Ingredients of Operating System Correctness*

Lessons Learned in the Formal Verification of PikeOS

Christoph Baumann[1], Bernhard Beckert[2], Holger Blasum[3], and Thorsten Bormer[2]

[1] Saarland University, Dept. of Computer Science, Saarbrücken, Germany
[2] Karlsruhe Institute of Technology, Germany
[3] SYSGO AG, Klein-Winternheim, Germany

Abstract. In the context of the Verisoft XT project functional correctness of the microkernel of PikeOS from SYSGO AG is shown at the source code level using the VCC verification tool, developed by Microsoft Research.
In this paper we outline a simulation theorem between a top-level abstract model and the system consisting of the kernel and user programs running in alternation on the real machine.
Based on an example of a typical code trace through the kernel, we identify the correctness properties of all components in the trace that are needed for the overall correctness proof of the microkernel.

1 Introduction

A common technique for validating the functionality of systems is testing. Ideally via the testing effort one generates some amount of assurance that the system works correctly. A more rigorous approach is to apply formal methods—in our case formal software verification to prove that a program meets its specification.

While code verification has a long history it was not until recently that formal methods were successfully applied to verify complex operating system kernels [8]. In the first phase of the Verisoft project it has been shown that pervasive formal verification of an academic operating system including its execution environment, like the underlying hardware and the compiler, is feasible.

In the subproject Avionics of the successor project Verisoft XT, this knowledge is applied and refined to the verification of a real world implementation of a microkernel used in industrial embedded systems (e.g. in avionics, automotive and control), namely PikeOS from SYSGO AG which operates in a safety-critical environment. One goal of the Verisoft XT sub-project Avionics is to prove functional properties of the source code of the microkernel using Microsoft's verification tool VCC [9]. The result of this is a machine checked mathematical proof that the implementation fulfills its specification.

Unlike the first phase of the Verisoft project we do not attempt the pervasive verification of our verification target, but rather focus on the large scale verification of real life software. Nonetheless, it is important to make explicit which parts of the correctness theorems are actually proven and to state the assumptions that we need to make about the components of the system (including the verification tools)—these are often left implicit when coming up with a proof of some correctness property of a complex system. We argue that pinpointing these correctness ingredients in a systematic manner is an important part of any verification methodology that is applied to large scale software systems.

The reason for this is that a formal proof just guarantees correctness of an implementation concerning a particular specification. Hence it is crucial to thoroughly analyse which properties of the system are captured by the specification and to identify missing parts, that might impair reliability.

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

This paper presents core assumptions and arguments that constitute the correctness of an operating system's microkernel. It also demonstrates how to combine these ingredients into a formal proof of a simulation theorem with VCC.

A presentation on functional correctness of low-level portions of the code was given at the Embedded World 2009 conference ("Better Avionics Software Reliability by Code Verification" [1]), and we were asked how that fits into a broader context. This work can also be seen as an answer to that question.

1.1 PikeOS as Verification Target

The PikeOS system [11] can be used as paravirtualizing hypervisor. For example, in a safe and secure manner, without unspecified interference, one can run a Linux system in one partition and (in avionics contexts) an ARINC-653 application or (in traffic control contexts) a POSIX or Java runtime-environment in another. The PikeOS system consists of a microkernel and a PikeOS system software component. In the following, unless clearly indicated otherwise, we will only speak about the PikeOS microkernel (for brevity also referred to as "kernel" instead of "microkernel"). The PikeOS kernel is particularly tailored to the context of embedded systems, featuring real-time functionality and orthogonal partitioning of resources such as processor time, user address space memory, and kernel resources. At the kernel level, the mechanisms for communication between threads are IPC, events, and shared memory. Moreover, the kernel is interruptible—a challenge in the formal verification of system calls that is often avoided for academic systems. Most parts of the PikeOS kernel, especially those that are generic, are written in C, while other parts that are close to the hardware are necessarily implemented in assembly. PikeOS runs on many platforms, including x86, PowerPC, MIPS, and ARM, among others—the verification target we have chosen for our project, is a PikeOS kernel setup for the PowerPC processor family (PowerPC OEA MPC5200 [5], a single processor setup). While the exact amount of assembly depends on the architecture one works on, in our particular case, PowerPC assembly is about one tenth of the codebase.

1.2 The Verifying C Compiler

In our project we utilize the Verifying C Compiler (VCC) developed by Microsoft Research [2]. VCC uses a specification language tailored to C, which allows a verification engineer to write the specification in a way close to the syntax and semantics of the programming language. In addition, this specification—we speak of *ghost code* or objects in *ghost memory*—is transparent to the normal C compilation process and does not interfere with the original C program.

From the source of the program and the specification, VCC automatically generates a logical formula. This formula, called *verification condition*, is rendered in predicate logic and has a property that, if it is satisfiable, then the program is correct w.r.t. its specification. The satisfiability of this formula is then verified by the SMT solver Z3 [3] (Satisfiability Modulo Theories).

The possible results Z3 may return are: (1) a proof for the validity of the formulas, (2) a counterexample, or (3) Z3 runs out of resources (time or space). In case (1) above, the program verification was successful. In cases (2) and (3), the verification engineer has to analyze the problem and correct the error. In case (3), she/he may also find that the program indeed satisfies the annotations. Then new annotations (stronger invariants, helpful lemmas, etc.) have to be added. This process is repeated until Z3 finds a proof.

