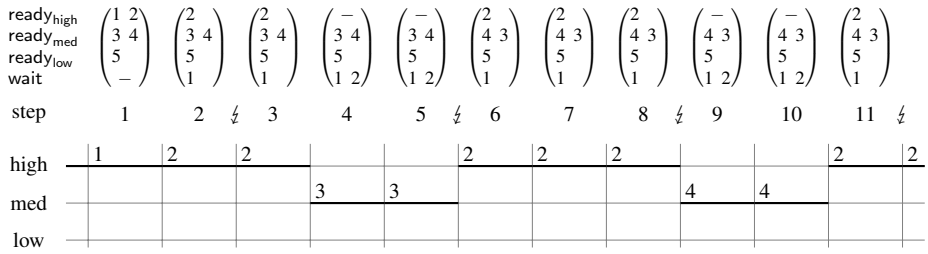

Fig. 13 An exemplary execution trace of VAMOS together with its scheduling data structures

7 A Temporal Property: Scheduler Fairness

In this section, we extend the formal VAMOS model for the reasoning about temporal behavior, we develop the notion of *prioritized fairness* and finally prove that our scheduler conforms with this definition.

At first, however, let us consider an example trace as shown in Fig. 13. The matrices at the top represent the ready queues and the wait queue of VAMOS. Our scheduler elects always the first process in the highest, non-empty ready queue for computation. In the first shown step, two processes are in the highest priority class, and process 1 is computing (as shown at the bottom) because it is the first in the highest ready queue. The elected process runs until it generates an exception or an external interrupt occurs. In the example, we assume that process 1 issues an IPC call and then has to wait for a suitable partner such that process 2 is computing in the second step.

A timer interrupt occurs (indicated by $\lightning$) and process 2 is charged for its computation in step 2, i. e., the counter $\lceil s.\mathsf{schedds.procdb}\ (\mathsf{Abs\_procnumT}\ 2)\rceil.\mathsf{ctsl}$ is increased. Process 2, however, is the only one in its ready queue and continues to compute in step 3. Now, we assume that process 2 issues an open IPC receive and is found waiting in step 4 while process 3 computes. In step 5, a timer interrupt occurs, which charges process 3. For this example, we assume that all processes have a single timeslice—hence, process 3 will be preempted as its timeslice ran out. Moreover, we assume that a timeout wakes up process 2. Now, process 2 continues to compute until, e. g., it issues again an IPC call and starts waiting in step 8. We will later return to this example.

There are various notions of fairness. In the operating-system community, fairness of static-priority scheduling is typically measured by the ratio of the CPU usage among equally prioritized tasks. Fagin and Williams [19] have illustrated this quantitative fairness notion by the *carpool problem*. We, in contrast, formalize the liveness property of strong fairness according to the taxonomy of Francez [23]. Certainly, it would be desirable to predict at least the waiting time of a certain process until it finally computes but this task is hard to achieve.

In principle, the waiting time of the most prioritized processes is linear and an upper bound for the waiting time could be given. In general, however, the waiting time of the processes necessarily depends on the computations of all more prioritized processes. In theory, this problem might be solvable by static program analysis [24], which reliably predicts an upper bound for the worst-case execution time of programs in real-time operating systems [30] with static scheduling. In practice, the predicted time bound or the complexity of the predicting computation would be farcically high in our setting.

Recall, for example, that the paging mechanism is built into our kernel. The execution of a *single instruction* on the VAMP can involve up to two memory accesses at independent addresses, namely, an instruction fetch and a data load or store. Each memory access in the user mode might trigger a page fault, which is handled in *more than 14,000 instructions*. Similarly, a cache access in hardware is by orders of magnitude faster than the direct access of the physical memory in case of a cache miss.

A static analysis can either assume that every user-mode instruction involves about 28,000 instructions in system mode, or it has to regard the whole computer system from the hardware including caches and the timer device via the kernel implementation up to the user processes at once in all its detail. Measurements [47,40] produce considerably lower time bounds but are not exhaustive and might thus miss an unlikely worst case. Our approach, on the contrary, abstracts from the implementation, focuses on the kernel behavior and is thus limited to a liveness property.

Besides the difficulty to prove a quantitative fairness property, the temporal property simply suffices for our original motivation: The here presented work is part of a larger proof effort that establishes the formal foundation of a verification environment for concurrent programs [16]. In this context, it is only relevant that a single process under concern will eventually progress in the concurrent environment. Even total correctness in the concurrent environment can be established without a detailed timing analysis.

Liveness properties are defined over infinite transition sequences, or *traces*. We regard only the VAMOS processor model $\mathscr{A}_{\mathsf{V}}$, here, because the kernel does not rely on a specific device system $\mathscr{A}_{\mathsf{D}}$. We represent the traces as two functions *states* and *inputs* that map the step number to the current state $s_{\mathsf{V}} \in \mathscr{S}_{\mathsf{V}}$ and to the input $i \in \Sigma_{\mathsf{V}}$ for the next step, respectively.

