# On the Correctness of Operating System Kernels

Mauro Gargano*, Mark Hillebrand*, Dirk Leinenbach*,**, and Wolfgang Paul

Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany
{gargano, mah, dirkl, wjp}@wjpserver.cs.uni-sb.de

**Abstract.** The Verisoft project aims at the pervasive formal verification of entire computer systems. In particular, the seamless verification of the *academic system* is attempted. This system consists of hardware (processor and devices) on top of which runs a microkernel, an operating system, and applications. In this paper we define the computation model CVM (communicating virtual machines) in which concurrent user processes interact with a generic microkernel written in C. We outline the correctness proof for concrete kernels, which implement this model. This result represents a crucial step towards the verification of a kernel, e.g. that in the academic system. We report on the current status of the formal verification.

## 1 Introduction

There is no need to argue about the importance of computer security [1] and operating system security is in the center of computer security. Making operating systems comfortable and at the same time utmost reliable is extremely hard. However, some small and highly reliable operating system kernels, e.g. [2,3,4], have been developed. A reliable kernel opens the way to uncouple the safety-critical applications running under an operating system from the non-critical ones. One runs *two* operating systems under a trusted kernel, a small trusted one for the safety-critical applications and a conventional one for all others. This minimizes the total size of the trusted components. For example, [5] describes a small operating system and Linux running under the L4 microkernel [6].

For critical applications one wishes of course to estimate, how much trust one should put into a system.For this purpose the *common criteria* for information technology security evaluation [7] define a hierarchy of *evaluation assurance levels* EAL-1 to EAL-7. These are disciplines for reviewing, testing / verifying, and documenting systems during and after development. Even the highest assurance level, EAL-7, does not require formal verification of the system implementation. Clearly, the common criteria, in the current revision, stay *behind* the state of the art available at that time: already nine years before Bevier [8] reported on the full formal verification of KIT, a small multitasking operating system kernel written in machine language. KIT implements a fixed number of processes, each occupying a fixed portion of the processor's memory. It provides the following verified services: process scheduling, error handling, message passing, and an interface to asynchronous devices. In terms of complexity, KIT is near to small real-time operating systems like e.g. OSEKTime [9].

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

** Work supported by DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

J. Hurd and T.F. Melham (Eds.): HM 2005, LNCS 3603, pp. 1–16, 2005.

In this paper we outline an approach to the pervasive verification of a considerably more powerful kernel, supporting virtual memory, memory management, system calls, user defined interrupts, etc. We outline substantial parts of its correctness proof. We report on the current status of the formal verification. The results presented in this paper were obtained in and are of crucial importance to the Verisoft project [10], funded by the German Federal Government. Verisoft has the mission to provide the technology for the formal pervasive verification of entire computer systems of industrial complexity.

## 2 Overview

To handle the design complexity, computer systems are organized in layers some of which are modeled by well established formal models. Examples are (i) the hardware layer that is modeled by switching circuits and memory components, (ii) the machine language layer that is modeled by random access machines [11] with an appropriate instruction set, and (iii) the programming language layer, e.g. for C, is, for operational semantics, modeled by abstract interpreters, also called abstract C machines. Correctness theorems for components of computer systems are often simulation theorems between *adjacent* layers. Processor correctness concerns a simulation between Layers (i) and (ii). Compiler correctness concerns a simulation between Layers (ii) and (iii).

Aiming at formulating and proving a correctness theorem for an operating system kernel we take a similar approach. We introduce an abstract parallel model of computation called *communicating virtual machines* (CVM) that formalizes concurrent user processes interacting with an operating system kernel. In this model user processes are virtual machines, i.e. processors with virtual memory. The so-called *abstract kernel* is represented as an abstract C machine. Beyond the usual C functions the abstract kernel can call a few special functions, called the *CVM primitives*, that alter the configuration of user processes. For instance, there are CVM primitives to increase / decrease the memory size of a user process or to copy data between user processes (and I/O devices).

By linking abstract kernels with a program implementing the CVM functionality we obtain the *concrete kernel*. In particular, the concrete kernel contains the implementation of the CVM primitives and the implementation of handlers for page faults (not visible in the abstract model). A crucial observation is that the concrete kernel *necessarily* contains assembler code because neither processor registers nor user processes are visible in the variables of a C program. Thus the correctness theorem for the concrete kernel will establish a simulation between CVM and Layer (ii) instead of Layer (iii). Since reasoning on assembler level is tedious we minimize its use in the concrete kernel.

The remainder of this paper is structured as follows. In Sect. 3 we define virtual machines and summarize results from [12] on the simulation of virtual machines by physical machines, processors with physical and swap memory. In Sect. 4 we define abstract $C0$ machines and summarize the compiler correctness proof from [13]. In Sect. 5 we define the CVM model using virtual machines to model computation of the user and abstract $C0$ machines to model computation of an abstract kernel. Section 6 sketches the construction of the concrete kernel containing the CVM implementation. We state the correctness proof for the concrete kernel and outline its proof. In Sect. 7 we report on the status of the formal verification. In Sect. 8 we conclude and sketch further work.

# 3 Virtual Memory Simulation

Let us introduce some notation. We denote bitvectors by $a \in \{0,1\}^n$. Bit $j$ of bitvector $a$ is denoted by $a[j]$, the sub bitvector consisting of bits $j$ to $k$ (with $k < j$) is denoted by $a[j\!:\!k]$. The concatenation of two bitvectors $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ is denoted by $a \circ b \in \{0,1\}^{n+m}$. Occasionally we will abuse notation and identify bitvectors $a$ with their value $\langle a \rangle = \sum_i a[i] \cdot 2^i$ and vice versa. Arithmetic is modulo $2^n$. We model memories $m$ as mappings from addresses $a \in \{0,1\}^{32}$ to byte values $m(a) \in \{0,1\}^8$. For natural numbers $d$ we denote by $m_d(a)$ the content of $d$ consecutive memory cells starting at address $a$, so $m_d(a) = m(a+d-1) \circ \cdots \circ m(a)$.

In the following sub sections we summarize results from [12].

## 3.1 Virtual Machines

Virtual machines consist of a processor operating on a (uniform) virtual memory. Configurations $c_{\mathrm{V}}$ of virtual machines have the following components:

- $c_{\mathrm{V}}.R \in \{0,1\}^{32}$ for a variety of processor registers $R$. We consider here pipelined DLX machines [14] with a delayed branch mechanism that is implemented by *two* program counters, called delayed program counter $c_{\mathrm{V}}.DPC \in \{0,1\}^{32}$ and program counter $c_{\mathrm{V}}.PC \in \{0,1\}^{32}$. For details see [15].
- The size $c_{\mathrm{V}}.V$ of the virtual memory measured in pages of 4K bytes. It defines the set of accessible virtual addresses $VA(c_{\mathrm{V}}) = \{a \in \{0,1\}^{32} \mid a < c_{\mathrm{V}}.V \cdot 4\mathrm{K}\}$. We split virtual addresses $va = va[31\!:\!0]$ into page index $va.px = va[31\!:\!12]$ and byte index $va.bx = va[11\!:\!0]$.
- A byte addressable virtual memory $c_{\mathrm{V}}.vm : VA(c_{\mathrm{V}}) \to \{0,1\}^8$.
- A write protection function $c_{\mathrm{V}}.p : VA(c_{\mathrm{V}}) \to \{0,1\}$ that only depends on the page index of virtual addresses. A virtual address $va$ is write protected if $c_{\mathrm{V}}.p(va) = 1$.