The main effort for the verification engineer in this context is to formulate a specification of the intended functionality for the respective program and annotate it accordingly in VCC language, so that the tool is able to deduce the formal proof automatically.

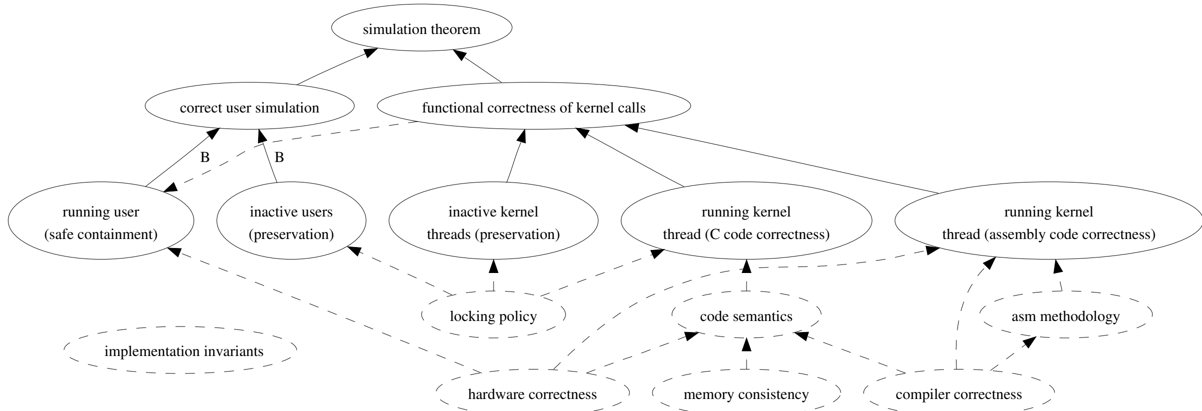

Fig. 1. Overview of the theorem and proof structure; dashed items represent core lemmata and proof arguments.

1.3 The Big Picture

The correctness of the kernel is stated as a simulation relation between an abstraction of the system and its implementation. Our approach is based on the verification of cvm, an academic microkernel framework, which was formally proven correct in the first Verisoft project [12]. However, we have to extend this methodology to meet the complexity of an industrial product (cf. Sect. 7) and our new toolchain (cf. Sect. 2.2 and 6).

The upper half of Figure 1 shows the structure of the simulation theorem. It involves argumentation about running user processes as well as the execution of C and assembly code in the kernel implementation. Moreover, separation properties have to be shown between currently running and inactive user processes and kernel threads (when execution is within the kernel address space, we speak of a kernel *thread*, when execution is within a user address space we speak of a user *process*). In addition to the simulation properties there are implementation invariants the code has to obey (see Fig. 2). They may concern generic data structures such as lists, queues, and trees and their general well-definedness requirements, but also instances of these that are used in the various modules of the kernel. The latter objects may incorporate specific requirements depending on their application. Memory separation is modelled as an implementation invariant on hard- and software entities that realise address translation for user and kernel threads.

In general the simulation relation has to hold for *all* execution steps. Nevertheless, at certain steps certain proof obligations are less obvious than others. Consequently we examine a typical execution trace of the system and focus on the cornerstone proof arguments that have to be produced respectively (see Sect. 3 to 8). Some examples of dependencies between proof obligations and arguments are displayed in Fig. 1 using dashed style. Observe that the user step depends inductively on the functional correctness of kernel methods. In addition, all proof steps assume the implementation invariants as an induction hypothesis (that dependency is not displayed so that the graph does not become too cluttered). The details of the theorem will be discussed in the following sections.

2 A Top Level View on the Simulation Theorem

A simulation theorem can be used to state the correctness of a system, i.e., that the specification is fulfilled by the implementation. The proof is then conducted by inductively showing that each step of the specification is realized by a certain number of steps in the implementation.

For the academic microkernel cvm a simulation theorem has already been developed and proven in the first Verisoft project [12]. While an industrially used operating system like PikeOS differs from cvm in many features (e.g., full C semantics, interruptible kernel, shared memory, real-world architecture), our principal approach to show its correctness remains the same: formally verifying a simulation theorem between an abstract specification and the concrete implementation of the

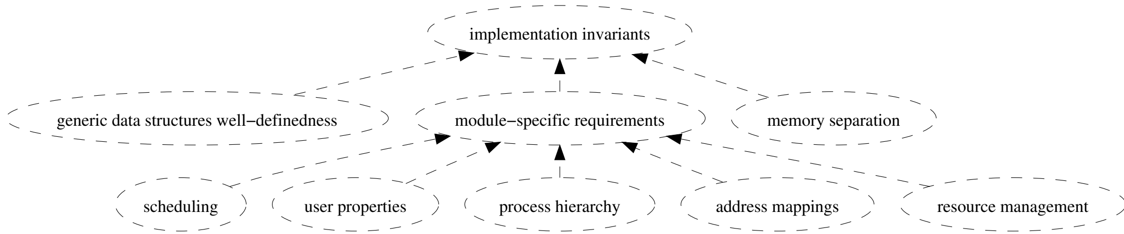

Fig. 2. Overview of typical implementation invariants, which must be obeyed by all execution steps.

system. In the following we give a coarse description of this theorem and how we model it with VCC.