Definition 1 (Trace) Two functions *states* and *inputs* describe a trace iff they meet the following conditions:

$$\mathit{states}\ 0 \in \mathscr{S}^0_{\mathsf{V}}$$
$$\forall n.\ \mathit{states}\ (n+1) = \delta_{\mathsf{V}}\ (\mathit{inputs}\ n)\ (\mathit{states}\ n)$$

In order to reason about the behavior of a trace, we need an invariant $\mathsf{inv}_{\mathsf{a}}$ over the abstract-state sequences. The complete invariant is quite elaborate, hence, we only sketch two queue-related properties, which we will refer to later on in the fairness proof:

- An active process is either in the wait or in the ready queue, and an inactive process is in neither queue. Formally:

  $\mathsf{inv}_{\mathsf{a}}\ s \Longrightarrow$
  if $s.\mathsf{procs}\ p = \bot$ **then** $p \notin \mathsf{set}\ s.\mathsf{schedds.wait} \wedge (\forall i.\ p \notin \mathsf{set}\ (s.\mathsf{schedds.ready}\ i))$
  else $(p \in \mathsf{set}\ s.\mathsf{schedds.wait}) \neq (p \in \mathsf{set}\ (s.\mathsf{schedds.ready}\ \lceil s.\mathsf{priodb}\ p \rceil))$

- A waiting process *pid* is in the process of an IPC operation, i. e.,

  $[\![\mathsf{inv}_{\mathsf{a}}\ s;\ \mathit{pid} \in \mathsf{set}\ s.\mathsf{schedds.wait}]\!] \Longrightarrow \mathsf{is\_ipc}\ (\omega_{\mathsf{asm}}\ \lceil s.\mathsf{procs}\ \mathit{pid} \rceil)$

We show that the constant $\mathsf{inv}_{\mathsf{a}}$ is indeed an invariant over the abstract traces:

Theorem 4 (Abstract invariant) *Predicate* $\mathsf{inv}_{\mathsf{a}}$ *is an invariant over traces, formally:* $\mathsf{inv}_{\mathsf{a}}\ (\mathit{states}\ m)$ *holds for arbitrary* m.

Proof We prove this statement by induction. At first, we establish $\mathsf{inv}_{\mathsf{a}}$ for the initial states: $s_{\mathsf{V}} \in \mathscr{S}^0_{\mathsf{V}} \Longrightarrow \mathsf{inv}_{\mathsf{a}}\ s_{\mathsf{V}}$. At second, we show that $\mathsf{inv}_{\mathsf{a}}$ is maintained by the transition function, i. e., $\mathsf{inv}_{\mathsf{a}}\ s_{\mathsf{V}} \Longrightarrow \mathsf{inv}_{\mathsf{a}}\ (\delta_{\mathsf{V}}\ i\ s_{\mathsf{V}})$. Finally, we derive our claim $\mathsf{inv}_{\mathsf{a}}\ (\mathit{states}\ m)$ by the definition of traces. □

Based on Definition 1, an unconditioned fairness property is:

$$\exists l{\geq}k.\ \mathsf{progress}\ ((\mathit{states}\ l).\mathsf{procs}\ \mathit{pid})\ ((\mathit{states}\ (l+1)).\mathsf{procs}\ \mathit{pid})$$

where the predicate $\mathsf{progress}\ p\ q$ holds iff q can be produced by a transition from p, i. e., $q = \delta_{\mathsf{asm}}\ i\ p$.[5] This statement, however, is too strong for the VAMOS scheduler because it neglects several aspects of our system:

- Processes are dynamic entities and PIDs may be reused. Thus, the process *pid* in step l might not be the same as the one in step k.
- A process *pid* might start an IPC operation with an infinite timeout. If there is never another process willing to communicate with process *pid*, the latter starves.
- Prioritization leads to the preemption of less prioritized process classes – without any time bound (according to our scheduling policy as described in Sect. 3).
- The implementation relies on a live timer device because the scheduler is activated only by the timer interrupt (see Sect. 4.3).

Consequently, we have to weaken our notion of fairness in this context. At first, we recall that liveness conditions are formulated over infinite traces. Hence, we exclude all terminating processes, i. e., $\forall n{\geq}k.\ (\mathit{states}\ n).\mathsf{procs}\ \mathit{pid} \neq \bot$, where *pid* is an arbitrary, fixed process ID.

At second, we assume a predicate $\mathsf{pending\_infinite\_ipc}\ s\ \mathit{pid}$, which examines a state s and holds iff the process *pid* is pending in an IPC operation with an infinite timeout. If a process forever remains pending in this state, it is certainly not the fault of the scheduler but a programming or protocol error. Thus, we do not consider such starving processes, i. e., $\forall n{\geq}k.\ \exists m.\ n \leq m \wedge \neg\ \mathsf{pending\_infinite\_ipc}\ (\mathit{states}\ m)\ \mathit{pid}$.

At third, the priority of a process is runtime-configurable by the kernel call CHG_SCHED_PARAMS. Upon a priority change, the process will be removed from its old priority's ready queue and *appended* to the new one. As long as the priority is only configured at the beginning (or more precisely: changed only a finite number of times), we can find an infinite subtrace without a priority change. If, however, there are infinitely many changes, the process may starve. We exclude this case because it contradicts the concept of static-priority scheduling, and state: $\forall n{\geq}k.\ (\mathit{states}\ (n+1)).\mathsf{priodb}\ \mathit{pid} = (\mathit{states}\ n).\mathsf{priodb}\ \mathit{pid}$.

At fourth, we assume that the timer device produces infinitely often timer interrupts as part of the inputs, i. e., $\forall n{\geq}k.\ \exists m{\geq}n.\ \mathsf{is\_timer\_on}\ (\mathit{inputs}\ m)$. Note that this condition is our only assumption about the device system $\mathscr{A}_{\mathsf{D}}$.