Computation of the virtual machine is modeled by the function $\delta_{\mathrm{V}}$ that computes for a given configuration $c_{\mathrm{V}}$ its successor configuration $c'_{\mathrm{V}}$. The virtual machine accesses the memory in the following situations: it reads the memory to fetch instructions and to execute load instructions, it writes the memory to execute store instructions.

However, any access to a virtual address $va \notin VA(c_{\mathrm{V}})$ or a write access to $va$ with $c_{\mathrm{V}}.p(va) = 1$ is illegal and leads to an exception. For the CVM model (cf. Sect. 5) we do not consider write protected pages and assume $c_{\mathrm{V}}.p(va) = 0$ for all $va \in VA(c_{\mathrm{V}})$.

Note that the effects of exceptions are not defined in a virtual machine model alone but in an extended context of a virtual machine running under a certain operating system (kernel). Also, the size of the virtual memory $c_{\mathrm{V}}.V$ cannot be changed by the virtual machine itself. This is described in more detail in Sect. 5.

## 3.2 Physical Machines and Address Translation

Physical machines consist of a processor operating on physical memory and swap memory. Configurations $c_{\mathrm{P}}$ of physical machines have components $c_{\mathrm{P}}.R$ for processor registers $R$, $c_{\mathrm{P}}.pm$ for the physical memory, and $c_{\mathrm{P}}.sm$ for the swap memory. The physical machine has several special purpose registers not present in virtual machines, e.g. the

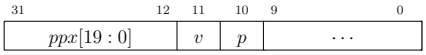


**Fig. 1.** Page Table Entry

mode register $mode$, the page table origin $pto$, and the page table length $ptl$. Computation of the physical machine is modeled by the next state function $\delta_{\mathrm{P}}$.

In system mode, i.e. if $c_{\mathrm{P}}.mode = 0$, the physical machine operates almost like a virtual machine with extra registers. In user mode, i.e. $c_{\mathrm{P}}.mode = 1$, memory accesses are subject to address translation: they either cause a page fault or are redirected to the translated physical memory address $pma(c_{\mathrm{P}}, va)$. The result of address translation depends on the contents of the *page table*, a region of the physical memory starting at address $c_{\mathrm{P}}.pto \cdot 4\mathrm{K}$ with $(c_{\mathrm{P}}.ptl + 1)$ entries of four bytes width.

The page table entry address for virtual address $va$ is defined as $ptea(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pto \cdot 4\mathrm{K} + 4 \cdot va.px$ and the page table entry of $va$ is defined as $pte(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pm_4(ptea(c_{\mathrm{P}}, va))$. As shown in Fig. 1, a page table entry consists of three components, the physical page index $ppx(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[31 : 12]$, the valid bit $v(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[11]$, and the write protection bit $p(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[10]$.

On user mode memory access to address $va$, a page fault is signaling if the page index exceeds the page table length, $va.px > c_{\mathrm{P}}.ptl$, if the page table entry is not valid, $v(c_{\mathrm{P}}, va) = 0$, or if for a write access the write protection is active, $p(c_{\mathrm{P}}, va) = 1$. On page fault the page fault handler, an interrupt service, is invoked.

Without a page fault, the access is performed on the (translated) physical memory address $pma(c_{\mathrm{P}}, va)$ defined as the concatenation of the physical page index and the byte index, $pma(c_{\mathrm{P}}, va) = ppx(c_{\mathrm{P}}, va) \circ va.bx$.

For example, the instruction $I(c_{\mathrm{P}})$ fetched in configuration $c_{\mathrm{P}}$ is defined as follows. If $c_{\mathrm{P}}.mode = 0$ we define $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(c_{\mathrm{P}}.DPC)$, otherwise, provided that there is no page fault, we define $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(pma(c_{\mathrm{P}}, c_{\mathrm{P}}.DPC))$.

### 3.3 Virtual Memory Simulation

A physical machine with appropriate page fault handlers can simulate virtual machines. For a simple page fault handler, virtual memory is stored on the swap memory of the physical machine and the physical memory acts as a write back cache. In addition to the architecturally defined physical memory address $pma(c_{\mathrm{P}}, va)$, the page fault handler maintains a swap memory address function $sma(c_{\mathrm{P}}, va)$.

We use a simulation relation $B(c_{\mathrm{V}}, c_{\mathrm{P}})$ to indicate that a (user mode) physical machine configuration $c_{\mathrm{P}}$ encodes virtual machine configuration $c_{\mathrm{V}}$. Essentially, $B(c_{\mathrm{V}},c_{\mathrm{P}})$ is the conjunction of the following three conditions:

- For every page of virtual memory there is a page table entry in the physical machine, $c_{\mathrm{V}}.V = c_{\mathrm{P}}.ptl + 1$.
- The write protection function of the virtual machine is encoded in the page table, $c_{\mathrm{V}}.p(va) = p(c_{\mathrm{P}}, va)$. As noted earlier in this paper we assume $p(c_{\mathrm{P}}, va) = c_{\mathrm{V}}.p(va) = 0$.
- The virtual memory is stored in physical and swap memory: if $v(c_{\mathrm{P}}, va)$ then $c_{\mathrm{V}}.vm(va) = c_{\mathrm{P}}.pm(pma(c_{\mathrm{P}}, va))$, else $c_{\mathrm{V}}.vm(va) = c_{\mathrm{P}}.sm(sma(c_{\mathrm{P}}, va))$.

The simulation theorem for a single virtual machine has the following form:

**Theorem 1.** *For all computations* $(c_{\mathrm{V}}^0, c_{\mathrm{V}}^1, \dots)$ *of the virtual machine there is a computation* $(c_{\mathrm{P}}^0, c_{\mathrm{P}}^1, \dots)$ *of the physical machine and there are step numbers* $(s(0), s(1), \dots)$ *such that for all* $i$ *and* $S = s(i)$ *we have* $B(c_{\mathrm{V}}^i, c_{\mathrm{P}}^S)$.

Thus step $i$ of the virtual machine is simulated after step $s(i)$ of the physical machine. Even for a simple handlers, the proof is not completely obvious since a single user mode instruction can cause two page faults. To avoid deadlock and guarantee forward progress, the page fault handler must not swap out the page that was swapped in during the last execution of the page fault handler.

