Verified Process-Context Switch for C-Programmed Kernels

Artem Starostin ^{*} and Alexandra Tsyban ^{**}

Computer Science Dept., Saarland University, Germany
{starostin, azul}@wjpsrvr.cs.uni-sb.de

Abstract. A context switch — an act of saving and restoring the state of a CPU such that multiple processes can share a single CPU resource — is an essential feature of multitasking operating systems. Commonly computationally intensive and necessarily accessing hardware registers, context-switch procedures are implemented as inline assembly portions in C-programmed operating-system kernels. Feasible verification of operating systems is usually attempted in some kind of C semantics. However, seamless verification of kernels requires reasoning about context-switch routines in semantics of assembly language. At the end of the day, both semantics meet together in an overall correctness theorem of operating system. The task of formal integration of correctness results achieved on different semantical layers is challenging but inevitable for systems verification.

The paper describes a formal approach to pervasive reasoning about interleaved computations of user processes and a C-programmed kernel. The interleaving is achieved by context-switch procedures implemented in inline assembly. We report on the correctness proof of the context-switch procedures and elaborate on our experience in formal integration of this result into the correctness proof of CVM, a verified framework for microkernel programmers.

1 Introduction

Pervasive systems verification [2] — an act of proving correct an entire computer system from gates to software — is a grand challenge [16] undertaken by the Verisoft project [20]. In our experience, the complexity of the problem turns out to be not in verification of individual components comprised by a system, but rather in *formal* integration of different correctness results achieved separately.

One representative example where different formal theories meet together is a correctness proof of process-context switch. Context-switch procedures save and restore the state (content of visible registers) of a CPU such that multiple processes can share a single CPU resource. This puts them at the heart

^{*} Work was supported by the International Max Planck Research School for Computer Science (IMPRS-CS).

^{**} Work was supported by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project.

of multitasking operating systems. Convincing arguments about correctness of context switch tie together at least the following formal theories: (i) a model of processor on which user processes and an operating system run, (ii) a virtual memory model used to implement separate address spaces of user processes, (iii) some sort of C semantics used to model computations of operating system, and (iv) inline assembly semantics used to reason about context-switch routines as the latter access hardware registers not visible via C variables. These formal theories have been developed in the scope of Verisoft [4, 5, 15, 19], however putting them together in a correctness theorem of context switch requires additional investigations. The complications appear when one needs to switch from a computation of model x to a computation of model y . In order to reason about y , it turns out that additional invariants about it have to be preserved under the computation of x . Determining such invariants and proving them is a tedious scientific task.

The paper describes a pervasive correctness proof of process-context switch. Our context-switch procedures have C signatures, their functionality is implemented as inline assembly portions. This makes possible to plug them into C implementations of operating-system kernels on the source code level. In order to guarantee correctness of context-switch procedures as parts of C-programmed kernels it is not sufficient only to show their functional correctness. In order to proceed with kernel computations additional invariants about the kernel have to hold during computations of user processes. We introduce a concept of *weak kernels* — together with a correctness relation towards a hardware model — a formalization of suspended kernels during user executions. Our correctness theorems of context-switch procedures formally describe how an active kernel is constructed out of weak one and vice versa. This result makes possible to produce a pervasive seamless proof of C-programmed microkernels, like those that are considered in Verisoft. The paper gives details on integration of correct context-switch procedures into CVM, a framework for building verified kernels [13]. All material presented in the paper is supported by formal theories ¹ in Isabelle/HOL, a verification environment common to Verisoft.

We believe the contributions of the paper fit into the program of the Verified Software Initiative [10], a state-of-the-art plan for research communities with the goal of making verification a core technology for developing reliable software. The authors personally see contributions to two fields: (i) theory unification, and (ii) experience reports on verification of realistic software systems. The former is confirmed as follows. We have not only exploited existing formal theories, rather evolved and adjusted them in a way they *formally* fit together. We have found and formalized additional invariants that made the integration possible. We have produced a formal proof for a critical part of realistic operating-system microkernel running on a verified processor — this argues for the second contribution.

Related Work. Much and, to the best of our knowledge, pioneering work on formal verification of process-context switch has been done by the FLINT group

¹ The theories are available at <http://www.verisoft.de/VerisoftRepository.html>.

at Yale University. [18] describes how XCAP — a verification framework implemented in the Coq proof assistant — is applied to certify an x86-assembly implementation of machine-context management. [8] shows the FLINT’s approach to integration of context switches with thread implementation. The correctness criteria considered are formulated, in contrast to the current paper, only at the assembly-language level: no results on integration of object code correctness into a high-level programming language are reported. Conversely, a large number of research groups working on OS verification consider only code implemented in high-level programming languages. The Singularity project builds an OS using Sing#, a type-safe programming language [11]. The NICTA group considers C implementation generated from an OS prototype designed in Haskell [7]. Both rely on unsafe portions of assembly code for context-switch.