So far, our restrictions have been evident adjustments to the considered system. The remaining problem, which regards priorities, is somewhat more involved. Let us consider a process *pid* in the highest priority class, i. e., $(\mathit{states}\ k).\mathsf{priodb}\ \mathit{pid} = \lfloor \mathsf{high\_prio} \rfloor$. From the assumptions above, we can show that this process *pid* will eventually make progress, i. e.,

$$\exists l{\geq}k.\ \mathsf{progress}\ ((\mathit{states}\ l).\mathsf{procs}\ \mathit{pid})\ ((\mathit{states}\ (l+1)).\mathsf{procs}\ \mathit{pid})$$

This formulation of fairness, however, states nothing about the processes residing in lower priority classes. A statement over fairness for the latter necessarily depends on the (intermittent) absence of a process in a higher priority class, which is ready to compute. All processes, which are willing to compute, are enqueued in the ready queue of its priority. Thus, we can determine the absence of a prioritized, ready process by examining the according ready queues, i. e., process *pid* has the maximal priority in state s if

[5] Technically, the matter is complicated by the fact that the amount of process memory is held with the processes. Transitions changing only this amount, are certainly not considered as progress.

$\forall prio > \lceil s.\mathsf{priodb}\ pid \rceil.\ s.\mathsf{schedds.ready}\ prio = []$

We encapsulate this property in a predicate $\mathsf{has\_maxprio}\ s\ pid$.

In order to extend our fairness statement to all processes, we examine our exact scheduling mechanism more carefully: The currently computing process *cup* is charged by increasing its consumed time $\lceil s.\mathsf{schedds.procdb}\ cup \rceil.\mathsf{ctsl}$ (and eventually preempted) if *and only if* the timer interrupt occurs. In other words, a process must have the maximal priority infinitely often *while* the timer interrupt occurs, i. e.,

$\forall n \geq k.\ \exists m \geq n.\ \mathsf{has\_maxprio}\ (states\ m)\ pid \wedge \mathsf{is\_timer\_on}\ (inputs\ m)$

Note that this assumption implies timer liveness.

The execution trace shown in Fig. 14 illustrates this problem: In step 8, process 2 computes and issues, say, an IPC receive. When the timer interrupt occurs, the scheduler recognizes that process 2 has been computing at the last step. Process 4 starts to compute in step 9 and sends a message to process 2 in step 10, which causes the latter to wake up. Thus, process 2 is found computing when the timer arrives in step 11. If the sequence of events between step 8 and 11 continues, process 4 will never get charged and process 3 starves although it has frequently the maximal priority.

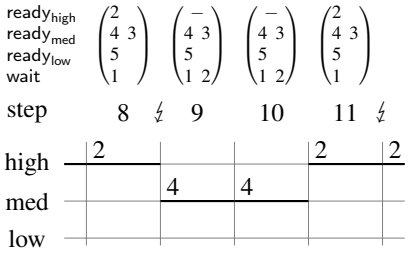

Fig. 14 A VAMOS trace with a race condition

Of course, it is possible to change the scheduling mechanism such that all processes found running in between two timer interrupts will be charged. This code change, however, would require extra bookkeeping during the IPC path, which is the most critical one for system performance. Most L4 kernels even completely skip the scheduler for performance reasons after some IPC operations [42, 17]. Hence, we favored the shorter path over an optimized scheduling.

It is crucial to understand the implications of this design choice, in particular, the possible exploits by malicious processes and the mechanisms to its prevention by prioritized processes. For an active exploit, the user process needs a clock of higher precision than the timer tick, which can only be achieved by another external timer device. If we exclude this unlikely setting, the process cannot predict the time when the tick arrives and the consistent recurrence of the race condition becomes exponentially unlikely.

An informal statement of likeliness, however, is not satisfactory to an assumption of a formal proof. Hence, we examine the necessary preconditions for the race condition. There are three possibilities for a switch to higher priority between timer ticks: (a) A process of higher priority might be created, (b) the scheduling priority of an existing process might be raised, or (c) an IPC call might resume a prioritized, waiting process. The former two are caused by kernel calls reserved for privileged processes. These processes are typically most prioritized and have to be trusted anyways. The latter requires the collaboration of the prioritized process. We regard this situation as a special form of priority inheritance [42].

Concluding, we state:

Corollary 1 (Prioritized Fairness) *The* VAMOS *scheduler is fair with respect to priority classes, i. e., at all suffixes of an infinite trace, all processes will eventually make progress, iff they (a) never terminate, (b) remain forever at the same priority class, (c) do not starve in*

an IPC operation with an infinite timeout, and (d) have infinitely often the maximal priority while the timer interrupt is raised.

We formalize this fact as:

$$\begin{array}{l}[\![(*\ \mathsf{trace}\ *)\ \mathit{states}\ 0 \in \mathscr{S}_{\mathsf{V}}^{0};\ \forall n.\ \mathit{states}\ (n+1) = \delta_{\mathsf{V}}\ (\mathit{inputs}\ n)\ (\mathit{states}\ n);\\ \quad (*\ \ \mathsf{a}\ \ *)\ \forall n{\geq}k.\ (\mathit{states}\ n).\mathsf{procs}\ \mathit{pid} \neq \bot;\\ \quad (*\ \ \mathsf{b}\ \ *)\ \forall n{\geq}k.\ (\mathit{states}\ (n+1)).\mathsf{priodb}\ \mathit{pid} = (\mathit{states}\ n).\mathsf{priodb}\ \mathit{pid};\\ \quad (*\ \ \mathsf{c}\ \ *)\ \forall n{\geq}k.\ \exists m{\geq}n.\ \neg\ \mathsf{pending\_infinite\_ipc}\ (\mathit{states}\ m)\ \mathit{pid};\\ \quad (*\ \ \mathsf{d}\ \ *)\ \forall n{\geq}k.\ \exists m{\geq}n.\ \mathsf{has\_maxprio}\ (\mathit{states}\ m)\ \mathit{pid} \wedge \mathsf{is\_timer\_on}\ (\mathit{inputs}\ m)[\![\\ \Longrightarrow \exists l{\geq}k.\ \mathsf{progress}\ ((\mathit{states}\ l).\mathsf{procs}\ \mathit{pid})\ ((\mathit{states}\ (l+1)).\mathsf{procs}\ \mathit{pid})\end{array}$$