2.1 Layers of Abstraction

For a simulation proof we need to look at the system at different layers of abstraction. In our case there are three of them. The first and most abstract one is cvm—the specification model. It consists on one hand of the *abstract kernel* which specifies the user-visible parts of the implementation and hides hardware functionality. The other part consists of the user processes running on the system. We interpret these processes as separate *virtual machines* that communicate with each other only via defined channels (e.g. shared memory, IPC). The *concrete kernel* layer represents the C and assembly implementation which precisely describes the functionality of most parts of the kernel, given one has assigned an unambiguous semantics to C by fixing a compiler and an architecture The *architecture* layer models the physical hardware on which assembly code, compiled C code, and user processes are executed. Formally we can model these layers as follows

- cvm—the abstract model consisting of:
  - $cvm.vm(i)$—the *virtual machine* of the i-th user process, consisting of the a CPU context $vm(i).cpu$ and a virtual memory portion $vm(i).m$ of some adjustable size.
  - $cvm.c(i)$—the C configuration of the *abstract kernel* thread i, comprising components like program code or a local memory stack and sharing global memory with the other threads. Note that $c(i)$ only becomes active when $vm(i)$ enters the kernel (e.g., via a system call).
- $k(i)$—the C configuration of the *concrete kernel* thread which implements $c(i)$, including additional non-user-visible data structures and assembly code.
- h—the model of the underlying hardware *architecture*, basically comprising the CPU context $h.cpu$ and physical memory $h.m$.

Now we define different relations connecting the different layers. For instance we define a *B-relation* [6] that relates specification and implementation of the user processes. It states that the context of the active user process agrees with the CPU registers and all other user processes are encoded in dedicated data structures of the kernel. For the virtual machines' memory it demands that memory contents are equal to those of the corresponding regions on the physical machine. There is also an *abstraction relation* between abstract and concrete kernel as well as a *compiler consistency relation* between C code and compiler-generated assembly code, that guarantee that the concrete kernel program is correctly executed on the underlying hardware.

Formally, we combine these relations into an overall relation $cvm\text{-}sim(cvm, k, h)$ stating that the cvm model is simulated by the concrete kernel k and the hardware state h (we also use the term "simulation relation"). In addition there are implementation invariants $impl\text{-}inv(cvm, k, h)$ for specific layers, which specify that the contained components and data structures remain well-defined (see Fig. 2).

For any n execution steps in the cvm model a trace of m hardware steps can be found that simulates the cvm execution, such that all three layers are consistent to each other.

With transitions on the cvm model and the hardware defined by step functions δ_{cvm} and δ_h, the overall simulation theorem between cvm, concrete kernel and architecture layer can be stated

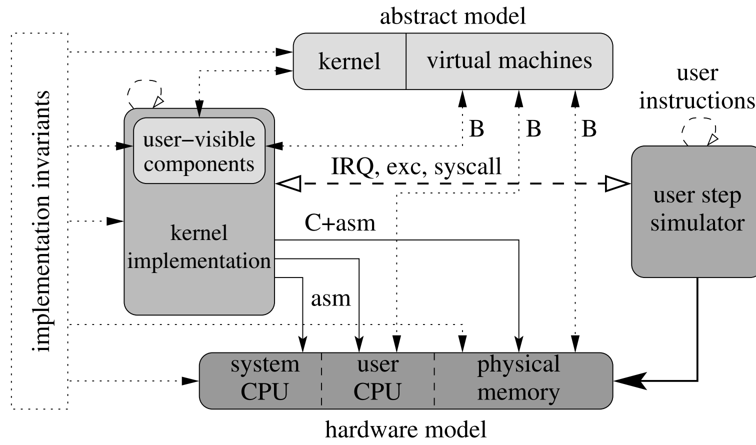

Fig. 3. A schematic view of the overall system model. Dotted arrows between the model parts indicate consistency relations; dotted arrows starting at "implementation invariants" represent logical dependencies; dashed arrows indicate program flow; continuous arrows denote modifying access.

as follows. Assuming validity and induction start preconditions on the initial configurations cvm^0 and h^0 we have:

$$\forall n\, \exists m \exists k \big(impl\text{-}inv(\delta^n_{cvm}(cvm^0), k, \delta^m_h(h^0)) \wedge cvm\text{-}sim(\delta^n_{cvm}(cvm^0), k, \delta^m_h(h^0)\big)$$

Observe that we link the specification down to the implementation and not vice versa, because the system has to simulate the abstract model completely. Only with this completeness argument the abstraction is well-defined.

2.2 Correctness Proof with VCC

To utilize our toolchain we have to formulate the above mathematical formulation of the theorem in the VCC specification language. Here we will restrict ourselves here to an overview of the most important components of the VCC model and identify, if applicable, corresponding parts of the formal model from the previous section. Note that all these components mentioned below, except for the kernel implementation, are introduced for specification and verification purposes using VCC and are not part of (and are not visible in) the PikeOS implementation.