Outline. We start in Sect. 2 by defining computational models of CPU, user processes, and a kernel. In Sect. 3 we describe the interaction between the models and state our verification objective. We continue in Sect. 4 by outlining our verification approach and presenting the correctness theorems of context switch. Finally, we conclude in Sect. 5.

Notation. Let bv denote the set of all bit vectors of length 32. We denote a list with elements of type t as t^* . An empty list is denoted by $[]$. A list x is accessed by $x[i]$. A sublist of a list x is $x[i..j]$. Concatenation of lists or list elements x and y is $x \circ y$. For a record x with components a , b , and c we agree on $x = x.(a, b, c) = (x.a, x.b, x.c)$. Record parts are accessed in the same fashion, e.g., $x.(a, b) = (x.a, x.b)$. For a natural number d we denote by $m_d(a)$ the content of d consecutive memory cells starting at address a .

2 Computational Models

We consider a scenario where a number of user processes interact with a kernel on a shared CPU resource. We use a *physical machine* [4] to model the CPU and *virtual machines* [5] to model user processes. We use the *C0 language* [15], a slightly restricted dialect of C developed in Verisoft, to implement the kernel. Note, that kernel implementation necessarily contains inline assembly portions as the kernel accesses hardware registers not visible to C0 variable. The C0 small-step semantics is used to model kernel computations. Next, we give details on each concept.

2.1 Physical Machine

The physical machine is a sequential abstraction of the VAMP hardware [4] — a formally verified processor with memory management units, out-of-order execution, and memory-mapped devices — as seen by a system software programmer. VAMP implements the DLX instruction set [17]. The model state pm is the record with the following components: (i) the normal $pm.pc :: bv$ and the delayed

Alias	Name	Description
<i>sr</i>	status reg.	stores an interrupt mask
<i>esr</i>	exceptional status reg.	stores the last interrupt mask before an interrupt
<i>eca</i>	exceptional cause reg.	stores masked interrupt cause
<i>epc</i>	exceptional <i>pc</i>	stores the last <i>pc</i> value before an interrupt
<i>edpc</i>	exceptional <i>dpc</i>	stores the last <i>dpc</i> value before an interrupt
<i>edata</i>	exceptional data reg.	stores data specific for a certain interrupt
<i>pto</i>	page-table origin	origin in the page table for address translation
<i>ptl</i>	page-table length	number of allocated physical memory pages
<i>mode</i>	mode reg.	value 0 is system, any other value is user

Table 1. Special purpose registers

$pm.dpc :: bv$ program counters used to implement the delayed branch mechanism, (ii) the general purpose $pm.gpr :: bv^*$ and the special purpose $pm.spr :: bv^*$ register files, and (iii) the byte-addressable physical memory $pm.m :: bv \mapsto bv$. The general purpose register file contains 32 registers: $pm.gpr[0]$ always contains zeros, and the last three registers are reserved for the global, the stack, and the heap pointers of a program. We abbreviate them as $pm.gp$, $pm.sp$, and $pm.hp$. The special purpose register file contains 9 registers described in Table 1. For each alias x we abbreviate access to the special register $pm.spr[x]$ as $pm.x$.

The computation of the model is defined by the transition function δ_{pm} which maps the current processor configuration pm and a bit vector of external inputs eev to the next configuration $pm' = \delta_{pm}(pm, eev)$. Due to the lack of space we do not present a complete definition of the step function, but rather an idea behind it. For formal details cf. [3]. Executions are possible in two modes: user and system. The processor switches to the system mode, denoted by $sys(pm)$, when an interrupt, either internal or external, arrives and the JISR (jump to interrupt service routine) signal, denoted by $jisr(pm, eev)$, is activated. The mode switch in the opposite direction happens when the RFE (return from exception) instruction, denoted by $rfe(pm)$, is executed. In user mode a memory address a is virtual and is subject to address translation. The translation either redirects to the computed physical memory address or generates a *page fault* interrupt which signals that the desired page is not in the physical memory. The decision is made by examining the *valid* bit $v(pm, a)$ stored in *page tables* maintained by memory management units of the physical machine. When on, it signals that the page storing the virtual address a resides in the main memory, otherwise, it is on a hard disk.

Interrupts. Internal interrupts comprise an illegal instruction, a misaligned access, page faults, overflows, and the *trap*. The latter is used to implement system calls to kernel libraries. External interrupts are signals from devices. VAMP supports maskable interrupts. For a maskable interrupt i , the bit $pm.sr[i]$ stores its mask. The *cause* register $ca(pm, eev)$ is the bit vector which stores raised interrupt signals. The *masked cause* register $mca(pm, eev)$ is defined as a bitwise

conjunction $ca(pm, evv)[i] \wedge pm.sr[i]$ over all maskable interrupts i . If at least one bit of $mca(pm, evv)$ is on the $jisr(pm, evv)$ is activated.