### 3.4 Synchronization Conditions

If the hardware implementation of a physical machine is pipelined, then an instruction $I(c_{\mathrm{P}}^i)$ that is in the memory stage may modify / affect a later instruction $I(c_{\mathrm{P}}^j)$ for $j > i$ after it has been fetched. It may (i) overwrite the instruction itself, (ii) overwrite its page table entry, or (iii) change the mode. In such situations instruction fetch (in particular translated fetch implemented by a memory management unit) would not work correctly. Of course it is possible to detect such data dependencies in hardware and to roll back the computation if necessary. Alternatively, the software to be run on the processor must adhere to certain *software synchronization conventions*. Let $iaddr(c_{\mathrm{P}}^j)$ denote the address of instruction $I(c_{\mathrm{P}}^j)$, possibly translated. If $I(c_{\mathrm{P}}^i)$ writes to address $iaddr(c_{\mathrm{P}}^j)$, then an intermediate instruction $I(c_{\mathrm{P}}^k)$ for $i < k < j$ must drain the pipe. The same must hold if $c_{\mathrm{P}}^j$ is in user mode and $I(c_{\mathrm{P}}^i)$ writes to $ptea(c_{\mathrm{P}}^j, c_{\mathrm{P}}^j.DPC)$. Finally, mode can only be changed to user mode by an `rfe` (return from exception) instruction (and the hardware guarantees that `rfe` instructions drain the pipe).

Conditions of this nature are hypotheses of the hardware correctness proof in [12]. It will be easy to show that they hold for the kernels constructed in Sect. 6.

## 4 Compilation

We sketch the formal semantics of $C0$, a subset of $C$, and state the correctness theorem of a $C0$ compiler, summarizing result from [13]. In Section 4.3 we extend the $C0$ semantics to inline assembler code.

### 4.1 $C0$ Semantics

Eventually we want to consider several programs running under an operating system. The computations of these programs then are interleaved. Therefore our compiler correctness statement is based on a small steps / structured operational semantics [16,17].

In $C0$ types are elementary ($bool$, $int$, ...), pointer types, or composite ($array$ or $struct$). A type is called simple if it is an elementary type or a pointer type. We define the (abstract) size of types for simple types $t$ by $size(t) = 1$, for arrays by $size(t[n]) = n \cdot size(t)$, and for structures by $size(struct\{n_1 : t_1, \dots, n_s : t_s\}) = \sum_i size(t_i)$. Values of variables with simple type are called *simple values*. Variables with composite types have *composite values* that are represented flat as a sequence of simple values.

*Configuration.* An $C0$ machine configuration $c_{\mathrm{C0}}$ has the following components:

1. The *program rest* $c_{\mathrm{C0}}.pr$. This is a sequence of $C0$ statements which still needs to be executed. In [16] the program rest is called *code component* of the configuration.
2. The *type table* $c_{\mathrm{C0}}.tt$ collects information about types used in the program.
3. The *function table* $c_{\mathrm{C0}}.ft$ contains information about the functions of a program. It maps function names $f$ to pairs $c_{\mathrm{C0}}.ft(f) = (c_{\mathrm{C0}}.ft(f).ty, c_{\mathrm{C0}}.ft(f).body)$ where $c_{\mathrm{C0}}.ft(f).ty$ specifies the types of the arguments, the local variables, and the result of the function, whereas $c_{\mathrm{C0}}.ft(f).body$ specifies the function body.
4. The *recursion depth* $c_{\mathrm{C0}}.rd$.
5. The *local memory stack* $c_{\mathrm{C0}}.lms$. It maps numbers $i \leq c_{\mathrm{C0}}.rd$ to memory frames (defined below). The global memory is $c_{\mathrm{C0}}.lms(0)$. We denote the top local memory frame of a configuration $c_{\mathrm{C0}}$ by $top(c_{\mathrm{C0}}) = c_{\mathrm{C0}}.lms(c_{\mathrm{C0}}.rd)$.
6. A *heap memory* $c_{\mathrm{C0}}.hm$. This is also a memory frame.

*Memory Frames.* We use a relatively explicit, low level memory model in the style of [18]. Memory frames $m$ have the following components: (i) the number $m.n$ of variables in $m$ (for local memory frames this also includes the parameters of the corresponding function definition), (ii) a function $m.name$ mapping variable numbers $i \in [0 : m.n - 1]$ to their names (not used for variables on the heap), (iii) a function $m.ty$ mapping variable numbers to their type. This permits to define the size of a memory frame $size(m)$ as the number of simple values stored in it, namely: $size(m) = \sum_{i=0}^{m.n-1} size(m.ty(i))$. (iv) a content function $m.ct$ mapping indices $0 \leq i < size(m)$ to simple values.

A *variable* of configuration $c_{\mathrm{C0}}$ is a pair $v = (m, i)$ where $m$ is a memory frame of $c_{\mathrm{C0}}$ and $i < m.n$ is the number of the variable in the frame. The type of a variable $(m, i)$ is defined by $ty((m, i)) = m.ty(i)$.

Sub variables $S = (m, i)s$ are formed from variables $(m, i)$ by appending a *selector* $s = (s_1, \ldots, s_t)$, where each component of a selector has the form $s_i = [j]$ for selecting array element number $j$ or the form $s_i = .n$ for selecting the struct component with name $n$. If the selector $s$ is consistent with the type of $(m, i)$, then $S = (m, i)s$ is a *sub variable* of $(m, i)$. Selectors are allowed to be empty. In $C0$, pointers $p$ may point to sub variables $(m, i)s$ in the global memory or on the heap. The value of such pointers simply has the form $(m, i)s$. Component $m.ct$ stores the current values $va(c_{\mathrm{C0}}, (m, i)s)$ of the simple sub variables $(m, i)s$ in the canonical order. Values of composite variables $x$ are represented in $m.ct$ in the obvious way by sequences of simple values starting from the abstract base address $ba(x)$ of variable $x$.

With the help of visibility rules and bindings we easily extend the definition of $va$, $ty$, and $ba$ from variables and sub variables to expressions $e$.

*Computation.* For space restrictions we cannot give the definitions of the (small-step) transition function $\delta_{\mathrm{C0}}$ mapping $C0$ configurations $c_{\mathrm{C0}}$ to their successor configuration $c'_{\mathrm{C0}} = \delta_{\mathrm{C0}}(c_{\mathrm{C0}})$. As an example we give a partial definition of the function call semantics.

Assume the program rest in configuration $c_{\mathrm{C0}}$ begins with a call of function $f$ with parameters $e_1, \ldots, e_n$ assigning the function's result to variable $v$, formally $c_{\mathrm{C0}}.pr =$

$fcall(f, v, e_1, \ldots, e_n); r$. In the new program rest, the call statement is replaced by the body of function $f$ taken from the function table, $c'_{\mathrm{C0}}.pr = c_{\mathrm{C0}}.ft(f).body; r$ and the recursion depth is incremented $c'_{\mathrm{C0}}.rd = c_{\mathrm{C0}}.rd+1$. Furthermore, the values of all parameters $e_i$ are stored in the new top local memory frame by updating its content function at the corresponding positions: $top(c'_{\mathrm{C0}}).ct_{size(ty(c_{\mathrm{C0}},e_i))}(ba(c_{\mathrm{C0}}, e_i)) = va(c_{\mathrm{C0}}, e_i)$.

### 4.2 Compiler Correctness

The compiler correctness statement (with respect to physical machines) depends on a simulation relation $consis(aba)(c_{\mathrm{C0}}, c_{\mathrm{P}})$ between configurations $c_{\mathrm{C0}}$ of $C0$ machines and configurations $c_{\mathrm{P}}$ of physical machines which run the compiled program. The relation is parameterized by a function $aba$ which maps sub variables $S$ of the $C0$ machine to their allocated base addresses $aba(c_{\mathrm{C0}}, S)$ in the physical machine. The allocation function may change during a computation (i) if the recursion depth and thus the set of local variables change due to calls and returns or (ii) if reachable variables are moved on the heap during garbage collection (not yet implemented).