- **abstract model**—*corresponding to* cvm *and* δ_{cvm}—the computational model that is implemented by the kernel code and the user steps. It contains the virtual machines of the user and an abstraction of visible kernel data structures in the ghost memory.
- **kernel implementation**—*corresponding to* $k(i)$—the code of the PikeOS kernel, written in C and assembly, annotated with VCC specification language
- **hardware model**—*corresponding to* h *and* δ_h—a definition of the PowerPC ISA, comes in two different flavors: (1) assembly semantics for inline assembly and assembly functions in the kernel, (2) a machine model for the user execution
- **user step simulator**—a C function simulating the possible executions steps of the user on the physical hardware, including exceptions, system calls and interrupts

See Fig. 3 for a schematic representation of the layers. We introduce the *user step simulator* in order to establish a simulation between the hardware and the virtual machines in VCC. It allows us to model all possible user execution steps and potential calls to the kernel.

Note also that there is a so called *high-level model* on top of the PikeOS system software which specifies the hypervisor use case (Sect. 1.1) from a bird's eye view (*not* part of the PikeOS kernel and thus *not* discussed here).

We conduct the inductive simulation proof by annotating the execution steps of the user machines and the kernel, stating that they simulate steps of the abstract model, i.e., all valid transitions specified for the abstract model must be implemented correctly by the low-level execution.

Moreover, data structure invariants of the abstract kernel model must be preserved by this execution. Obviously we must also prove that the simulation properties hold after initialization/boot.

3 A Typical Trace Through the Kernel

In the following, we give an example of an abstracted execution trace of user processes running on top of the kernel. This allows to identify the different types of instructions of the virtual machine, the phases of executing privileged instructions (via interaction with the kernel), as well as all the aspects that have to be taken into account to ensure the correct system behavior. We use this trace as a running example throughout the rest of this paper.

Without loss of generality, we can assume that our examplary system consists of the PikeOS kernel and two user processes running as virtual machines. They may run on top of the kernel and may share (portions of) their memory. Our verification target runs on a single CPU hardware; thus, only one user process is running at a time. The illusion of concurrency as seen by the user processes is achieved by preemptive time sharing using the kernel, as well as hardware interrupts.

Unless a user process invokes system mode features that are provided by the kernel or an interrupt occurs, the process's machine code instructions are executed directly on the physical machine. Each of these instructions is executed atomically (modulo exception handling), and interrupts can only occur at well defined points during execution of machine code, according to the PowerPC specification [4]. Handling the execution of these user mode instructions in our model is shown in the next section.

If a user process wants to make use of kernel functionality (e.g., to create a memory mapping), it invokes kernel system calls by using the "sc" assembly instruction, causing the control flow to jump to the system call handler inside the kernel. The kernel system call handler is responsible for saving relevant parts of the user context, enabling a return to user process execution after the system call has finished. Correctness properties of the context save and restore mechanisms are discussed in Sect. 5. After saving the user context, the system call handler invokes the appropriate C method in the kernel that implements the functionality of the system call. Functional correctness of these kernel primitives is the topic of Sect. 6.2.

When executing kernel code (with system mode privileges), for performance issues, preemption of the kernel thread is enabled at certain points. Also interrupts are enabled outside special code blocks that need to be executed atomically. How to deal with this interruptible kernel C code is shown in Sect. 7.

Before the kernel can return the control flow to the user process at the next instruction after the system call invocation, return values of the system call have to be assigned to the right registers and the non-volatile registers of the CPU context have to be restored (if necessary) to their old values before entering system mode.

The fact that the execution of system calls is interruptible makes it necessary to take into account the effect of concurrently running user processes, when reasoning about functional properties of a system call. We will address this issue in Sect. 7.2.

4 User Steps

When the kernel schedules a thread to run in user mode, machine code of that user is executed on the physical hardware. We imitate this by executing our user step simulator, which is implemented as a big case distinction over all user instructions that could be executed. A restriction is that only the user-visible portion of the CPU may be modified and only memory that was granted

to the respective thread may be accessed. However, the kernel must enforce that. Concerning the simulation proof, there are mainly two topics to care about for user steps.

4.1 Virtualization

Recall that the processor part of B-relation maps the virtual machines' CPU state either to the physical CPU state or kernel memory, where the CPU states of inactive users are stored. Thus for the running user process i we know that the CPU is coupled with the virtual machine $cvm.vm(i)$. In order to establish this simulation in VCC, user transitions must be propagated to the abstract model. By fixing transition rules (two-state invariants) for virtual machine components, we define all valid machine steps starting from a given state. If the implementation, i.e., user simulation, does not break these invariants we can conclude the induction step for the "running user" case.

Also, the execution of one user process must not affect the configuration of the remaining virtual machines in an unspecified way. This can be guaranteed by considering all instructions that write memory. Address translation mechanisms must have been set up in such a way that the virtual memories of the users only overlap for explicitly shared memory regions. Hence, for the abstract model, transitions on other users' virtual memories are only defined for shared memory accesses. A faulty implementation might enable a user process to write to another user's memory without causing an exception. However this bug would show up in a discrepancy between that other user's still intact virtual memory in the specification model and the corresponding modified physical memory region. Thus, in such a case, it were impossible to prove that B-relation still holds after the faulty step.