Semantics of JISR and RFE. On $jisr(pm, evv)$, the program counters $pm.(dpc, pc)$ are set to the kernel start address ($kstart, kstart + 4$). The exceptional versions of registers $pm.(edpc, epc, esr)$ are assigned their normal versions. Register $pm.eca$ stores the masked interrupt cause $mca(pm, evv)$. Finally, $pm.edata$ stores data needed to process a particular kind of an interrupt, e.g., an address of a page missing in the main memory for page-fault interrupts, or a system call code for the trap interrupt. The mode is switched to system. On $rfe(pm)$ the normal versions of registers $pm.(dpc, pc, sr)$ are assigned their exceptional versions. The mode is switched to user.

2.2 Virtual Machines

A virtual machine is a hardware abstraction with an illusion of address space exceeding the physical memory. The state of a virtual machine vm is a record comprising the same components as the physical machine. There are different rules for accessing special purpose registers: only reading is allowed. As for the rest, the semantics of virtual machines simply implements the DLX instruction set architecture. There is no address translation, hence page faults are invisible. There is no interaction with devices. Transitions are modeled by the function δ_{vm} which takes the current virtual machine configuration vm and yields an updated one $vm' = \delta_{vm}(vm)$.

2.3 C0 Small-Step Semantics

The C0 concrete syntax and visibility rules for variables are very similar to standard C. Operational semantics, though, is similar to Pascal. The main language restrictions compared to C comprise absence of pointer arithmetic, side effects, and pointers to functions and local variables. A list of assembly instruction il can be embedded in C0 by a special statement $asm(il)$.

A C0 program π is a record with the following components: (i) a global symbol table $\pi.gst$ which is a list of variable names together with their types, (ii) a type name environment $\pi.te$ which maps type names to types, and (iii) a function table $\pi.ft$ which maps function names to functions. A function is represented by symbol tables for parameters and local variables, the function's return type, and the function's body.

Configurations c of the C0 small-step semantics are records with two components: the memory configuration $c.mem$ and the program rest $c.pr$. The program rest stores those statements which still have to be executed. The memory configuration is a record comprising the following components: (i) a global memory frame $c.mem.gm$, (ii) a stack of local memory frames $c.mem.lm$, and (iii) a memory frame for heap variables $c.mem.hm$. A memory frame m consists of a symbol table $m.st$ which lists the variables of the frame and of a content function $m.ct$ which maps addresses to memory cells. A single memory cell can store values

of elementary types. Values of aggregate types are stored flattened as a consecutive sequence of memory cells. Executions of the C0 small-step semantics are modeled by the function δ_{C0} which yields for a given C0 configuration c and a program π either a special error state or an updated configuration $c' = \delta_{C0}(c, \pi)$. A computation ends in the error state if, for instance, expression evaluation detects an error in the program, e.g., null-pointer dereference, or out-of-boundary array access. Details are available through [14].

3 Verification Objective

CVM (communicating virtual machines) [9] is a computational model for concurrent user processes interacting with a generic microkernel and devices. CVM is implemented in C0 with inline assembly as a framework featuring virtual memory support, memory management, and low-level inter-process and devices communications. The framework can be linked on the source code level with an abstract kernel, an interface to users, in order to obtain a concrete kernel, a program that can run on a target machine, like the kernel we discuss in the paper. CVM has been verified to a large extent within Verisoft [13].

Let a C0 with inline assembly program π_k be a concrete kernel implementation. Let k be a C0 small-step configuration corresponding to π_k . The kernel interacts with N user processes $up[1..N]$. Each process $up[i]$ is modeled by a virtual machine. When a user execution is interrupted the hardware generates the JISR signal which triggers the kernel start. Kernel executions begin with the SAVE procedure which stores the content of the hardware registers to kernel variables. After that, depending on the interrupt, kernel performs one or several jobs: handles interrupts, invokes system calls, schedules the next process. Kernel execution ends with the call of the RESTORE procedure which copies the context of the scheduled user process to hardware registers and executes the RFE instruction.

The CVM verification objective is to justify correctness of (pseudo-)parallel executions of user processes and the kernel on the underlying hardware. This is expressed as a simulation theorem between the physical machine and virtual machines interleaving with the kernel: for all steps of an interleaved execution of the kernel and user processes there exists a number of hardware steps after which an appropriate simulation relation holds. The simulation relation is split into two parts: each relates a hardware configuration either to a kernel configuration, or to configurations of user processes. One of the most interesting proof cases is switching between executions of different user process by means of a kernel invocation. The task is tricky since correctness theorems of SAVE and RESTORE (i) tie together different semantics used to model user processes and the kernel, (ii) formalize a connection between correctness relations for users and the kernel, and (iii) involve inline assembly semantics to prove the functional correctness.