Proof We prove the theorem by case distinction. From the invariant $\mathsf{inv}_{\mathsf{a}}$ we know that a process *pid* can either be inactive, waiting, or ready. The first case immediately contradicts Assumption (a).

If a process is waiting, we can infer from the invariant $\mathsf{inv}_{\mathsf{a}}$ that this process is in the process of an IPC operation, i. e., $\mathsf{is\_ipc}\ (\omega_{\mathsf{asm}}\ \lceil s.\mathsf{procs}\ \mathit{pid}\rceil)$ and that there is no suitable partner ready for communication.

Let us assume that the process forever remains in the wait queue. If the process has issued an infinite IPC call, we have a contradiction with Assumption (c). If, however, a finite timeout was specified with the call, the timeout will eventually elapse. We know this because of timer liveness and the fact that `handle_timer` is invoked whenever the timer interrupt occurs (by `kdispatch`, see Fig. 5). Upon each method invocation, `handle_timer` increases the variable `current_time` and checks for elapsed timeouts (as specified in handleTimer, see Sect. 5.1). Once, the timeout is elapsed, the process is dequeued from the wait queue (as specified in checkTimeouts, see Fig. 10).

Thus, we know that the process cannot remain forever in the wait queue. The transition function δ_{V} specifies exactly two cases, where a process is dequeued from the wait queue: (a) there is a partner issuing a kernel call for IPC or (b) the operation times out. In both cases, VAMOS responds to the process with a result value that indicates success or failure. Formally: The transition function δ_{V} involves an update of the process $s.\mathsf{procs}\ \mathit{pid}$ by δ_{asm} *res* $\lceil s.\mathsf{procs}\ \mathit{pid}\rceil$ with the result value *res*—our definition of progress.

The remaining case is that process *pid* is ready at step k. We know that ready processes reside in the ready queue corresponding to their priority. When examining δ_{V}, we can observe that a process will be dequeued from a ready queue only if (a) it becomes inactive, (b) its priority changes, or (c) it has been computing. The first two cases lead to a contradiction with our assumptions (a) and (b)).

Computing usually means that the process makes progress immediately. Most notably, our implementation guarantees liveness, i. e., a started user process performs at least one step between two subsequent kernel entries (first phase in δ_{V}). Still, there might be no immediate progress if a computing process calls the kernel for an IPC operation. In this case, however, the kernel will enqueue the process in the wait queue, and we have shown fairness for all processes in this queue.

Finally, we regard the case that a process forever remains in the ready queue. Together with Assumption (d), we can infer that the process will move forward in the ready queue until it is the first one: The assumption ensures that the process has the current maximum priority infinitely often while the timer interrupt is active. When the timer interrupt occurs, the scheduler function `handle_timer` is invoked, and it charges the current process *cup*, i. e., the first process in *pid*'s ready queue (see Fig. 11). Charging a process means increasing the consumed time $\lceil s.\mathsf{schedds.procdb}\ \mathit{cup}\rceil.\mathsf{ctsl}$ unless it exceeds the timeslice, and moving the process to the end of its ready queue if the timeslice is exceeded. Thus, the consumed time value of *cup* increases strictly monotonic as long as the current process remains the first

in its ready queue. Moreover, timeslices are bound by a fixed value because they are stored in a 32-bit number in the implementation. Hence, the consumed time eventually exceeds the timeslice, and *cup* is moved to the end of its ready queue. That means, *cup* appears *after pid* in the ready queue. By induction, we can infer that process *pid* will eventually be the first in its ready queue and thus, when it eventually has the maximum priority, it is the current process. The current process computes in the next step and we have already shown that a computing process eventually makes progress. □

Temporal properties like our prioritized fairness are often described in temporal logics. The advantage of such a logic is succinctness while the Isabelle/HOL formalization (Corollary 1) is crucial for the property transfer. We have combined the best of both worlds by an embedding of future-time linear temporal logic (LTL) into Isabelle/HOL. Thus, we can finally present our main result in LTL:

Theorem 5 (Prioritized Fairness) *The* VAMOS *scheduler is fair with respect to priority classes, i. e., if a process* pid *(a) finally never terminates, (b) finally remains forever at the same priority class, (c) does infinitely often not pend in an IPC operation with an infinite timeout, and (d) has infinitely often the maximal priority while the timer interrupt is raised, it will always eventually progress.*