*Simulation Relation.* The simulation relation consists essentially of four conditions:

1. Value consistency $v\text{-}consis(aba)(c_{\mathrm{C0}}, c_{\mathrm{P}})$: this condition states, that reachable elementary sub variables $x$ have the same value in the $C0$ machine and in the physical machine. Let $asize(x)$ be the number of bytes needed to store a value of type $ty(x)$. Then we require $c_{\mathrm{P}}.pm_{asize(x)}(aba(c_{\mathrm{C0}}, x)) = va(c_{\mathrm{C0}}, x)$.
2. Pointer consistency $p\text{-}consis(aba)(c_{\mathrm{C0}}, c_{\mathrm{P}})$: This predicate requires for reachable pointer variables $p$ which point to a sub variable $y$ that the value stored at the allocated address of variable $p$ in the physical machine is the allocated base address of $y$, i.e. $c_{\mathrm{P}}.pm_4(aba(c_{\mathrm{C0}}, p)) = aba(c_{\mathrm{C0}}, y)$. This induces a sub graph isomorphism between the reachable portions of the heaps of the $C0$ and the physical machine.
3. Control consistency $c\text{-}consis(c_{\mathrm{C0}}, c_{\mathrm{P}})$: This condition states that the delayed PC of the physical machine (used to fetch instructions) points to the start of the translated code of the program rest $c_{\mathrm{C0}}.pr$ of the $C0$ machine. We denote by $caddr(s)$ the address of the first assembler instruction which is generated for statement $s$. We require $c_{\mathrm{P}}.DPC = caddr(c_{\mathrm{C0}}.pr)$ and $c_{\mathrm{P}}.PC = c_{\mathrm{P}}.DPC + 4$.
4. Code consistency $code\text{-}consis(c_{\mathrm{C0}}, c_{\mathrm{P}})$: This condition requires that the compiled code of the $C0$ program is stored in the physical machine $c_{\mathrm{P}}$ beginning at the code start address $cstart$. Thus it requires that the compiled code is not changed during the computation of the physical machine and thereby forbids self modifying code.

**Theorem 2.** *For every C0 machine computation* $(c^0_{\mathrm{C0}}, c^1_{\mathrm{C0}}, \ldots)$ *there are a computation* $(c^0_{\mathrm{P}}, c^1_{\mathrm{P}}, \ldots)$ *of the physical machine, step numbers* $(s(0), s(1), \ldots)$*, and a sequence of allocation functions* $(aba^0, aba^1, \ldots)$ *such that for all steps* $i$ *and* $S = s(i)$ *we have* $consis(aba^i)(c^i_{\mathrm{C0}}, c^S_{\mathrm{P}})$.

### 4.3 Inline Assembler Code Semantics

For sequences $u$ of assembler instructions (we do not distinguish here between assembler and machine language) we extend $C0$ by statements of the form $asm(u)$ and call

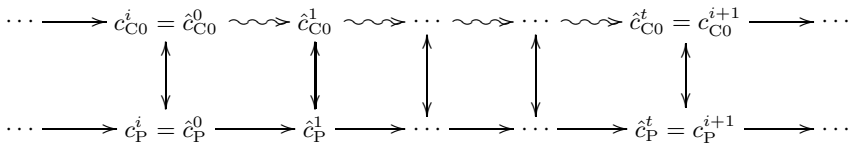


**Fig. 2.** Execution of Inline Assembler Code

the resulting language $C0_A$. In $C0_A$ the use of inline assembler code is restricted: (i) only a certain subset of DLX instructions is allowed (e.g. no load or store of bytes or half words, only *relative* jumps), (ii) the target address of store word instructions must be outside the code and data regions of the $C0_A$ program or it must be equal to the allocated base address of a sub variable of the $C0_A$ program with type $int$ or $unsigned\ int$ (this implies that inline assembler code cannot change the stack layout of the $C0_A$ program), (iii) the last assembler instruction in $u$ must not be a jump or branch instruction, (iv) the execution of $u$ must terminate, (v) the target of jump and branch instructions must not be outside the code of $u$, and (vi) the execution of $u$ must not generate misalignment or illegal instruction interrupts.

As pointed out in Sect. 2, operating system kernels necessarily contain assembler code; thus a formal semantics of programs in $C0_A$ has to be defined. Inline assembler portions of $C0_A$ programs can modify parts of the machine which are not visible to $C0$, e.g. the processor registers or memory which is not reachable via $C0$ variables. Thus to define the meaning of inline assembler code the transition function $\delta_{\mathrm{C0A}}$ of $C0_A$ machines needs, in addition to a $C0$ configuration $c_{\mathrm{C0}}$, a physical machine configuration as a second input parameter. Also the result of $\delta_{\mathrm{C0A}}$ consists of a $C0$ configuration *and* a physical machine configuration. To express the meaning of inline assembler code which changes memory cells holding $C0$ variables we parameterize the $C0_A$ transition function over an allocated base address function $aba$ like we did for the $consis$ relation.

As long as no inline assembler code is executed, we set $\delta_{\mathrm{C0A}}(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}) = (\delta_{\mathrm{C0}}(c^i_{\mathrm{C0}}), x)$ ignoring the second input parameter and setting the second output parameter to an arbitrary, fixed physical machine configuration $x$.

However, when executing inline assembler code, $c^i_{\mathrm{C0}}.pr = asm(u); r$, the definition of $\delta_{\mathrm{C0A}}(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}) = (c^{i+1}_{\mathrm{C0}}, c^{i+1}_{\mathrm{P}})$ is more difficult. We take $c_{\mathrm{P}}$ as the start configuration for the execution of assembler code sequence $u$. The execution of $u$ leads to a physical machine computation $(\hat{c}^0_{\mathrm{P}}, \ldots, \hat{c}^t_{\mathrm{P}})$ with $\hat{c}^t_{\mathrm{P}}.DPC = caddr(r)$ and $\hat{c}^t_{\mathrm{P}}.PC = \hat{c}^t_{\mathrm{P}}.DPC + 4$ by the restrictions on inline assembler. We construct a corresponding sequence $(\hat{c}^0_{\mathrm{C0}}, \ldots, \hat{c}^t_{\mathrm{C0}})$ of intermediate $C0$ machine configurations reflecting successively the possible updates of the C0 variables by the assembler instructions (see Fig. 2). We set $\hat{c}^0_{\mathrm{C0}} = c^i_{\mathrm{C0}}$ except for the program rest: $\hat{c}^0_{\mathrm{C0}}.pr = r$. If the instruction $I(\hat{c}^j_{\mathrm{P}})$ executed in configuration $\hat{c}^j_{\mathrm{P}}$ for $j < t$ writes the value $v$ to the word at an address $ea(\hat{c}^j_{\mathrm{P}})$ equaling the allocated base address of some $C0$ variable $x$, we update the corresponding variable in the $\hat{c}^{j+1}_{\mathrm{C0}}$ by $va(\hat{c}^{j+1}_{\mathrm{C0}}, x) = v$. Finally the result of the $C0_A$ transition function is defined by $c^{i+1}_{\mathrm{C0}} = \hat{c}^t_{\mathrm{C0}}$ and $c^{i+1}_{\mathrm{P}} = \hat{c}^t_{\mathrm{P}}$.