Correctness criteria of user processes must reflect two main features realized by CVM: memory virtualization and multitasking. Correct memory virtualization ensures that each user process has a notion of its own isolated virtual

memory exceeding the physical memory. CVM implements memory virtualization by means of demand paging: [1] reports on formal verification of the used paging mechanism. Multitasking is a method by which multiple processes share a single computational resource like CPU. Correct multitasking ensures reassigning a CPU from one process to another. This paper focuses on correctness of multitasking implementation in CVM.

The rest of this section is organized as follows. We give some details on the kernel implementation, discuss the notion of contexts, and describe our implementation of context-switch procedures. Next, we state the correctness relations for the kernel and user processes and describe at which parts of an interleaved run which relations hold. Finally, we discuss how to combine correctness criteria by introducing invariants over a suspended kernel.

3.1 Kernel

We consider a non-preemptive kernel. On a user interrupt the kernel needs to know which process has been preempted. When `RESTORE` is invoked the kernel needs to know which process has to be resumed. In order to deal with it, the kernel has the variable `cup` of unsigned integer type storing the identifier of the last executed process. The kernel scheduler assigns the new value to `cup`, thus the process `up[cup]` will be resumed.

We allow dynamic memory allocation for the kernel. The kernel contains the variable `kheap` of integer type which stores the amount of memory allocated by the kernel on the heap. Base addresses of kernel global and stack memories are defined by the constants `kglobal` and `kstack`, respectively.

3.2 Contexts and Context-Switch Procedures

We split registers of user processes into two groups: those whose values affect user executions and those that are used only to implement the interrupt mechanism.

Registers Relevant for User Executions. This group comprises delayed and normal program counters, 31 general purpose registers excluding `gpr[0]` — according to the DLX instruction set architecture it always contains zeros —, the status register, page-table origin, and page-table length registers. We call these registers the *context* of a process. For a virtual machine `vm` it is formally retrieved by the function $ctx(vm) = vm.pc \circ vm.dpc \circ vm.gpr[1..31] \circ vm.sr \circ vm.pto \circ vm.ptl$. We store contexts of processes in a special data structure, called the *process control blocks*. The kernel symbol table $\pi_k.gst$ includes the variable `pcb`, an integer array with $36 \cdot N$ items capable to store all contexts of user processes.

Registers Needed to Process Interrupts. In order to react appropriately to an interrupt which occurs during a user execution the kernel has to know the interrupt cause and exceptional data, e.g., if an interrupt happens due to the trap instruction the kernel has to know its parameter in order to execute the right system call. Because of that we declare two variables in π_k : (i) the variable `eca`

of integer type used to store the current exceptional cause register, and (ii) the variable *edata* of integer type storing data for interrupt handling.

The context-switch procedures are C0 functions with inline assembly bodies. The SAVE procedure starts by computing the memory address $a = 4 \cdot \&(pcb[cup \cdot 36])$. Starting at address *a* we write to the memory *pm.m* consecutively the contents of the following registers: (i) *pm.epc* and *pm.edpc* as they are equal to *pm.pc* and *pm.dpc*, and (ii) *pm.gpr*[1..31]. Note, that we do not save registers *pm.(sr, pto, ptl)* because they could not be modified by a user. Next we save registers *pm.eca* and *pm.edata* to corresponding kernel variables and assign *pm.gp*, *pm.sp*, and *pm.hp* the values of *kglobal*, *kstack*, and *kheap*. The last statement of the SAVE routine is a C0 call to the kernel dispatcher.

The RESTORE function is quite symmetrical to SAVE. In the C0 portion of the function we compute an offset in *pcb* corresponding to the next scheduler process identifier *cup*. By means of inline assembly we remember the value of the kernel heap pointer in variable *kheap* and write the context of the next scheduled process from *pcb* into hardware registers *pm.(epc, edpc, gpr*[1..31], *esr, pto, ptl)*. Note, that *pm.sr*, *pm.pto* and *pm.ptl* registers are updated as the kernel can modify their values, e.g., when it allocates additional memory for some user process. The last instruction of RESTORE is RFE.

Assembly bodies of the context-switch procedures contain all in all about 100 instructions.

