Temporal Fairness of a Microkernel Scheduler

Matthias Daum

Computer Science Dept., Saarland University
Saarbrücken, Germany
md11@wjpserver.cs.uni-saarland.de

1 Introduction

We report on the formal verification of a microkernel's temporal property, namely that its multi-priority process scheduler guarantees progress, i. e., strong fairness. This paper summarizes parts of a journal article [2].

Our microkernel VAMOS provides an infrastructure for a dynamically changing number of processes, for memory virtualization, for inter-process communication (IPC), and for user-level device-driver support. The kernel establishes process switches according to IPCs and timer events. The process scheduling, however, follows a hierarchy of priorities, favoring, e. g., system processes over application processes over maintenance processes. Processes may control the kernel tasks and access kernel services via so-called *kernel calls*. A minimal access control is built into the kernel to restrict the kernel calls according to a user-defined policy.

For space restrictions, we cannot present the microkernel or our formal foundation at large. We refer the interested reader to a previous publication [1]. In this paper, we confine ourselves to a short summary on the scheduler functionality as well as the formal kernel model that serves as basis for our verification efforts. In Section 4, we reveal insights into our fairness theorem. Finally, we conclude in Section 5.

2 Scheduler

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. This policy is also known as *static-priority scheduling* though technically, priorities might be adjusted through kernel calls at any time during run-time.

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 run-times while priorities lead to the preemption of lower process classes.

Inside the kernel, time measurements are based on a timer device, which periodically raises its interrupt line. In particular, we measure the run-time of processes and the waiting time for IPC in clock ticks. Note that the kernel implementation maintains a solely relative notion of time.

3 Underlying Kernel Model

We base our temporal proof on an operational kernel model, or more precisely, the formal specification of the kernel, which is used for the verification of functional implementation correctness. This model is a state automaton. The state space contains the state of the currently running processes as well as information about the scheduling internals, access control rights, and pending device communication. A transition of the VAMOS specification depends on a device input and has up to three phases:

(a) At first, the current process is advanced. This operation is either process local or it handles a kernel call. (b) If the device input indicates that the timer-interrupt line is raised, the scheduler increases the consumed time of the current process and a global counter, which measures the waiting time for IPC. Afterwards, the kernel wakes up all processes with elapsed IPC timeouts. (c) Finally, VAMOS delivers interrupts to waiting driver processes and saves the remaining interrupts for later delivery.

4 Scheduler Fairness

According to the taxonomy of Francez [3], we formalize the temporal property of strong fairness for our scheduler. In order to state this property in reasonable succinctness, we establish several abstraction layers on top of the operational kernel model. At first, the operational model only formalizes states and transitions. For temporal reasoning, however, we need execution traces. We represent a trace by two functions *states* and *inputs* that map the step number to the current state and to the device input for the next step, respectively.

Definition 1 (Trace). Two functions *states* and *inputs* describe a trace iff (a) The first state is in the set of initial states and (b) for each step number, the transition relation holds for the current state and input and the successor state.

Furthermore, we need an invariant inv_{V} over the state sequences for reasoning about the behavior of a trace. This invariant is quite elaborate such that space restrictions prohibit its presentation. We show that this predicate indeed is an invariant over the traces:

Lemma 1 (Invariant). *Predicate* inv_{V} *is an invariant over traces.*

Proof. We prove this statement by induction. At first, we establish inv_{V} for all initial states. At second, we show that inv_{V} is maintained by the transition function. Finally, we derive our claim by the definition of traces. Note that despite this short sketch, the formal proof is quite elaborate because of the complexity of the kernel specification and the invariant. Its verification took more than two person months. □

With these prerequisites in place, we can formulate our scheduler's fairness property. Recall that the process scheduling follows a hierarchy of priorities. We describe this phenomenon by the notion of *prioritized fairness.* In our system, the scheduler can only guarantee fairness under several assumptions:

1. Any temporal fairness statement certainly excludes terminating processes. Note that processes might still terminate in our system, we can just not say that the scheduler would treat terminating processes fairly.
2. Changing the priority class of a process contradicts the notion of *static*-priority scheduling. The scheduler does not treat a process fair if its priority is changed infinitely often.
3. If a process chooses to wait infinitely long for an IPC operation, it might starve. This problem is certainly not the fault of the scheduler but a programming or protocol error.

4. Our scheduler relies on a live timer device. At this time, the next process is selected from the highest, non-empty priority class. In other words, a process is only scheduled at all if it has the maximal priority when the timer is on.

With the current state of affairs, however, a formal statement requires explicit quantification on step numbers over the *states* and *inputs* functions. We have proven the theorem on this layer for methodical reasons: The larger context of this work is pervasive systems verification and the long-term goal is a coherent theory in a single theorem prover. Consequently, a specialized theorem prover for temporal logic is insufficient in this context. Hence, we combined the best of both worlds and embedded future-time linear temporal logic (LTL) with *action Kripke structures* into Isabelle/HOL [4]. Thus, we can integrate our main theorem into the overall context while presenting it in LTL:

Theorem 1 (Prioritized Fairness). *The* VAMOS *scheduler is fair with respect to priority classes, i. e., if a process pid (1) finally never terminates, (2) finally remains forever at the same priority class, (3) does infinitely often not pend in an IPC operation with an infinite timeout, and (4) has infinitely often the currently maximal priority when the timer interrupt is raised, it will always eventually progress. Formally:*

$$\begin{aligned}\langle \mathcal{S}^0_{\mathsf{V}}, \Sigma_{\mathsf{V}}, \delta_{\mathsf{V}}\rangle_{\mathsf{A}} \vDash\ & \Diamond\Box(\lambda(i,s,n) : \mathit{process\_exists}(s,\mathit{pid})) \implies \\ & \Diamond\Box(\lambda(i,s,n) : n.\mathit{priority}(\mathit{pid}) = s.\mathit{priority}(\mathit{pid})) \implies \\ & \Box\Diamond(\lambda(i,s,n) : \neg \mathit{pending\_infinite\_ipc}(s,\mathit{pid})) \implies \\ & \Box\Diamond(\lambda(i,s,n) : \mathit{has\_maxprio}(s,\mathit{pid}) \wedge \mathit{is\_timer\_on}(i) \implies \\ & \Box\Diamond(\lambda(i,s,n) : \mathit{progress}(s.\mathit{procs}(\mathit{pid}), n.\mathit{procs}(\mathit{pid})))\end{aligned}$$

For lack of space, we can neither present the detailed formalization of the above theorem nor a proof sketch. The formal proof based on the quantification over the trace functions took about eight person months and is available at the public Verisoft repository.[1] Though the LTL layer permits a considerably more succinct statement, its formalization and the abstraction of the original statement could be accomplished in just about a person week.

5 Conclusion

During verification, we found a bug in the temporal behavior of our implementation, which cannot be found in a step-wise refinement proof. This finding demonstrates the immediate practical usefulness of our proof. Furthermore, our proof can be integrated with a refinement proof that links our underlying model to the C implementation. To our knowledge, this is the first verification result that establishes a temporal property on a C implementation.

References

1. Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In *VERIFY Workshop*, pages 56–70. CEUR-WS.org, 2008.
2. Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *Journal of Automated Reasoning, Special Issue on Operating System Verification*, 42(2-4):349–388, 2009. DOI: 10.1007/s10817-009-9119-8.
3. Nissim Francez. *Fairness.* Springer, September 1986.
4. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002.

[1] The public Verisoft Repository is available at `http://www.verisoft.de/VerisoftRepository.html`.