Observe that for the definition from above we do not need to know $c^i_{\mathrm{P}}$ exactly. Nevertheless the definition keeps configurations consistent:

**Lemma 1.** *If the program rest of $c^i_{\mathrm{C0}}$ starts with an inline assembler statement we have* $consis(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}) \implies consis(aba)(\delta_{\mathrm{C0A}}(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}))$.

## 5 CVM Semantics

We introduce communicating virtual machines (CVM), a model of computation for a generic abstract operating system kernel interacting with a fixed number of user processes. CVM uses the $C0$ language semantics to model computation of the (abstract) kernel and virtual machines to model computation of the user processes. It is a pseudo-parallel model in the sense that in every step of computation either the kernel or one user process can make progress.

From a kernel implementor's point of view, CVM encapsulates the low-level functionality of a microkernel and provides access to it as a library of functions, the so-called CVM primitives. Accordingly, the abstract kernel may be 'linked' against the implementation of these primitives to produce the concrete kernel, a $C0_A$ program, that may be run on the target machine. This construction and its correctness will be treated in Sect. 6. In the following sections we define CVM configurations, CVM computations, and show how abstract kernels implement system calls as regular $C0$ function calls.

### 5.1 Configurations

A CVM configuration $c_{\mathrm{CVM}}$ has the following components:

- User process virtual machine configurations $c_{\mathrm{CVM}}.up(u)$ for user process indices $u \in \{1, \ldots, P\}$ (and $P$ fixed, e.g. $P = 128$).
- A $C0$ machine configuration $c_{\mathrm{CVM}}.ca$ of the so-called *abstract kernel*. As we will see below, the kernel configuration, in particular its initial configuration, must have a certain form: (i) it must have a global variable named $i$ of type $int$, (ii) it declares certain functions $f \in CVMP$, the CVM primitives, with empty body, arguments, and effects as described below, and (iii) it must have a function $kdispatch$ that takes two integer arguments and returns an integer; when starting with a call to $kdispatch$ as initial program rest the kernel must eventually call $start$, a CVM primitive that passes control to one of the user processes.
- The component $c_{\mathrm{CVM}}.cp$ denotes the current process: $c_{\mathrm{CVM}}.cp = 0$ means that the kernel is running; $c_{\mathrm{CVM}}.cp = u > 0$ means that user process $u$ is running.

### 5.2 Computation

A computation of the CVM machine is parameterized over a list of external interrupt events $eevs$, one event mask $eev^e$ with $e$ signals for each user process step (the kernel runs uninterruptibly).

In this section we define the next state function $\delta_{\mathrm{CVM}}$ of the CVM model. It maps the external events' list $eevs$ and a CVM configuration $c_{\mathrm{CVM}}$ to its successor configurations $c'_{\mathrm{CVM}}$ and the new external events' list $eevs'$, so $\delta_{\mathrm{CVM}}(eevs, c_{\mathrm{CVM}}) = (c'_{\mathrm{CVM}}, eevs')$. In the definitions below we only list components that are changed.

*User Computation.* If the current process $c_{\mathrm{CVM}}.cp$ in configuration $c_{\mathrm{CVM}}$ is non-zero then user process $u = c_{\mathrm{CVM}}.cp$ is meant to make a step. Let $eevs = eev; eevs'$, i.e. $eev \in \{0,1\}^e$ denotes the first element of the external events' list and $eevs'$ its remainder, the next external events' list.

Let the predicate $JISR(c_{\mathrm{V}}, eev)$ denote that an interrupt occurred in configuration $c_{\mathrm{V}}$, either internally or with respect to the events $eev$. If $JISR(c_{\mathrm{V}}, eev)$, then the (masked) exception cause is encoded in the bitvector $mca(c_{\mathrm{V}}, eev)$ and an additional 'parameter' of the exception (for internal exceptions only) is denoted by $edata(c_{\mathrm{V}})$. For details on the definition of $JISR$, $mca$, and $edata$ see e.g. [12,15].

For $\neg JISR(c_{\mathrm{CVM}}.up(u), eev)$ a CVM step simply consists of a step of the virtual machine $c_{\mathrm{CVM}}.up(u)$, so $c'_{\mathrm{CVM}}.up(u) = \delta_{\mathrm{V}}(c_{\mathrm{CVM}}.up(u))$. Otherwise, execution of the abstract kernel starts. The kernel's entry point is the function $kdispatch$ that is called with the exception masked cause and the exception data. We set the current process component and the kernel's recursion depth to zero, $c'_{\mathrm{CVM}}.cp = 0$ and $c'_{\mathrm{CVM}}.ca.rd = 0$, and the kernel's program rest to the function call $c'_{\mathrm{CVM}}.ca.pr = fcall(kdispatch, i, mca(c_{\mathrm{V}}, eev), edata(c_{\mathrm{V}}))$.

*Kernel Computation.* Initially (after power-up) and after an interrupt, as seen above, the kernel starts execution with a call of the function $kdispatch$. User process execution continues when the kernel calls the $start$ CVM primitive.

If we have $c_{\mathrm{CVM}}.cp = 0$ and the kernel's program rest does not start with a call to a CVM primitive, a regular $C0$ semantics step is performed, $c'_{\mathrm{CVM}}.ca = \delta_{\mathrm{C0}}(c_{\mathrm{CVM}}.ca)$.

Otherwise, we have $c_{\mathrm{CVM}}.cp = 0$ and $c_{\mathrm{CVM}}.cp.pr = fcall(f, v, e_1, \ldots, e_n); r$ for a CVM primitive $f$, an integer variable $v$ and integer expressions $e_1$ to $e_n$. The CVM primitive $f = start$ to start user processes is defined below. For $f \neq start$, the CVM primitive $f$ is specified by a function $f_{\mathrm{S}}$ that takes $n$ integer arguments, a $P$-tuple of virtual machines and returns an integer and an updated $P$-tuple of virtual machines. The new CVM configuration after calling such a primitive is defined as follows. First, we compute the application of $f_{\mathrm{S}}$ to the values $E_i = va(c_{\mathrm{CVM}}.ca, e_i)$ of the expressions $e_i$ and set $(v_{\mathrm{S}}, c'_{\mathrm{CVM}}.up) = f_{\mathrm{S}}(E_1, \ldots, E_n, c_{\mathrm{CVM}}.up)$. Then, the program rest of the kernel is set to $c'_{\mathrm{CVM}}.pr = r$ and the return value $v_{\mathrm{S}}$ is stored in the return variable $v$ of the call to the CVM primitive.

Below we describe the special CVM primitive $start$ and then a few selected other primitives. For lack of space, we ignore any pre conditions or corner cases; these are straightforward to specify and resolve.