4.2 Kernel Robustness

In addition, we must show that neither the stored CPU contexts of the inactive user threads nor the kernel state are harmed by the running user step. This we can justify by another argument of separate memory regions. No user process may write to the kernel memory, i.e., the mapped physical address ranges of users and the kernel may not overlap and any memory violation of the user must lead to a page fault. A user process must also not be able to execute system mode instructions and any illegal action must result in an exception.

Memory separation properties of the user machines are covered by the implementation invariants on the address translation mechanisms in hard- and software, stating that the established virtual address regions do not overlap on the physical memory, except for the specified cases (e.g., shared memory). They must hold before and after user steps and kernel calls, i.e., those and other invariants must not be broken by user or kernel execution. It is thus necessary to look at traces through the kernel, which we do in Sect. 5 to 8.

5 Kernel Entry

Whenever there is a faulty instruction (exception), an external interrupt (e.g. ticker) or a system call invoked during the execution of the user process, the kernel must be called. In fact this is managed by the CPU interrupt service routine which masks all interrupts, disables address translation, and stores the old CPU state and the next sequential program counter. After that it jumps to the interrupt handler code of the interrupt that occured. We model this in the user step simulator by setting the corresponding hardware registers properly and calling the interrupt handler of the kernel for each interrupting case.

At their points of entry, the handlers for interrupts, exceptions and system calls are handled similarly. They mainly differ in the target address of the actual handler they are redirecting to and

concerning the set of register they save and restore. For exceptions the complete user context is saved; the interrupt handler stores only the volatile user registers as, e.g., general purpose registers 14–31 are preserved by the ABI (Application Binary Interface [13]). The system call handler only saves a minimal set of registers. There are three ways how a thread's user CPU context may be represented during kernel execution, depending on the cause for kernel entry.

1. The user context is completely stored in a kernel variable on the stack.
2. Only volatile registers are stored on the stack. The ABI guarantees that non-volatile registers are preserved by calls to C functions.
3. For system calls we do not need to restore the entire user context, hence only few registers (e.g., stack pointer) must be saved.

The B-relation has to incorporate this case split such that it holds after process save is completed.

The process save itself is implemented in assembly language that directly operates on the hardware model (we describe our approach for handling assembly code in the next section). However, as soon as we jump into the C portion of the kernel we do not prove a consistency relation between the kernel code and the user-visible registers anymore, because C code intensely clobbers on general purpose registers and the like. Besides not having exact compiler semantics it would also not be feasible to model all these transitions on the hardware layer for our current toolchain. Instead we assume compiler correctness and annotate all C-functions, stating that they modify the user-visible registers, but restore the non-volatile ones upon return according to the ABI.

6 Common Kernel Operations

In the following we examine typical verification scenarios that we meet on a trace through a kernel like that of PikeOS.

6.1 Assembly Primitives

Naturally, a kernel incorporates a good share of low-level, hardware-related functionality, which necessarily has to be implemented by assembly code. Such functionality is often encapsulated in so-called primitives and a general methodology to verify them has been developed during the first Verisoft project [10]. Nevertheless we have to adjust this approach to meet the requirements of our new toolchain.

Methodology The first step towards verifying assembly code is to define the instruction set architecture (ISA) of the processor used in the project. This happens to be a Freescale MPC5200 processor with a POWERPC G2_LE core in our case. The ISA is specified by C data structures representing the processor register state and specification functions for each instruction modeling their semantics. Physical memory is basically modeled by a pointer to the starting address and a size. Using pointer arithmetic and taking address translation into account, virtual memory regions can be bound to physical ones via implementation invariants.

Assembly language statements are automatically replaced by their hardware model counterpart functions, such that we construct a C program that works on the hardware state and that again can be verified using VCC. Note that the hardware model resides in a proper C memory region outside the regular address range and thus can be described as an *external state*. We cannot put the hardware state into ghost memory because this would prohibit data exchange between C variables and registers.

Our method for verifying inline assembly is an extension of the general assembly methodology desribed above. The main additional effort is to properly initialize registers with the contents of C

variables and to write back results from the registers to the variables at the end of the assembly block, according to the ABI and the parameters specified in the `__asm__` statement.

Correctness The justification for correctness of our approach relies on a number of arguments. Observe that adding the hardware state to the program as an external variable and replacing assembly code by the calls to specification functions does not alter the semantics of C statements in the remaining program (as provided by VCC). Our approach is to be seen as an amendment to the VCC verification methodology. For plain register operations, the correctness of assembly statement verification concerns mainly hardware correctness, i.e., the processor must correctly implement the execution of hardware instructions according to their specification. However greater care must be taken for memory and jump instructions.

Whenever assembly code accesses C data structures the coupling between the virtual and the physical memory must be made explicit. That means that we need to argue that certain C variables are residing in the physical memory at their allocated addresses and that, e.g., any modification of physical memory will propagate to the corresponding variable in the C memory model. This is guaranteed by the compiler's allocation function, which maps C variables to addresses and the proper setup of address translation for the kernel virtual memory.

Jump instructions come in different flavors and so do their translations to C code. Local jumps can be implemented as `goto`'s, while far and implicit jumps may be realised by interpreting jump addresses as function pointers and dereferencing them. This of course requires that jump addresses are stated as labels or function names, which is mostly the case for well-behaved assembly code, and that the compiler correctly translates labels and function names back to jump addresses. In case of local jumps with explicit addresses labels can be inserted easily. However, our methodology cannot handle far jumps into the middle of function bodies automatically. Here manual editing and code refactoring might be needed.