Formally:

$$\begin{aligned}\langle \mathscr{S}^0_{\mathsf{V}},\ \Sigma_{\mathsf{V}},\ \delta_{\mathsf{V}}\rangle_{\mathsf{A}} \models\ & (*\ \mathsf{a}\ *)\ \Diamond\Box\ (\lambda(i,\ s,\ n).\ s.\mathsf{procs}\ pid \neq \bot) \longrightarrow \\ & (*\ \mathsf{b}\ *)\ \Diamond\Box\ (\lambda(i,\ s,\ n).\ n.\mathsf{priodb}\ pid = s.\mathsf{priodb}\ pid) \longrightarrow \\ & (*\ \mathsf{c}\ *)\ \Box\Diamond\ (\lambda(i,\ s,\ n).\ \neg\ \mathsf{pending\_infinite\_ipc}\ s\ pid) \longrightarrow \\ & (*\ \mathsf{d}\ *)\ \Box\Diamond\ (\lambda(i,\ s,\ n).\ \mathsf{has\_maxprio}\ s\ pid \wedge \mathsf{is\_timer\_on}\ i) \longrightarrow \\ & \Box\Diamond\ (\lambda(i,\ s,\ n).\ \mathsf{progress}\ (s.\mathsf{procs}\ pid)\ (n.\mathsf{procs}\ pid))\end{aligned}$$

The term $\langle \mathscr{S}^0_{\mathsf{V}},\ \Sigma_{\mathsf{V}},\ \delta_{\mathsf{V}}\rangle_{\mathsf{A}}$ describes an *action Kripke structure*. This structure defines the set of all infinite traces that can be produced by the set of initial states $\mathscr{S}^0_{\mathsf{V}}$, the input alphabet Σ_{V}, and the transition function δ_{V}. Each trace in this set is a function from natural numbers to triples $(s,\ i,\ n)$, where s is the current state, i is the input for the transition, and n is the next state.

Proof We deduce the theorem from Corollary 1 mainly by the expansion of the LTL definitions. An LTL formula φ is valid for a Kripke structure K, i. e., $K \models \varphi$, iff φ is valid for all its traces:

$$K \models \varphi \equiv \forall t{\in}K.\ (t,0) \models \varphi$$

whereas the validity $(t,j) \models \varphi$ of a formula φ for trace t at position j is defined by cases, i. e.,

$$\begin{aligned}&(t,j) \models P \equiv P\ (t\ j)\\ &(t,j) \models \Box\varphi \equiv \forall k{\geq}j.\ (t,k) \models \varphi\\ &(t,j) \models \Diamond\varphi \equiv \exists k{\geq}j.\ (t,k) \models \varphi\end{aligned}$$

Ultimately, Corollary 1 all-quantifies over a single suffix while in LTL, Assumption (a), Assumption (b), and the conclusion might hold for different suffixes. We instantiate the position k in the corollary with the maximal position of the tree suffixes and derive our claim. □

According to Francez [23], the classic definition of strong fairness is formalized in LTL as: $\Box\Diamond enabled \longrightarrow \Box\Diamond selected$, i. e., if a certain process is infinitely often ready to compute (LTL-predicate *enabled*), it will always eventually compute (*selected*). Technically, our theorem has a different structure.

If we examine the Assumptions (a) and (b), however, we observe that these assumptions are of a very basic kind: It is certainly impossible to prove temporal fairness for a terminating process, and a changed priority contradicts the notion of *static*-priority scheduling. We maintain that any real system has similar basic assumptions and do not consider these conditions as part of the *enabledness*.

We may confine our consideration to a system with a static number of processes and without priority changes, i. e., after a time of system set up, the kernel calls PROCESS_KILL and CHG_SCHED_PARAMS are no longer used. In this scenario, the Assumptions (a) and (b) become *set up conditions* and we just consider the state after set up as the initial state. In this system, we have only two enabledness assumptions of the kind $\Box\Diamond \mathit{enabled}$. This artifact is important because the conjunction of both enabledness assumptions would be considerably weaker.

8 Conclusion

We provide a formal proof of a microkernel's key property, namely the temporal fairness property of its multi-priority process scheduler. The proof architecture links a layer of behavioral reasoning over system-trace sets over a concrete, fairly realistic implementation of the VAMOS kernel written in C down to a formal, foundational model of a RISC processor called VAMP. To our knowledge, such a proof based on a model stack of this concrete level of detail and with such a clean, seamless logical foundation has been undertaken for the first time.

We believe that our work represents a significant step towards the grand challenge of "real code verification", although we compromised in a number of ways in order to achieve our goal:

- the VAMP is not a "real", i. e., industrial-strength processor,
- C0 is a typed, simplified fragment of C; this forces to shift more low-level computations into assembly programs as necessary in a more powerful C execution model,
- the compiler of C0 is not optimizing, and
- the VAMOS code has been written by the verification engineers themselves, and often with foresight to the tool-chain and the verification task.

Despite these simplifications, which were partly applied for project-pragmatic reasons, we maintain that our models are still not too far away from industrial-strength processors, C code and microkernel implementations such as the PikeOS kernel. Whether it is ever possible to verify system-level programming code of a substantial size written *without* any foresight to verification is a fully open question at present.

We would like to argue that the possibility of adapting specifications, code, and tool-chain to each other greatly simplified the task of achieving our goal. Besides the obvious foresight that all code was written in C0 and had to live with the restrictions imposed by our tool-chain, we see the following (incomplete) list of mutual influences:

- The abstract model uses infinite datatypes to model key entities like time, processes, etc., while the concrete implementation of these entities is bound to bit-vector representations of numbers. We ensured the refinement relation by using a solely relative notion of time within the kernel. Furthermore, a capability-like management of process identifiers allows for a conceptually infinite name space of identifiers.
- The function `kernel_step` assures that at least one assembly step of a user process is executed (actually, an earlier version of the CVM layer contained a flaw in this respect). Without this property of `kernel_step`, our global theorem breaks.