3.3 Kernel Correctness Relation: C0 Consistency

Our correctness relation for the kernel maps a C0 small-step semantics state to a state of the physical machine. A C0 configuration *c* corresponding to a program π is related to an underlying physical machine *pm* by the compiler simulation relation $consis(alloc)(\pi, c, pm)$ parameterized over an allocation function *alloc* which maps C0 variables to the physical memory cells. The relation describes consistency of (i) code, (ii) control, and (iii) data. The code consistency $consis_{code}$ requires that the compiled code lies at the correct address in the memory *pm.m*. The control consistency $consis_c$ states that the delayed program counter *pm.dpc* points to the start of the translated code of the first statement of *c.pr* and that return addresses of all stack frames are correct. Essentially, the data consistency $consis_d$ is a conjunction of the correctness relation for: (i) allocation, $consis_{alloc}$: the allocation function *alloc* conforms with the allocated base address of global and local variables, and the reachable portions of the heaps in *c.hm* and *pm.m* are isomorphic, (ii) values and pointers, $consis_v$ and $consis_p$: the respective variables and pointers of *c* and *pm* have the same values, (iii) registers, $consis_r$: the heap pointer *pm.hp* points to the first free address of *c.mem.hm*, and the global resp. the stack pointers *pm.gp* resp. *pm.sp* refer to the beginning of the global resp. top local memory frames of *c*. A scheme of the C0 consistency relation is depicted in Fig. 1. For complete formal definitions cf. [14].

Fig. 1. C0 consistency (left) and C0 weak consistency (right) relations.

3.4 Correctness of User Processes: Context-Encoding Relation

The correctness of user processes is stated by the context-encoding relation $\mathcal{C}$ which maps the vector of user processes up to the physical machine configuration pm . For a process pid and a register r which belong to the context, let $ad_{pid}^r(pm)$ be the physical memory address in the machine pm where the register r of the process pid is stored in the pcb array. The content of this register is then $r_{pid} = pm.m_4(ad_{pid}^r(pm))$. If r refers to a general purpose register i of a process pid we write $gpr[i]_{pid}$. Also we define $gpr[i..j]_{pid} = gpr[i]_{pid} \circ \dots \circ gpr[j]_{pid}$. The address and the content of the cup variable are $ad^{cup}(pm)$ and $cup(pm) = pm.m_4[ad^{cup}(pm)]$, respectively. For an inactive process pid we define its stored context as $stored-ctx(pm, pid) = pc_{pid}(pm) \circ dpc_{pid}(pm) \circ gpr[1..31]_{pid}(pm) \circ sr_{pid}(pm) \circ pto_{pid}(pm) \circ ptl_{pid}(pm)$. In case $sys(pm)$ the kernel is running and all user process are suspended. Otherwise a user with identifier $pid = cup(pm)$ is running and its context occupies physical registers of the hardware. We define the context of a physical machine pm as $active-ctx(pm) = pm.pc \circ pm.dpc \circ pm.gpr[1..31] \circ pm.sr \circ pm.pto \circ pm.ptl$. Altogether, we formalize both stored and active process contexts as:

$$proc-ctx(pm, pid) = \begin{cases} stored-ctx(pm, pid) & \text{if } sys(pm) \vee cup(pm) \neq pid \\ active-ctx(pm) & \text{otherwise} \end{cases}.$$

The context-encoding relation states that all user processes are simultaneously encoded by the hardware:

$$\mathcal{C}(up, pm) = \forall pid : proc-ctx(pm, pid) = ctx(up[pid]).$$

3.5 Combining Correctness Relations

The correctness of user processes, expressed by the context-encoding relation, has to hold at every step of user executions. A kernel run might contain a system call which affects users, e.g., writing to a user register. The abstract semantics, i.e., specification, of system calls is defined over the vector of user processes up . The concrete semantics, i.e., implementation, affects though the underlying physical machine. Abstract and concrete semantics must agree: universally this

Fig. 2. Overall correctness: context-encoding relation holds throughout users and kernel executions, C0 consistency holds during kernel runs, and C0 weak consistency holds during users computations.

is expressed as the the context-encoding relation must be preserved under kernel runs.

The correctness of the kernel, expressed by the C0 consistency relation has to hold after an execution of every statement of the kernel. However, a C0 small-step semantics configuration encoding the kernel and consistent to the underlying hardware cannot be simply constructed from a physical machine configuration after a user execution. There are two reasons for that: (i) there is no information about C0 types in the memory model of the physical machine, and (ii) there is no one-to-one mapping between the heap of the physical machine and the heap memory frame of the C0 configuration. In order to be able to resume a C0 kernel computation after a user execution users have to respect the constraint that they do not affect kernel code and kernel data, including a mapping between heaps. We call this constraint the *C0 weak consistency* and define formally below. Fig. 2 depicts overall correctness.

3.6 Correctness of Suspended Kernels: C0 Weak Consistency

The parts of the C0 configuration k encoding the kernel that is possible to preserve unchanged during user computation are the global memory frame $k.mem.gm$ and the heap memory frame $k.mem.hm$. These two components define a C0 small-step semantics configuration which we call a *weak kernel*. Formally, we construct a weak kernel k_{weak} by means of the function $weak-kernel(gm, hm) = k_{weak}$, s.t. $k_{weak}.mem = (gm, [], hm)$ and $k_{weak}.pr = skip$.