Besides the compiler correctness, we also rely on the correct implementation of our hardware model and assembly translation algorithm.

Typical Applications In the PikeOS kernel we face assembly code in many places. Naturally kernel entry and exit routines as well as thread switch are implemented as assembly functions. Furthermore there are smaller functions which enable access to hardware registers and system configuration or facilitate assembly code to speed up program execution. Several assembly primitives also involve copying data, e.g., from kernel to user address space and vice versa. Besides their functional correctness we mainly have to prove that these routines keep the simulation relation and implementation invariants intact, i.e., they do not modify memory regions outside their specified range.

6.2 Kernel Functionality

The biggest effort in the simulation proof is functional C verification, since the simulation theorem must be verified for all possible program steps, i.e., all code traces that correspond to transitions on the abstract kernel model. Apart from the functional correctness of the C functions we have to ensure not to touch other user or kernel thread contexts in an unspecified way while we are running in the kernel, in order to verify the simulation theorem. We also must verify that the various implementation invariants on the C data structures are maintained by the kernel implementation (cf. Fig. 2). We give two examples for module-specific requirements.

Memory Management Since the kernel must ensure the correct simulation of user machines, a big part of its functionality concerns memory mangement. Typical tasks involve dynamic allocation of memory during boot or after initialization, modifying mappings of memory regions, e.g., in order to set up communication between user processes and handling page faults. Generally these routines are rather hardware-oriented and the usual precautions of not damaging

simulation properties have to be considered when dealing with assembly code and direct access to physical memory. Moreover, there are specific implementation invariants on the kernel data structures that realize the assignment of kernel memory resources and user-space memory resources and that must be preserved by the corresponding C code.

User Process Management Another significant portion of the kernel functions deal with user process administration, e.g., to impose a hierachy on a set of user processes (abilities), control scheduling, delivering messages and events to users etc. Sometimes such routines are little more than plain getter- or setter-functions, while often we also have to deal with more generic data structures like lists, queues or trees. Once more verification mainly has to ensure that data structures remain in a well-defined state. In the case of generic data structures it is imperative to find a specification for the functions manipulating these objects that is general enough to be applied to all use cases.

Assumptions for C Verification We use the semantics of C here as the main ingredient, which is incorporated in our tool VCC. This semantics relies naturally on compiler and hardware correctness, but also on the assumption that the code runs on sequentially consistent memory. For a single processor machine this requirement is quite trivial and boils down to claims on the configuration of the address translation mechanism and software conditions to ensure cache consistency (e.g., for self-modifying code).

While we need to care about memory consistency, we do not map the execution of C code down to the physical memory and processor context, as doing so would be infeasible for our toolchain, even if we had an exact compiler definition. Thus we can only rely on ABI guarantees when returning from C functions to assembly language context.

7 Concurrency

For the scalability of the verification task to large software systems, modularization is vital. The VCC methodology supports this by being able to verify a method in a concurrent setting without imposing a fixed scheduling or having to inspect all possible traces through the system. The effects of the environment of the currently running thread are rather described by transitions on shared data structures.

Moreover, as preemption may occur due to external interrupts, the exact location of the interrupt in terms of C statements of the current thread is not known. It may even be below the granularity of the C statements, as interrupts may occur after each assembly instruction. To be able to use the VCC methodology to reason about C code with these kind of interrupts, we introduced the notion of interruptible C semantics and justified why it is appropriate to model concurrency in this way—this topic, however, is out of the scope of this paper.

As a result, we make scheduling of other threads implicit when verifying a method which is preemptable—the effects of the other scheduled threads become visible when the current thread accesses shared state. As a consequence of this, explicitly calling the scheduler in the kernel is equivalent to a no-operation.

7.1 Locking

In our verification setup, concurrency originates only from interrupts and explicit scheduling by the kernel, as the system under verification has only a single processor. For performance reasons, not only processes running in user mode are preemptable, but also kernel threads may be interrupted at certain points of the execution. Access to the shared physical hardware by multiple user processes has to be rigidly controlled by the kernel in order to maintain non-interference.

To enforce a certain access policy to shared resources, the kernel is designed to use the following access mechanisms:

1. Reading and writing of data is done through locks with arbitrary locking granularity to achieve high performance by locking only the relevant state of the system that may be accessed concurrently
2. Disabling possible preemption by either using hardware features to disable external interrupts of the system completely or by enforcing that the scheduler of the kernel is not invoked
3. Memory separation, e.g., if it is known that all concurrently running processes can each only access a distinct part of the system, as common in interrupt handlers.
4. Access to shared memory by lock-free algorithms.

In the following, we discuss how to model these access mechanisms with the help of VCC in order reason about access to system state done through these mechanisms in a sequential fashion.

For the first access mechanism, namely locks implemented in the kernel, we re-use the verification methodology for locks provided in [7] that is available for VCC. For this purpose, we have to add VCC specification annotations to the PikeOS lock implementation. Acquiring a lock in this methodology grants the currently running thread exclusive access (on the specification level) to the object protected by the lock. This makes it possible to reason sequentially about access to this object.