- The fairly simple refinement scheme between `kernel_step` and δ_{V+D} required a lot of experimenting within the definition of the abstraction relation, which has to cope with the fact that parts of the concrete computations "overtake" their abstract counterparts and vice versa.
- The precise form of the contracts and the invariants in the implementation certainly needed the foresight on what was actually required in the proofs at higher levels.

It turned out that a major obstacle of our work was the lack of early, systematic validation of the specification; at the end, we found substantially more errors there than in the (fairly well-tested) implementation, although some intricate bugs could only be revealed by the verification. One of the bugs revealed during the verification was a race condition involving the coincidence of four special cases at the same time: If (a) a process issued an IPC call, (b) the IPC partner was not yet waiting, and in the same processor cycle, (c) the timer interrupt was raised, and (d) the timeslice of the process was used up, then, the process was erroneously re-added to the ready queue.

In conclusion, we point out that there is a fundamental difference between our pervasive approach and the idea of an incremental verification focusing at one layer at a time. In the course of our work, we detected several flaws in the kernel design, in its specification, and in its implementation. Most notably, we found the above mentioned flaw in the CVM layer during the verification of the fairness property, which demonstrates that bugs are not necessarily revealed during the verification of a single layer. It has been crucial for our success that we thought pervasive right from the beginning.

Furthermore, we experienced that pervasiveness entails more than just cumulative verification efforts on several (isolated) layers. In fact, it was a challenging task to integrate models and proofs into a uniform, coherent theory. The effort for the actual fairness proof on the abstract level is a small fraction (about ten person months) compared to the whole verification effort establishing its foundation, the abstract kernel model (more than two person years for VAMOS excluding CVM). Besides that, there was a considerable effort for the management of change throughout the different abstraction layers. Apart from the necessary effort, however, we are confident that our methodology can be adapted and reapplied in similar contexts.

The formal verification work of this paper is complete with the exception of the contracts for the CVM operations, which have been shown by Tsyban *et al.* [27,48,49] but on a substantially lower abstraction level than the form used in this paper. The missing link is a transfer lemma similar to the one shown by Alkassar *et al.* [4]. Moreover, for some of the kernel subroutines, the implementation correctness has not yet been shown.

8.1 Related Work

The Kit Project [8,9,36] can only be referred as groundbreaking to the area of pervasive operating-system verification. The main difference to more recent verification attempts of "real software" is that it relies on a LISP execution model that is nowadays considered fairly abstract. Moreover, the corner stone theorem of this work is on memory separation of processes.

Since then, a number of smaller and larger operating system verification projects have been started. In the former category fall projects like the FLINT project [38], the MASK project, the AAMP7 project, the EMBEDDED DEVICE project and EROS/Coyotos [46], for all of which we refer the interested reader to the excellent and comprehensive overview by Klein [29].

Larger verification projects currently aim at the complete code-level verification of the following operating system kernels:

PikeOS kernel [28]. This 6,000 loc L4-derivative is part of a commercial product available for Intel's x86, PowerPC, and ARM. At the time of writing, the verification just started as part of the Verisoft XT project. The proofs are carried out by the VCC verification environment (a descendant of the Spec# program-verification environment of Microsoft Research), which uses a trusted tool chain comprising the C translator *VCC*, the verification condition generator *Boogie*, and the automated theorem prover *Z3*. The supported C fragment is a large fragment of ANSI C. It is too early to make estimates on the portion of verified code.

seL4 [25]. The 10,000 loc kernel is at the brink of becoming a commercial product and based on the ARM11 platform. The verification project L4.verified links a "Formal Low-Level Design" (i. e., a model mechanically derived from a Haskell Prototype) to the efficient C implementation. At the time of writing, this effort is claimed to be to 70% complete. The approach uses a trusted compiler to the verified Isabelle/Simpl framework. The supported C fragment is a large fragment of ANSI C.

HYPER-V [43]. The 50,000 loc kernel is part of a commercial virtualization environment based on Intel's x86 (plus additional hardware for MMU virtualization). The verification is realized by the VCC verification environment (see above); proofs are performed by Z3 and Isabelle/HOL-Boogie [11] using the axiomatization of the VCC memory model as foundation. The supported C fragment is a large fragment of ANSI C. While the 30 person year project is well under way, is too early to make estimates on the portion of verified code.

Summing up, we have to state a trade-off between the code size, the supported C fragment, the complexity of the underlying machine model, and the trustworthiness of the logical foundations of the used tool chain. With respect to the latter, the L4.verified approach comes closest to our work. Still, the approach requires trust on the abstraction of the architecture (whose informal description comprises about 800 pages of natural text) into a C execution model and its compilation into Isabelle/Simpl: this is not exactly easy to swallow for the skeptic and the paranoid. The VCC technology [15] demands an even higher level of trust: The current VCC version uses an architecture-dependent axiomatization of the execution model consisting of about 200 axioms, which introduce some rather abstract concepts like concurrency and ownership of references. Though critical subproblems of the foundation are tackled by informal as well as formal proof methods, the integration into a uniform foundational theory is significantly less prioritized.

In this respect, the Verisoft XT approach is fundamentally post-hoc; tool chains, methodologies etc. are driven by the need to deal with the existing code that can only be changed if errors have been revealed. In contrast, we based our work on a model stack covering the processor architecture formally, consequently we wrote our code in a restricted C subset and developed our kernel implementation with foresight to our tool chain. While the post-hoc approach has the advantage to complement the conventional development workflow and to focus the verification efforts on well-established code, we maintain that our methodology in adapting all three — specifications, tools, implementations — to each other can be pursued with a substantially higher effectivity and with a cleaner semantic foundation.