A weak kernel k_{weak} is related to a physical machine configuration pm w.r.t. the kernel program π_k and an allocation function $alloc$ by the C0 weak consistency relation $consis_{weak}(alloc)(\pi_k, k_{weak}, pm)$. The relation comprises: (i) the code consistency $consis_{code}$, which guarantees that users do not modify the kernel code, (ii) the value consistency $consis_v$ and the pointer consistency $consis_p$, which guarantees that users do not modify the kernel data, and (iii) the allocation consistency $consis_{alloc}$ and the kernel heap consistency $consis_{kheap}$, which

defines the mapping between heaps of the physical machine and the C0 configuration. The kernel heap consistency might be seen as a part of the register consistency: exploiting the fact that the variable *kheap* stores the value of the kernel heap pointer, $consis_{kheap}$ states that the value in *kheap* points to the first free address of $k_{weak}.mem.hm$. Fig. 1 compares normal and weak C0 consistency relations.

4 Correctness of Process-Context Switch

In this section we state the correctness theorems of process-context switch. The general idea behind their proofs is to conclude functional correctness, i.e., an input-output relation over configurations of the physical machine running the kernel, by means of inline assembly semantics and to show that the functional correctness result implies all necessary relations. Next, we present our approach to verification of inline assembly portions in C0 code. For more detailed description cf. [19].

4.1 Formal Inline Assembly Semantics

In brief, our approach to deal with verification of assembly statements is as follows. On an assembly statement the C-execution is no longer possible, thus we switch to the consistent underlying physical machine and continue the computations directly there. When all assembly instructions have been executed we switch back to the C0 level. In order to regain C0 consistency we have to reflect effects of assembly instructions on the C0 small-step configuration. Examining the allocation function *alloc* we determine which variables of the C0 configuration have changed. We retrieve their values from the physical machine and write back to the C0 memory configuration.

Let c be the C0 configuration with $c.pr = asm(il);r$, and let pm be the physical machine consistent to c , i.e., $consis(alloc)(\pi, c, pm)$. From the control consistency we have that the delayed program of pm point to the address of the assembly statement: $pm.dpc = ad(asm(il))$, where $ad(s)$ yields for a statement s its address in the memory of pm . This makes it possible to start reasoning about the correctness of the assembly code *il* directly in the semantics of the physical machine. Let pm' be the physical machines configuration after executing *il*. In order to formally specify the effect of an execution of $asm(il)$ on the C0 configuration c we define the function $upd(c, pm, pm') = c'$ which analyzes the difference between pm and pm' and projects it to the C0 level updating the configuration c to c' . A number of restrictions are imposed on the changes in the physical machine, which guarantee that the C0 configuration is not destroyed by the assembly portion *il*, namely: (i) the program pointers after the execution of *il* point to the end of *il*: $pm'.dpc = pm.dpc + 4 \cdot |il|$, (ii) the global, stack, and heap pointers are unchanged: $pm'.(gp, sp, hp) = pm.(gp, sp, hp)$, (iii) the memory region where the compiled code is stored stays the same, i.e., we forbid self-modifying code, (iv) the memory occupied by the local memory frames remains

the same except for top local memory frame, and (v) pointers change is forbidden except setting them to *null*. The inline assembly semantics forces us to prove that assembly portions meet these restrictions.

It is easy to notice, that assembly implementations of context-switch procedures do not meet several restrictions. Both, SAVE and RESTORE procedures rewrite global, stack, and heap pointers, and therefore violate restriction ii. The RESTORE routine has RFE as the last instruction — its semantics rewrites program counters, which violates the first restriction. Fortunately, SAVE is the first function of the kernel and RESTORE is the last. Moreover, there is no C0 state-ments in SAVE before, and in RESTORE after their assembly portions. Thus, we do not construct C0 configurations before the assembly portion of SAVE, and after the assembly portion of RESTORE. As long as there is no complete C0 configurations the stated restrictions are irrelevant. Certainly, we need to have a C0 configurations after the assembly portion of SAVE in order to proceed with a kernel C0 computation. For this we project the semantics of assembly parts on the memory of a weak kernel and construct the desired C0 configuration from it. The details are given in the proof sketch of the correctness theorem for SAVE.

4.2 Correctness Theorems of Process-Context Switch

We start with a simpler case of context restore: the next theorem justifies a switch from a kernel to a user executions.