In a similar fashion these specification primitives are used to handle the second variant of access control to shared resources, by limiting concurrency. A typical way to get exclusive access to memory on single processor systems is by restricting the sources of interrupting events. Two instances of such a constraining interrupt policy are disabling external interrupts altogether and the alternative of just disabling preemption by the kernel (i.e., no other user process is scheduled by the kernel, but hardware interrupts are still handled).

Disabling external interrupts (e.g., the timer interrupt) on a single processor machine ensures sequential execution of the currently running kernel thread, by means of the hardware, until interrupts are enabled again. This behavior corresponds (on the specification level) to a locking scheme with a mandatory lock, i.e., the other threads of the system do not have to cooperate in order to enforce the lock. This makes it possible to model this access policy using VCC by the same locking mechanism as used with ordinary locks, by transferring exclusive access of the whole state to the currently running thread for the duration that the interrupts are disabled. In the source code of the kernel, the primitives that disable resp. enable these interrupts are amended by the ghost code that performs the necessary transfer of exclusive access ("ownership")—again these additional annotations do not belong to the PikeOS implementation but to the specification level.

7.2 Specification of System Calls

One major ingredient of the overall correctness of PikeOS is the correct implementation of the system calls of the kernel, according to their specification given by the PikeOS API documents.

In case of sequentially executed code, the specification of the functionality of the code is done (as usual) via pre- and postconditions, describing the effect of the code on the state of the system.

In the case of system calls, the assembly code of the system call is executed sequentally handler up to the call of the C method that implements the appropriate system call functionality. Therefore the precondition for the system call method can be stated exactly.

After the system call handler has set up the state for the execution of the actual system call, kernel execution is again interruptible. In this concurrent case, specifying code with pre- and postconditions becomes inconvenient: from a given state that satisfies the precondition of a concurrently executed method, not only the effect of the system call under verification has to be taken into account when formulating the postcondition, but also an arbitrary number of steps of the environment.

Therefore, the specification of system calls is provided as definitions of transitions (two-state invariants) on the abstract model at the top-level of the system. The effect on the real hardware and implementation state of the kernel is then propagated via coupling invariants between abstract and concrete system state.

The initial and final states of such transitions are states of the system that are visible to the user processes, i.e., internal computations of the kernel are visible to the environment as one atomic state update, as long as no interrupt inside the kernel thread is possible. For the verification of system calls, with a structure as defined in the beginning of this section, there may exist several sequentially executed code blocks with interruptible code segments in between. In the interruptible code parts, the kernel may schedule other user threads running in user mode, which can witness the change of state due to the partly executed system call.

On the abstract level, such a system call would therefore correspond to several transitions, one for each non-interruptible code block in the system call. On a higher level, it has to be shown that the effect of performing all of these transitions that constitute one system call, interrupted by transitions of the environment, has the desired functionality as described in the kernel API documentation.

8 Thread Switch and Return to User Mode

8.1 Context Switch

In the formalism developed in the first phase of the Verisoft project, it was possible to describe the effect of the context switch, i.e., switching the CPU contexts of C threads, in an explicit manner. This is because the C state of the threads was in the old methodology—unlike in VCC—explicitely visible to the verification engineer as a part of the model. Due to the availability of the formal compiler semantics one could also define consistency relations between the machine state and the C state of threads.

For the verification of PikeOS, we rely on the correctness of various components, such as the C compiler, to be able to focus on large scale verification of a system like the PikeOS kernel. In addition, using the VCC methodology alone it is not possible to reason about the impact of a running C thread on the hardware level. Due to the modularity of verification, even if there was a coupling relation between C- and hardware-level within VCC, the view is restricted to the current thread and the method under verification. The contexts of previous and concurrent method invocations are invisible to the verification engineer. Therefore, we have to model the context switch implicitly using VCC's built-in features.

The actual (assembly) implementations of the process save and restore mechanisms in PikeOS each have a corresponding transition on the system state and can be verified using the assembly verification technique described before.

In the currently running kernel thread that performs the context switch we have, however, a gap between loading the context of the next thread to be scheduled and eventually returning to the original thread via another context switch. Between those two points, an arbitrary number of steps of the other threads of the system are performed. This can be modeled on the specification level by giving up exclusive access to the system state at the point when switching the contexts. From the caller site, the method that switches the contexts thus performs an arbitrary number of transitions on this state, according to the system transitions defined by two-state invariants.

One of these transitions describes the effect of restoring one of the stored CPU contexts of the threads in the system. If we assume a fair scheduler, eventually one such transition will be performed that restores the initial thread that performed its process save. This justifies handling the context switch as a (special kind of) method invocation from which control flow eventually returns.

Between process save and restore of a particular thread, a trace of transitions specified on the abstract model of the system is executed. These transitions include the potential save and restore steps of all other threads. As we verify these methods for an arbitrary thread, we can deduce that context switch works for any fair scheduling of C threads. However we have to argue formally on the meta level that this approach is actually equivalent to a proof of the switch to another thread on the source code level.

8.2 Back to Userland

There are several preemption points and places where the kernel execution itself could be interrupted. Nevertheless at some point the kernel finishes the handling of the user interruption and jumps back into user space.