With respect to the proof architecture — separating a refinement proof relating abstract and concrete system steps from a proof establishing behavioral properties on the abstract level — our work is rather typical: Cock *et al.* [14] have applied this architecture to establish security properties (e. g., termination of the kernel calls and well-typedness of kernel

objects) on the behavioral level over an abstraction of the L4 Microkernel. The range of abstraction levels and the involved models resembles our work; the target property, however, is from a completely different domain. Another application of the architecture is the DARMA case study [7], where a client-server system for the digital signature of critical documents is proven to establish a number of high-level security properties (e. g., no client will ever get a digital signature without having a valid session with the server). Again this work is different in that it targets a temporal top-level property stemming from the security domain; in contrast to our work, however, the abstraction level of the "concrete level" in DARMA is fairly coarse.

8.2 Future Work

We foresee various possible extensions of our work with widely varying realization efforts. An apparent and less expensive extension is a detailed analysis of the effective runtime costs for the bookkeeping of process switches between timer interrupts. An accompanying study on the possible consequences of priority inversion for the programming model of prioritized processes would permit a well-balanced choice for either a simpler programming model at additional runtime IPC costs or the current, faster solution.

Certainly, additional microkernel features could extend our work. As an example, we could introduce multi-threading generalizing the multi-processing of VAMOS. This feature amounts to sharable address spaces (possibly including user-level paging) and is primarily an extension of the CVM framework. We assume only a small impact on the code verification because VAMOS alters processes only via CVM primitives. Certainly, the progress predicate needs to be adjusted but we do not foresee any complications because the progress of a process necessarily implies altered registers, not only memory changes.

A similar extension is the mapping of device addresses directly into user memory. This change would abandon the kernel calls DEV_READ and DEV_WRITE for device communication, and thus substantially improve the system performance by reducing both, the size of the kernel and the frequency of kernel calls. Despite these benefits, we complicate the abstract kernel model with respect to device communication because we need a more sophisticated detection of device accesses. For that reason, we did not implement this optimization right from the beginning. From our experience today, however, we do not foresee substantial obstacles in this change.

Following the last two arguments, we could even strive for the direct memory access (DMA) by devices. Admittedly, this feature requires a more elaborate reorganization of the abstract kernel models and prevents the clean isolation of processor and devices as we currently maintain it for the fairness proof.

Another direction of further research is a port of the CVM framework to a mass-market processor such as an embedded PowerPC core or to an optimizing compiler supporting a larger subset of C. In contrast to additional microkernel features, this change would not necessarily require changes to the implementation of VAMOS (apart from CVM). Provided that the port maintains the same CVM specification, we could hence draw on the current proofs of code correctness and fairness.

Acknowledgements We thank Andrew Baumann, Sebastian Bogan, Christian Hennrich, Sarah Hoffmann, and the anonymous reviewers during the publication process for reviewing, constructive criticism and helpful suggestions, as well as Norbert Schirmer for taming Isabelle's document-generation system.

References

1. Alkassar, E., Hillebrand, M.A.: Formal functional verification of device drivers. In: J. Woodcock, N. Shankar (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 225–239. Springer (2008)
2. Alkassar, E., Hillebrand, M.A., Leinenbach, D., Schirmer, N.W., Starostin, A.: The Verisoft approach to systems verification. In: N. Shankar, J. Woodcock (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 209–224. Springer (2008)
3. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load – leveraging a semantics stack for systems verification. J. Autom. Reasoning, Special Issue on Operating System Verification (2009). To appear in this volume.
4. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: TACAS, *LNCS*, vol. 4963, pp. 109–123. Springer (2008)
5. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Kluwer Academic Publishers (2002)
6. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: F.S. de Boer, M.M. Bonsangue, S. Graf, W.P. de Roever (eds.) Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, *LNCS*, vol. 4111, pp. 364–387. Springer (2006)
7. Basin, D.A., Kuruma, H., Miyazaki, K., Takaragi, K., Wolff, B.: Verifying a signature architecture: a comparative case study. Formal Asp. Comput. **19**(1), 63–91 (2007)
8. Bevier, W.R.: Kit and the short stack. J. Autom. Reasoning **5**(4), 519–530 (1989)
9. Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. J. Autom. Reasoning **5**(4), 411–428 (1989)
10. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. STTT **8**(4-5), 411–430 (2006)
11. Böhme, S., Leino, K.R.M., Wolff, B.: HOL-Boogie - an interactive prover for the Boogie program-verifier. In: O.A. Mohamed, C. Muñoz, S. Tahar (eds.) TPHOLs, *LNCS*, vol. 5170, pp. 150–166. Springer (2008)
12. Brock, B., Kaufmann, M., Moore, J.S.: ACL2 theorems about commercial microprocessors. In: FMCAD, pp. 275–293 (1996)
13. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940)
14. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: O.A. Mohamed, C. Muñoz, S. Tahar (eds.) TPHOLs, *LNCS*, vol. 5170, pp. 167–182. Springer (2008)
15. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C (2008). Availabe via http://research.microsoft.com/apps/pubs/default.aspx?id=77174
16. Daum, M., Dörrenbächer, J., Wolff, B., Schmidt, M.: A verification approach for system-level concurrent programs. In: J. Woodcock, N. Shankar (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 161–176. Springer (2008)
17. Elphinstone, K., Greenaway, D., Ruocco, S.: Lazy scheduling and direct process switch – merit or myths? In: Workshop on Operating System Platforms for Embedded Real-Time Applications. Pisa, Italy (2007). Availabe at http://www.ertos.nicta.com.au/publications/papers/Elphinstone_GR_07.pdf
18. Engler, D.R., Kaashoek, M.F., O'Toole, J.: Exokernel: An operating system architecture for application-level resource management. In: SOSP, pp. 251–266. ACM (1995)
19. Fagin, R., Williams, J.H.: A fair carpool scheduling algorithm. IBM Journal of Research and Development **27**(2), 133–139 (1983)
20. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: CAV, pp. 173–177 (2007)
21. Fleisch, B.D., Co, M.A.A.: Workplace microkernel and OS: a case study. Softw. Pract. Exper. **28**(6), 569–591 (1998)
22. Fox, A.C.J.: Formal specification and verification of ARM6. In: TPHOLs, pp. 25–40 (2003)
23. Francez, N.: Fairness. Springer (1986)
24. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. White paper, AbsInt Angewandte Informatik GmbH (2004). Availabe via http://www.absint.com/wcet.htm
25. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: taking microkernels to the next level. Operating Systems Review **41**(4), 3–11 (2007)
26. Hillebrand, M.A., Paul, W.J.: On the architecture of system verification environments. In: Haifa Verification Conference, pp. 153–168. Springer (2007)