- The CVM primitive $start$, taking one argument, hands control over to the specified user process. For $c_{\mathrm{CVM}}.ca.pr = fcall(start, v, e_1); r$ and $u = va(c_{\mathrm{CVM}}.ca, e_1)$ we set $c'_{\mathrm{CVM}}.cp = u$. By this definition, the kernel stops execution and is restarted again on the next interrupt (with a fresh program rest as described before).
- The CVM primitive $alloc$ increases the memory size of user process $u$ by $x$ pages. We define $alloc_{\mathrm{S}}(u, x, c_{\mathrm{CVM}}.up) = (0, c'_{\mathrm{CVM}}.up)$ by increasing the memory size, $c'_{\mathrm{CVM}}.up(u).V = c_{\mathrm{CVM}}.up(u).V + x$, and afterwards clearing the new pages, $c'_{\mathrm{CVM}}.up(u).vm(y) = 0^8$ for $c_{\mathrm{CVM}}.up(u).V \cdot 4\mathrm{K} \leq y < c'_{\mathrm{CVM}}.up(u).V \cdot 4\mathrm{K}$.
- Likewise, the CVM primitive *free* with specification function $free_{\mathrm{S}}$ decreases the memory size of a user process $u$ by $x$ pages.

- The CVM primitive $copy$ copies memory between user processes. So we define $copy_{\mathrm{S}}(u_1, a_1, u_2, a_2, d, c_{\mathrm{CVM}}.up) = (0, c'_{\mathrm{CVM}}.up)$ by $c'_{\mathrm{CVM}}.up(u_2).vm_d(a_2) = c_{\mathrm{CVM}}.up(u_1).vm_d(a_1)$.
- The CVM primitive $get\_vm\_gpr$ reads register $GPR[r]$ of process $u$; we define $get\_vm\_gpr_{\mathrm{S}}(r, u, c_{\mathrm{CVM}}.up) = (c_{\mathrm{CVM}}.up(u).GPR[r], c_{\mathrm{CVM}}.vm)$. As described below, this primitive is used to read parameters of system calls.
- The CVM primitive $set\_vm\_gpr$ writes register $GPR[r]$ of process $u$; we define $set\_vm\_gpr_{\mathrm{S}}(r, u, x, c_{\mathrm{CVM}}.up) = (0, c'_{\mathrm{CVM}}.up)$ by $c'_{\mathrm{CVM}}.up(u).GPR[r] = x$. This primitive is used to set return values of system calls.

The remaining CVM primitives include process initialization ($reset$ and $clone$) or device port I/O ($input$ and $output$).

### 5.3 Abstract Kernels and System Calls

The binary interface of a kernel specifies how user processes can make system calls to the kernel. We describe an exemplary binary interface, also used in the VAMOS kernel [10]: a system call number $j$ is invoked by a trap instruction with immediate constant $j$. System calls have additional parameters that are taken from general purpose registers of the user process; if system call $j$ has $n$ parameters we pass parameter number $x$ with $1 \leq x \leq n$ in register $GPR[10 + x]$ of the calling process. Furthermore, after completion of the system call the kernel notifies the user process of the result of the system call by updating a return value register, e.g. $GPR[20]$, of the calling process.

In a CVM based kernel such a system call interface is implemented as follows. Let the kernel maintain a variable $cu$ that indicates the last process that has been started. Execution of a trap instruction with immediate constant $j$ causes an interrupt with index 5. In the absence of other higher-prioritized interrupts, this interrupt entails a function call $kdispatch(mca, j)$ in the abstract kernel with $mca[5:0] = 100000$ that the kernel then detects as a system call $j$ of process $cu$. Testing the parameter $j$ the kernel determines the number of parameters $n$ and a function $f$ that is meant to handle the system call. It calls the $get\_vm\_gpr$ CVM primitive repeatedly for all $1 \leq x \leq n$ with $fcall(get\_vm\_gpr, e_x, cu, x)$ to set the parameters of the call such that $e_x = c_{\mathrm{CVM}}.up(cu).GPR[10 + x]$. Then, the actual call of the handler is implemented as an ordinary $C0$ function call $fcall(f, r, e_1, \ldots, e_n)$ in the abstract kernel. The return result is passed back to the user ($fcall(set\_vm\_gpr, i, r)$) and the user process is reactivated ($fcall(start, i, cu)$). We see that not only the semantics but also the implementation of a trap interrupt is formally a function call.

## 6 Concrete Kernels and Correctness

The concrete kernel $cc$ is an implementation of the CVM model for a given abstract kernel $ca$. We construct the concrete kernel by *linking* the abstract kernel $ca$, a $C0$ program, with a CVM implementation $cvm$, a $C0_A$ program. Formally, this is written using a link operator $ld$ as $cc = ld(ca, cvm)$. The function table of the linked program $cc$ is constructed from the function tables of the input programs. For functions present in both programs, *defined functions* (with a non-empty body) take precedence over

*declared functions* (without a body). We do not formally define the $ld$ operator here; it may only be applied under various restrictions concerning the input programs, e.g. the names of global variables of both programs must be distinct, function signatures must match, and no function may be defined in both input programs. We require that the abstract kernel $ca$ defines $kdispatch$ and declares all CVM primitives while the CVM implementation $cvm$ defines the primitives and declares $kdispatch$.

### 6.1 CVM Implementation

*Data Structures.* The CVM implementation maintains data structures for the simulation of the virtual machines and multiprocessing. These include: (i) An array of process control blocks $pcb[u]$ for the kernel ($u = 0$) and the user processes ($u > 0$). Process control blocks are structures with components $pcb[u].R$ for every processor register $R$ of the *physical* machine. (ii) The integer array $ptspace$ on the heap holds the page tables of all user processes. Its base address must be a multiple of 4K. (iii) Data structures (e.g. doubly-linked lists) for the management of physical and swap memory (including victim selection for page faults). (iv) The variable $cup$ keeping track of the current user process thus encoding the $c_{\mathrm{CVM}}.cp$ component.

*Entering System Mode.* If the concrete kernel enters system mode, its program rest is initialized with $init_1; init_2$. In all other cases than reset, the first part $init_1$ will (i) write all processor register $R$ to the process control block $PCB[cup].R$ of the process $cup$ that was interrupted and (ii) restore the registers of the kernel from process control block $PCB[0]$. Only after the execution of $init_1$, compiler consistency holds. In the second part $init_2$, the CVM implementation detects whether the interrupt was due to a page fault or for other causes. Page faults are handled silently without calling the abstract kernel (cf. below). For other interrupts, we call $kdispatch$ with parameters obtained from $PCB[cup]$.

*Leaving System Mode.* The $start$ CVM primitive enters user mode again. It is implemented using inline assembler. First, we assign the parameter $u$ of $start$ to $cup$. Second, we write the physical processor registers to $PCB[0]$ to save the concrete kernel state. Third, we restore the physical processor registers for process $u$ from $PCB[u]$ and execute an `rfe` (return from exception).

*Page Fault Handler.* The page fault handler establishes the simulation relation $B$ as described in [12] and summarized in Sect. 3. Only with correct page fault handlers, user mode steps in the physical machine without interrupts simulate steps of a virtual machine. Again, note that a user mode instruction can produce up to two page faults.