Back in the user step simulator, we have to assure that the global system and kernel invariants still hold and assert the postcondition of the particular interrupting user step that was executed. Moreover, the kernel call must not have violated the two-state invariant of the virtual machines. That means for each kernel execution that is visible for the user there is a sequence of system transitions, each with a corresponding two-state invariant in the user model, which reflects the transition's effect on user state. Note that there might have been intermediate scheduling and thread switches. Thus, a postcondition of the kernel call expressing functional properties can hardly be specified. We can mainly guarantee that implementation invariants and simulation relations still hold. The functional correctness of the kernel calls is encoded in the transitions on the abstract model.

Upon returning to user mode, also the context of the user process has to be correctly restored to the state before the interruption—modulo kernel functionality that intentionally modifies the user context of other threads.

To get stronger guarantees as postconditions of a kernel call, assumptions about the processes running on top of the kernel have to be made—e.g., either the possible transitions of user programs have to follow a certain policy or there may be a designated, trusted process of the system with special rights that enforces certain behavior of the other processes.

9 Conclusion

In this paper, we have outlined the overall correctness property of the PikeOS kernel, which is formulated as a simulation relation between the concrete PikeOS system and an abstract model of it. In addition, an overview of the different components and abstraction levels of the abstract model of the system was presented, as well as coupling relations between the layers (e.g., the B-relation) and the notion of integrity invariants of the system state.

For the proof of this simulation relation, we identified a number of "ingredients" that have to be taken into account in addition to the verification of functional properties of the kernel implementation with the VCC verification tool. This includes correctness properties of address translation and memory separation, techniques to handle (inline) assembly code, and assumptions resp. requirements on various components like the compiler, hardware and implementation policies like the PowerPC ABI. Furthermore, we gave a brief look on how to deal with concurrency and locking using VCC, as well as how to model the thread switch within the PikeOS scheduler.

As we have pointed out, there are still other correctness properties to consider, which are out of the scope of this paper or belong to future work. For the correctness proofs presented in this paper to be valid, we have to define a memory consistency model (this imposes, e.g., restrictions on self-modifying code and requirements on cache transparency) and provide meta-arguments about correct tool- and methodology-usage (see Sect. 6.1 and Sect. 7). These items are currently work in progress. In Sect. 6.2, we introduced several typical kernel functionalities and

their relevance for the overall correctness proof. One functionality which is not yet put into this context and is not discussed in this paper is inter-process communication. One possibility to proceed beyond the scope of the Verisoft XT project would be, instead of proving the correctness of the PikeOS microkernel implementation alone, to extend the proof to the implementation of the PikeOS system software to obtain a formally verified hypervisor implementation (see Sect. 8.2).

We have presented a single processor setup. With regards to a multiprocessor setup, we benefit from that we are already using locks as VCC methodology designed for concurrent verification to deal with the (interruptible) bulk of the C code verified for functional correctness. Some additional effort would include review of bottom-level assumptions (e.g. memory consistency model) and of meta-arguments at the top.

Acknowledgement

We thank Jacques Brygier, Knut Degen, Sergey Tverdyshev, Michael Werner, Alexander Züpke (SYSGO AG), Dilyana Dimova, Artem Starostin (Saarland University), Markus Dahlweid, Thomas Santen, Stephan Tobies (European Microsoft Innovation Center) for support and comments.

References

1. Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Better avionics software reliability by code verification. In *Proceedings, embedded world Conference, Nuremberg, Germany*, 2009. ISBN 978-3-7723-3798-7, `http://www.uni-koblenz.de/~beckert/pub/embeddedworld2009.pdf`.
2. Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. Available at `http://research.microsoft.com/vcc`.
3. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 14th International Conference, Budapest, Hungary*, LNCS 4963, pages 337–340. Springer, 2008.
4. Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture*, 3rd edition, September 2005.
5. Freescale Semiconductor. *MPC5200B User's Manual, Rev. 1.3*. Sep 2006. `http://www.freescale.com/files/32bit/doc/ref_manual/MPC5200BUM.pdf`.
6. Mauro Gargano, Mark A. Hillebrand, Dirk Leinenbach, and Wolfgang J. Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.
7. Mark A. Hillebrand and Dirk C. Leinenbach. Formal verification of a reader-writer lock implementation in C. In *Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 123–141. Elsevier Science B.V., 2009.
8. Gerwin Klein. Operating system verification — an overview. Technical Report NRL-955, NICTA, Sydney, Australia, June 2008. Available at `http://wwwbroy.informatik.tu-muenchen.de/~kleing/papers/os-overview.pdf`.
9. Microsoft Research. VCC homepage. `http://vcc.codeplex.com`.
10. Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In G. Klein R. Huuck and B. Schlich, editors, *3rd International Workshop on Systems Software Verification (SSV08)*, volume 217 of *ENTCS*, pages 169–185. Elsevier Science B. V., 2008.
11. SYSGO AG. PikeOS RTOS Technology. `http://www.pikeos.com`.
12. Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Dept. Computer Science, Saarland Univ., 2009. `http://www-wjp.cs.uni-sb.de/publikationen/Tsy09.pdf`.
13. Steve Zucker and Kari Karhi. *System V Application Binary Interface: PowerPC processor Supplement*. SunSoft, Mountain View, CA, USA, 802-3334-10 edition, Sep 1995. `http://refspecs.freestandards.org/elf/elfspec_ppc.pdf`.