27. In der Rieden, T., Tsyban, A.: CVM – a verified framework for microkernel programmers. In: Systems Software Verification, *ENTCS*, vol. 217, pp. 151–168. Elsevier Science B.V. (2008)
28. Kaiser, R.: Combining partitioning and virtualization for safety-critical systems. White Paper WP_CPV_10_A4_R10, SYSGO AG (2007). Availabe via http://www.sysgo.com/news-events/whitepapers/
29. Klein, G.: Operating system verification — an overview. Tech. Rep. NRL-955, NICTA, Sydney, Australia (2008)
30. Knapp, S., Paul, W.: Realistic worst case execution time analysis in the context of pervasive system verification. In: T. Reps, M. Sagiv, J. Bauer (eds.) Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday, *Lecture Notes in Computer Science*, vol. 4444, pp. 53–81. Springer (2007). URL http://www.verisoft.de/.rsrc/PublikationSeite/KP06.pdf
31. Leinenbach, D.: Compiler verification in the context of pervasive system verification. Ph.D. thesis, Saarland University, Saarbrcken (2008). URL http://www-wjp.cs.uni-sb.de/publikationen/Lei08.pdf
32. Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM, pp. 2–12. IEEE Computer Society (2005)
33. Liedtke, J.: Improving IPC by kernel design. In: SOSP, pp. 175–188. ACM (1993)
34. Liedtke, J.: On μ-kernel construction. In: SOSP, pp. 237–250. ACM (1995)
35. Liedtke, J.: Towards real microkernels. Commun. ACM **39**(9), 70–77 (1996)
36. Moore, J.S.: A grand challenge proposal for formal methods: A verified stack. In: 10th Anniversary Colloquium of UNU/IIST, pp. 161–172. Springer (2002)
37. Moore, J.S., Lynch, T.W., Kaufmann, M.: A mechanically checked proof of the $\mathrm{AMD5_K86^{TM}}$ floating point division program. IEEE Trans. Computers **47**(9), 913–926 (1998)
38. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: TPHOLs, *LNCS*, vol. 4732, pp. 189–206. Springer (2007)
39. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
40. Petters, S.M., Zadarnowski, P., Heiser, G.: Measurements or static analysis or both? In: C. Rochange (ed.) WCET, *Dagstuhl Seminar Proceedings*, vol. 07002. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
41. Rashid, R., Avadis Tevanin, J., Young, M., Golub, D., Baron, R.: Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE Trans. Comput. **37**(8), 896–908 (1988)
42. Ruocco, S.: Real-time programming and L4 microkernels. In: Workshop on Operating System Platforms for Embedded Real-Time Applications. Dresden, Germany (2006). Availabe at http://www.ertos.nicta.com.au/publications/papers/Ruocco_06.pdf
43. Samman, T.: Verifying 50,000 lines of C code. Futures, Microsoft's European Innovation Magazine (21) (2008)
44. Schirmer, N.: A verification environment for sequential imperative programs in Isabelle/HOL. In: LPAR, LNCS, pp. 398–414. Springer (2005). URL http://isabelle.in.tum.de/~schirmer/pub/hoare-lpar04.html
45. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, TU Munich (2006)
46. Shapiro, J., Doerrie, M.S., Northup, E., Sridhar, S., Miller, M.: Towards a verified, general-purpose operating system kernel. In: FM Workshop on OS Verification, Tech. Rep. 0401005T-1, pp. 1–19. National ICT Australia (2004). URL http://www.coyotos.org/docs/osverify-2004/osverify-2004.pdf
47. Singal, M., Petters, S.M.: Issues in analysing L4 for its WCET. In: Workshop on Microkernels for Embedded Systems. Sydney, Australia (2007). Availabe at http://www.ertos.nicta.com.au/publications/papers/Singal_Petters_07.pdf
48. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Systems Software Verification, *ENTCS*, vol. 217, pp. 169–185. Elsevier Science B.V. (2008)
49. Starostin, A., Tsyban, A.: Verified process-context switch for C-programmed kernels. In: N. Shankar, J. Woodcock (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 240–254. Springer (2008)