Theorem 1 (Correctness of Process-Context Restore). Let k be a C0 small-step configuration of the kernel program π_k , up a vector of user processes configurations, and pm a configuration of the underlying physical machine. Assume that (i) the kernel is invoking the RESTORE procedure: $k.pr = call(RESTORE)$, (ii) the kernel is consistent to the physical machine: $consis(alloc)(\pi_k, k, pm)$, and (iii) the context-encoding relation holds: $\mathcal{C}(up, pm)$, then there exists a number of steps T of the physical machine, s.t. $pm' = \delta_{pm}^T(pm)$ after which (i) a user execution starts: $\neg sys(pm')$, (ii) the context-encoding relation is preserved: $\mathcal{C}(up, pm')$, and (iii) there exists a weak kernel configuration k_{weak} , s.t. it is weak-consistent to the physical machine: $consis_{weak}(alloc)(\pi_k, k_{weak}, pm')$.

Proof Sketch. Executing the assembly body of RESTORE in the semantics of the physical machine we obtain the number of steps T . By showing the functional correctness of RESTORE we obtain all necessary arguments to conclude $\neg sys(pm')$ and $\mathcal{C}(up, pm')$. We project the semantics of the assembly portion to global and heap memories $k.mem.gm$ and $k.mem.hm$ obtaining the new values gm' and hm' . We construct the weak kernel $k_{weak} = weak-kernel(gm', hm')$. From the initial C0 consistency relation $consis(alloc)(\pi_k, k, pm)$ it is fairly easy to conclude the desired weak consistency $consis_{weak}(alloc)(\pi_k, k_{weak}, pm')$.

The theorem for context save describes the computation which starts directly after the last user step: the JISR semantics is applied on the physical

machine. However, since the mode register of the physical machine is set to system, the context-encoding relation does not hold at this point. In order to regain it we have to artificially undo the JISR effect. For this we define the function $\delta_{pm}^{-jistr}(pm)$ which returns the updated configuration pm' of the physical machine, s.t. $pm'.(pc, dpc, sr, mode) = pm.(epc, edpc, esr, 1)$.

Theorem 2 (Correctness of Process-Context Save). Let k_{weak} be a weak kernel configuration corresponding to the kernel program π_k , up a vector of user processes configurations, and pm a configuration of the underlying physical machine. Assume that (i) the program counters point to the kernel start: $pm.(dpc, pc) = (kstart, kstart + 4)$, (ii) the suspended kernel is weak-consistent to the physical machine: $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm)$, and (iii) the context-encoding relation holds with the physical machine configuration on which the JISR effect is undone: $\mathcal{C}(up, \delta_{pm}^{-jistr}(pm))$, then there exists a number of steps T of the physical machine, s.t. $pm' = \delta_{pm}^T(pm)$ after which (i) the context-encoding relation is preserved: $\mathcal{C}(up, pm')$, and (ii) there exists a kernel C0 small-step configuration k , s.t. the consistency relation $consis(alloc)(\pi_k, k, pm)$ holds.

Proof Sketch. The code consistency conjunct of $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm)$ ensures that the translated code of SAVE resides in the memory of the physical machine at address $kstart$. By executing the assembly body of SAVE in semantics of the physical machine we get the number of steps T . From the functional correctness of the SAVE code we conclude the $\mathcal{C}$ -relation. We project modifications over $k_{\text{weak}}.mem.gm$ and $k_{\text{weak}}.mem.hm$ done in assembly semantics to the C0 level and obtain the new values gm' and hm' . We specify a single local memory frame lm for the SAVE function and define a program rest pr containing a single statement — a call to the kernel dispatcher. The resulting C0 small-step configuration k is constructed as $k.mem = (gm', lm, hm')$ and $k.pr = pr$. We exploit the initial weak consistency $consis_{\text{weak}}(alloc)(\pi_k, k_{\text{weak}}, pm)$ to show the coincident conjuncts of $consis(alloc)(\pi_k, k, pm')$. The register consistency conjunct follows from the fact that we have loaded kernel values to $pm'.(gp, sp, hp)$. Execution of the assembly body on the physical-machine level sets the program counters to the address of the first C0 statement, thus we conclude the remaining control consistency.

4.3 Considering Additional Invariants

We have presented how CVM implements correct multitasking by means of verified process-context switch. Being a part of the kernel, context-switch procedures must also respect correctness of memory virtualization. Next, we describe additional invariants which we consider in order to guarantee that.