To reason about multiple user processes $u$, we have to slightly modify and extend the $B$ relation. Have a virtual machine configuration $c_{\mathrm{V}}$ and a physical machine configuration $c_{\mathrm{P}}$. If user process $u$ is not running, i.e. $cup \neq u$ or $c_{\mathrm{P}}.mode = 0$, we demand that the user-visible processor registers of the process *and* the location and size of its page table (via special-purpose registers $pto$ and $ptl$) are stored in the process control block $pcb[u]$. Thus, we parameterize $B$ over user processes $u$ and set $B(u)(c_{\mathrm{V}}, c_{\mathrm{P}}) = B(c_{\mathrm{V}}, \hat{c}_{\mathrm{P}})$ where $\hat{c}_{\mathrm{P}}$ is defined by $\hat{c}_{\mathrm{P}}.m = c_{\mathrm{P}}.m$ for $m \in \{pm, sm\}$ and $\hat{c}_{\mathrm{P}}.R = c_{\mathrm{P}}.R$ if $cup = u$ and $c_{\mathrm{P}}.mode = 1$ or $\hat{c}_{\mathrm{P}}.R = pcb[u].R$ otherwise.

*Implementation of the CVM Primitives.* The implementation of CVM primitives like $get\_vm\_gpr$ and $set\_vm\_gpr$ is straightforward with the entry and exit mechanism updating the process control blocks described before. For CVM primitives $alloc$ and $free$ the page table length of the process has to to be increased or decreased, resp.; various other data structures concerning memory management have to be adjusted as well. Such operations are closely interconnected with the page fault handler. Since the page tables are accessible as a C0 data structure, inline assembler is only required to clear physical pages. Similarly, the $copy$ implementation requires assembler to copy physical pages.

### 6.2 Simulation Relation Between Abstract Kernel and Concrete Kernel

We proceed as in Sect. 4.2 by defining a simulation relation $konsis(kalloc)(cc, ca)$ that states whether a concrete kernel configuration $cc$ encodes an abstract kernel configuration $ca$; this relation is parameterized over a function $kalloc$ mapping variables in the abstract to variables in the concrete kernel. Note that the concrete kernel may has more variables than the abstract kernel (i) in the global memory frame $cc.lms(0)$ and (ii) on the heap $cc.hm$.

By placing the additional global variables *behind* the global variables of the abstract kernel, indices of variables from the abstract kernel stay unchanged for any memory frame $lms(i)$. Hence, we define $kalloc(ca.lms(i), j) = (cc.lms(i), j)$. Heap variables $(ca.hm, j)$ in the abstract kernel must be mapped injectively to heap variables $kalloc(ca.hm, j) = (cc.hm, j')$ in the concrete kernel. Below we demand that the abstract heap is embedded isomorphically in the concrete heap. For sub variables $Vs$, we trivially extend $kalloc(Vs) = kalloc(V)s$. Now we set $konsis(kalloc)(cc, ca)$ iff (i) program rests and recursion depths coincide, $cc.pr = ca.pr$ and $cc.rd = ca.rd$, (ii) corresponding elementary sub variables $S$ and $kalloc(S)$ have the same value, $va(ca, S) = va(cc, kalloc(S))$, (iii) reachable pointer sub variables $P$ and $kalloc(P)$ must point to corresponding locations, $kalloc(va(ca, P)) = va(cc, kalloc(P))$. For pointers $P$ to heap variables, i.e. $va(ca, P) = (ca.hm, j)$, this establishes a sub graph isomorphism between the heaps of the abstract and the concrete kernel.

### 6.3 Correctness of the Concrete Kernel

Our formulation of a correctness theorem for an operating system kernel written in $C0$ uses the result for virtual memory simulation (Section 3), the compiler correctness theorem (Section 4), Lemma 1 on the execution of inline assembler (Section 4.3), and the simulation relation between abstract kernels and concrete kernels (Section 6).

Consider an initial CVM configuration $c^0_{\mathrm{CVM}}$ with a valid abstract kernel configuration $c^0_{\mathrm{CVM}}.ca$. Let $cc^0 = ld(c^0_{\mathrm{CVM}}.ca, cvm)$ denote the initial configuration of the concrete kernel and let $(c^0_{\mathrm{P}}, c^1_{\mathrm{P}}, \dots)$ denote a physical machine computation with $c^0_{\mathrm{P}}$ code-consistent to $cc^0$. After $z(0)$ steps the physical machine is fully consistent to $cc^0$ under an allocated base address function $aba^0$, i.e. $consis(aba^0)(cc^0, c^{z(0)}_{\mathrm{P}})$.

We construct the list of external events $eevs_{\mathrm{CVM}} = (eev^0_{\mathrm{CVM}}, eev^1_{\mathrm{CVM}}, \dots)$ that parameterizes the CVM computation based on the external event signals $eev^k_{\mathrm{P}}$ seen by the physical processor in step $k$. We sample the external events for non-page-faulting user mode steps. Formally, let the sequence $x(l)$ enumerate these steps ascendingly and

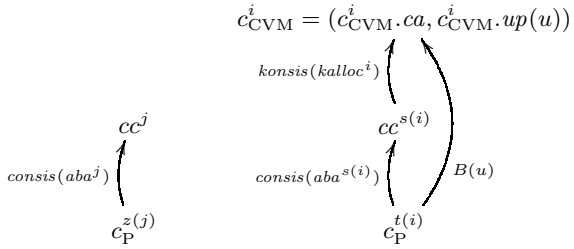


**Fig. 3.** Consistency Relations Between the Different Configurations

define $eev^l_{\mathrm{CVM}} = eev^{x(l)}_{\mathrm{P}}$. Hence, page faults may 'shadow' events with respect to the CVM; this problem must be treated when fully specifying I/O devices.

Let $(c^0_{\mathrm{CVM}}, c^1_{\mathrm{CVM}}, \dots)$ denote the CVM computation parameterized over the external events' list $eevs_{\mathrm{CVM}}$. Then, there exists (i) a sequence of concrete kernel configurations $(cc^0, cc^1, \dots)$, (ii) a sequence of step numbers $z(j)$ and allocated base address functions $aba^j$ relating the physical machine computation to the sequence of concrete kernel configurations, and (iii) a sequence of step numbers $s(i)$ and functions $kalloc^i$ relating the abstract kernel's computation to the sequence of concrete kernel configurations such that for all $i$ and $j$ the following simulation relations hold:

- Configuration $cc^{s(i)}$ of the concrete kernel after step $s(i)$ encodes configuration $c^i_{\mathrm{CVM}}.ca$ of the abstract kernel after step $i$, so $konsis(kalloc^i)(c^i_{\mathrm{CVM}}.ca, cc^{s(i)})$.
- Configuration $c_{\mathrm{P}}^{z(j)}$ of the physical machine after step $z(j)$ encodes configuration $cc^j$ of the concrete kernel after step $j$, so $consis(aba^j)(cc^j, c_{\mathrm{P}}^{z(j)})$.
- For all user processes $u$ the configuration $c^i_{\mathrm{CVM}}.up(u)$ of virtual machine $u$ after step $i$ of the CVM machine is encoded by the configuration $c_{\mathrm{P}}^{t(i)}$ of the physical machine after step $t(i) = z(s(i))$. With the $B$ relation introduced in Sect. 3 and parameterized in Sect. 6, we require $B(u)(c^i_{\mathrm{CVM}}.up(u), c_{\mathrm{P}}^{t(i)})$.
- The physical machine computation and the computation of the concrete kernel must fit together. Unless $cc^j.pr = asm(u, \texttt{rfe}); r$, i.e. the program rest starts with assembler code that returns to user mode, $cc^{j+1}$ is computed by $\delta_{\mathrm{C0A}}$ parameterized with the current allocated base address function $aba^j$ applied to the current physical machine configuration $c_{\mathrm{P}}^{z(j)}$ and $cc^j$. Formally, $(cc^{j+1}, c_{\mathrm{P}}^{z(j+1)}) = \delta_{\mathrm{C0A}}(aba^j)(cc^j, c_{\mathrm{P}}^{z(j)})$. Observe that the $aba$ parameter and the second input for $\delta_{\mathrm{C0A}}$ are used only for executing inline assembler code. In the other case, i.e. if $cc^j.pr = asm(u, \texttt{rfe}); r$, the next configuration of the concrete kernel $cc^{j+1}$ is obtained from $cc^j$ by setting $cc^{j+1}.pr = init_2$ and $cc^{j+1}.rd = 1$.

The claim of the correctness theorem is illustrated in Fig. 3. Its proof is by induction on $i$ with a case split along the cases of the CVM semantics from Sect. 5. In the proof the sequence numbers $z(j)$ and $s(i)$ are defined inductively: (i) Unless $cc^j.pr = asm(u, \texttt{rfe}); r$ we set $z(j+1)$ as in the induction step of the compiler correctness theorem. If the program rest starts with $asm(u, \texttt{rfe})$ just before returning to user mode, we set $z(j+1)$ to the index of that that system mode configuration that marks the completed initialization part $init_1$ of the concrete kernel. This resembles the base case

of compiler correctness. (ii) We set $s(i+1) = s(i)$ if $c^i_{\mathrm{CVM}}.cp \neq 0$ or $c^i_{\mathrm{CVM}}.pr = fcall(start, v, e_1); r$, $s(i+1) = s(i)+1$ if $c^i_{\mathrm{CVM}}.cp = 0$ and if $c^i_{\mathrm{CVM}}.ca.pr$ does not start with a CVM primitive call. In this case, the concrete kernel simulates one step of the abstract. We set $s(i+1) = s(i)+x$ if $c^i_{\mathrm{CVM}}.cp = 0$ and if the program rest $c^i_{\mathrm{CVM}}.ca.pr$ starts with a call of a CVM primitive other than $start$, and $x$ is the number of steps the implementation of the CVM primitive takes to return.

## 7 Status of the Formal Verification

At the time of this writing a considerable part of the presented work has been formalized in the theorem prover Isabelle/HOL [19]: (i) based on the specification of the VAMP processor [12,20] (a DLX-like processor verified in PVS) we have specified a formal VAMP assembler semantics, (ii) we have defined formal semantics for $C0$ and $C0_A$, (iii) we specified as an Isabelle/HOL function, implemented in $C0$, and verified the compiler's code generation, (iv) large parts of the compiler simulation theorem (Sect. 4.2) have been verified (we plan to finish this proof until fall 2005), (v) data structures and algorithm used in the CVM implementation have been specified and verified, (vi) the CVM and the VAMOS microkernel semantics have been formally specified.

## 8 Summary and Further Work

The work presented here depends crucially on a recent theory of virtual memory simulation from [12] and a compiler correctness proof in form of a step by step simulation theorem from [13]. We have presented the new abstract CVM model. In this model the formalisms for machine language specification and for programming language semantics have been combined in a natural way, allowing to treat system calls not only intuitively but also formally as function calls. Also, due to the parallelism, CVM permits to specify operating system kernel without reference to inline assembler code. We have provided an approach to the pervasive verification of an operating system kernel, which handles virtual memory and is written in a high level language with inline assembler code, and outlined substantial parts of its proof.

The 'trivial' further work is the completion of the formal verification effort that we expect to be completed in 2006 if things go well or in 2007 if things go not so well. In the next years CVM will be used in the Verisoft project [10] in several places: (i) a correctness proof for a simple operating system (called SOS) will be based on CVM with a particular abstract kernel (called VAMOS) inspired by [6]. (ii) Based on verified hardware [12,20], verified compilers, and SOS the verification of entire systems for electronic signatures of emails and for biometric access control will be attempted.

## References

1. Hutt, A.E., Hoyt, D.B., Bosworth, S., eds.: Computer Security Handbook. John Wiley & Sons, Inc., New York, NY, USA (1995)
2. Shapiro, J.S., Hardy, N.: Eros: A principle-driven operating system from the ground up. IEEE Software **19** (2002) 26–33

3. Rushby, J.: Proof of separability: A verification technique for a class of security kernels. In: Proc. 5th International Symposium on Programming, Turin, Italy, Springer (1982) 352–367
4. Wulf, W.A., Cohen, E.S., Corwin, W.M., Jones, A.K., Levin, R., Pierson, C., Pollack, F.J.: HYDRA: The kernel of a multiprocessor operating system. CACM **17** (1974)
5. Pfitzmann, B., Riordan, J., Stüble, C., Waidner, M., Weber, A.: The PERSEUS system architecture. In Fox, D., Köhntopp, M., Pfitzmann, A., eds.: VIS 2001, Sicherheit in komplexen IT-Infrastrukturen, Vieweg Verlag (2001) 1–18
6. Liedtke, J.: On micro-kernel construction. In: Proceedings of the 15th ACM Symposium on Operating systems principles, ACM Press (1995) 237–250
7. The Common Criteria Project Sponsoring Organisations: Common Criteria for Information Technology Security Evaluation version 2.1, Part I. `http://www.commoncriteriaportal.org/public/files/ccpart1v21.pdf` (1999)
8. Bevier, W.R.: Kit: A study in operating system verification. IEEE Transactions on Software Engineering **15** (1989) 1382–1396
9. OSEK group: OSEK/VDX time-triggered operating system. `http://www.osek-vdx.org/mirror/ttos10.pdf` (2001)
10. The Verisoft Consortium: The Verisoft project. `http://www.verisoft.de/` (2003)
11. Aho, A.V., Hopcroft, J.E., Ullman, J.: Data Structures and Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
12. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. Technical report, Verisoft project (2005) `http://www.verisoft.de/.rsrc/SubProject2/verificationmm.pdf`.
13. Leinenbach, D., Paul, W., Petrova, E.: Compiler verification in the context of pervasive system verification. Draft manuscript (2005)
14. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Second edn. Morgan Kaufmann, San Mateo, CA (1996)
15. Müller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer (2000)
16. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. John Wiley & Sons, Inc., New York, NY, USA (1992, revised online version: 1999)
17. Winskel, G.: The formal semantics of programming languages. The MIT Press (1993)
18. Norrish, M.: C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory (1998)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of Lecture Notes in Computer Science (LNCS). Springer (2002)
20. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP processor. In Geist, D., Tronci, E., eds.: CHARME '03, Springer (2003) 51–65