Memory Virtualization Relation. Memories of user processes are stored partly in the memory of the physical machine and partly on the hard disk. In case user process pid accesses address a with the corresponding page present in the physical memory, i.e., valid bit $v(pm, a)$ is on, the kernel provides the translated

physical memory address $pma(a, pid)$. Otherwise, the page resides in the swap memory of the hard disk $hd.sm$ at the swap memory address $sma(a, pid)$. It is subject to move to the main memory by the page-fault handler. We reconstruct the virtual memory of a user process with an identifier pid by means of the function $proc-m(pm, hd, pid)$:

$$proc-m(pm, hd, pid) = \lambda a : \begin{cases} pm.m(pma(a, pid)) & \text{if } v(pm, a) \\ hd.sm(sma(a, pid)) & \text{otherwise} \end{cases}.$$

The memory virtualization relation, called the $\mathcal{B}$ -relation is defined as follows:

$$\mathcal{B}(up, pm, hd) = \forall pid : proc-m(pm, hd, pid) = up[pid].m.$$

Proving $\mathcal{B}$ -relation over Context Switch. The relation for memories might be affected by the assembly portions in SAVE and RESTORE. In order to show formally that this is not the case we prove that assembly code does not store in: (i) physical memory region occupied by virtual memories of user processes, (ii) page tables — this guarantees correctness of address translation, and (iii) hard disk ports — this argues that the data on the hard disk remains unchanged.

5 Summing Up

We have presented the first formal correctness proof of process-context switch for pervasively verified C-programmed kernels. We have stepped beyond existing approaches to reasoning about context switch, and inline assembly systems code in general, only in single semantics of underlying machine model because it is unacceptable for pervasive verification of operating systems. We have unified separate models — therefore, formal theories — for user processes, a kernel written in C with inline assembly, and hardware in correctness theorems for context switch. By that, a solid foundation for a trustworthy multitasking is laid. Our result is successfully integrated into formal theories of CVM, a verified framework for building kernels. Based on CVM — thus, exploiting the presented verified context-switch mechanism — two microkernels have been built, tested, and verified to a large extent in Verisoft: (i) VAMOS [6], an L4-inspired microkernel, and (ii) OLOS [12], an OSEKtime-like operating system, used in a distributed automotive real-time system establishing eCall functionality.

The formal proof of Theorem 1 conducted in Isabelle/HOL and its integration into the CVM model are about 5K steps long. The same task for SAVE, i.e., Theorem 2, is accomplished in 10K steps: the problem of constructing a kernel C0 state after a user execution required much effort, also correctness proof of physical memory modifications performed in SAVE is involved. The above numbers include 1.5K resp. 2K lines of interactive proofs for the functional correctness of RESTORE resp. SAVE assembly bodies. We believe that the latter numbers could come to naught by integrating automatic verification tools like assembly model checkers into Isabelle/HOL.

References

1. E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In *14th Intl Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 109–123. Springer, 2008.
2. W.R. Bevier, W.A. Hunt, Jr., J.S. Moore, and W.D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.
3. S. Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Dept., 2005.
4. S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W. Paul. Putting it all together: Formal verification of the VAMP. *Intl Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, 2006.
5. I. Dalinger, M. Hillebrand, and W. Paul. On the verification of memory management mechanisms. In *13th Adv. Research Working Conference on Correct Hardware Design and Verification Methods*, pages 301–316. Springer, 2005.
6. M. Daum, J. Dörenbacher, B. Wolff, and M. Schmidt. A verification approach for system-level concurrent programs. In *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments*, LNCS. Springer, 2008.
7. G. Heiser et al. Towards trustworthy computing systems: Taking microkernels to the next level. *Operating Systems Review*, 41(4):3–11, 2007.
8. X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008.
9. M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *18th Intl Conference on Theorem Proving in Higher Order Logics*, pages 1–16. Springer, 2005.
10. T. Hoare, J. Misra, G.T. Leavens, and Natarajan S. The verified software initiative: A manifesto. <http://qpq.csl.sri.com/vsr/manifesto.pdf>, 2007.
11. G.C. Hunt and J.R. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.
12. T. In der Rieden and S. Knapp. An approach to the pervasive formal specification and verification of an automotive system. In *10th Intl Workshop on Formal Methods for Industrial Critical Systems*, pages 115–124. IEEE, 2005.
13. T. In der Rieden and A. Tsyban. CVM — a verified framework for microkernel programmers. In *3rd Intl Workshop on Systems Software Verification*. Elsevier Science B.V., 2008.
14. D. Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Dept., 2008.
15. D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd Intl Workshop on Systems Software Verification*. Elsevier Science B. V., 2008.
16. J.S. Moore. A grand challenge proposal for formal methods: A verified stack. In *10th Anniversary Colloquium of UNU/IIST*, pages 161–172. Springer, 2003.
17. S.M. Müller and W.J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.
18. Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In *20th Intl Conference on Theorem Proving in Higher Order Logics*, LNCS, pages 189–206. Springer, 2007.
19. A. Starostin and A. Tsyban. Correct microkernel primitives. In *3rd Intl Workshop on Systems Software Verification*. Elsevier Science B. V., 2008.
20. The Verisoft Consortium. The Verisoft Project. <http://www.verisoft.de>, 2003.