Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 217 (2008) 169–185

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Correct Microkernel Primitives

Artem Starostin[a,1] Alexandra Tsyban[a,2]

[a] *Computer Science Department, Saarland University*
P.O. Box 15 11 50, 66041 Saarbrücken, Germany

Abstract

Primitives are basic means provided by a microkernel to implementors of operating system services. Intensively used within every OS and commonly implemented in a mixture of high-level and assembly programming languages, primitives are meaningful and challenging candidates for formal verification. We report on the accomplished correctness proof of academic microkernel primitives. We describe how a novel approach to verification of programs written in C with inline assembler is successfully applied to a piece of realistic system software. Necessary and sufficient criteria covering functional correctness and requirements for the integration into a formal model of memory virtualization are determined and formally proven. The presented results are important milestones on the way to a pervasively verified operating system.

Keywords: Formal Verification, Theorem Proving, Operating System, Microkernel, Inline Assembler.

1 Introduction

Correctness guarantees for computer systems is a hot research topic. Since there are a lot of examples when the correctness of separate computer components has been successfully established, the formal verification of an entire industrial-size system is being brought to the forefront. In [8] Moore, the head of the famous CLI project, proposes the grand challenge of whole computer system pervasive verification.

Verisoft[3] is a research project inspired by the problem of a complete computer system correctness. The project aims at the development of the pervasive verification technology [10] and demonstrating it by applying to an exemplary computer system. A prototypic system comprises (i) a pipelined microprocessor with memory management units, (ii) a number of devices, in particular, a hard disk, (iii) a

[1] Email: starostin@wjpserver.cs.uni-sb.de. Work was supported by the International Max Planck Research School for Computer Science (IMPRS).

[2] Email: azul@wjpserver.cs.uni-sb.de. Work was supported by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project.

[3] www.verisoft.de

doi:10.1016/j.entcs.2008.06.048

microkernel, (iv) a simple operating system, and (v) an exemplary user application. Pervasive formal verification of the whole system is attempted. The process is supported by a variety of computer aided verification tools, both interactive and automated, in order to minimize the possibility of errors induced by verification engineers.

This work relates to the problem of operating system microkernel correctness. A microkernel is the minimal kernel which, basically, provides no operating system services at all, but only the mechanisms necessary to implement such services. The mechanisms include process and memory management, address spaces, low-level IPC, and I/O. Usually, they are implemented in the form of *primitives*, microkernel routines which provide this functionality to the upper layers. Since every service of an operating system makes use of primitives, the correctness of the latter becomes of special importance.

In the current paper we discuss the correctness issues of primitives of an academic operating system microkernel. We describe how the methodology for the system software verification developed in the frame of Verisoft is successfully applied to primitives implemented in C with inline assembler. We outline the correctness criteria of microkernel primitives. Stating the correctness theorems we show what it means that a primitive fulfills these correctness criteria. We sketch a general idea how such theorems are proven. In a case study we elaborate on particular for the example details of specifications and proofs.

The contribution of this paper is that (i) *all* necessary and sufficient correctness criteria of primitives of a microkernel for a pervasively verified system are determined and formally proven, (ii) a novel, convenient for formal use, approach to verification of C with inline assembler programs is presented, and (iii) an important part of a realistic microkernel is proven correct showing that seamless formal verification of crucial parts of operating systems is feasible. All material presented in the paper is supported by formal theories in a computer aided theorem prover.

Related Work

A number of research projects suggest ideas to microkernel verification. Choosing reasoning either in C or assembler semantics, to the best of our knowledge, nobody exploits their combination. The L4.verified project, targets at constructing seL4 [4], a formally verified operating system kernel. From the system's prototype designed in Haskel both formal model and C implementation are generated. A richer compared to Verisoft subset of C including pointer arithmetic is used, which, however, provides less expressive semantics than inline assembler as the latter makes possible to accesses even registers of a processor. A substantial progress seems to be achieved in the verification of the model, but only exemplary parts of the source code are reported verified. The FLINT project exploits an x86 assembly code verification environment for certification of context switching routines [9], an important microkernel part. No results on integration of object code correctness into a high-level programming language are reported. The recent Robin project aims at the verification of Nova microhypervisor [12]. Although implementation is in (a subset of) C++ with inline assembler, the verification is planned to cover only C++ parts. Currently there is no connection to real object code, which seems to be planned for the (far) future. It is planned to build a model precise enough to catch virtual memory aliasing and address space separation errors, however it is unclear whether these properties will be shown to be respected by the hypervisor's implementation.

Outline

In Sect. 2 we discuss implementation issues and formal model of a microkernel. We briefly formalize all concepts necessary to present the microkernel correctness criteria which have to be satisfied by its primitives. Next, in Sect. 3, we elaborate on our verification methodology and sketch the semantics of C programs with inline assembler parts. In Sect. 4 we proceed with the correctness theorem for a primitive. The presented approach is supported by the case study in Sect. 5 for which the primitive that copies data between processes is selected. We conclude in Sect. 6.

Notation

We denote the set of boolean values by $\mathbb{B}$ and the set of natural numbers including zero by $\mathbb{N}$. We denote the set of natural numbers less then x by $\mathbb{N}_x$. We denote the list of n elements of type T by T^n. The elements of a list x are accessed by $x[i]$, its length is denoted by $|x|$. The operator $\langle x \rangle$ yields for a bit string $x \in \mathbb{B}^n$ the natural number represented by x. We allow to interchange a bitvector x with its value $\langle x \rangle$. The set of all possible configurations of a concept x is defined by C_x.

2 An Academic Operating System Microkernel

We consider an exemplary academic microkernel which provides mechanisms for the (i) process and memory management, (ii) address spaces, (iii) IPC, and (iv) device communication.

2.1 Implementation Issues

The microkernel implements the *Communicating Virtual Machines (CVM)* [3] model which defines the parallel execution of concurrent user processes interacting with a kernel. According to the model the microkernel is split into two logical parts: (i) the *abstract* kernel which provides an interface to a user or an operating system and could be implemented in a pure high-level programming language, and (ii) the *lower layers* which implement the desired functionality stated in the beginning of Sect. 2. The implementation of the low-level functionality necessarily contains assembler portions because processor registers and user processes could not be accessed by ordinary C variables. By linking the two kernel parts together the *concrete* kernel, a program which can run on a target machine, is obtained.

The kernel lower layers could be split into three logical parts: (i) primitives, (ii) a page fault handler, and (iii) context switch routines. Within the paper we

Name	Description	Comment
copy	copies data between processes	A
phys_copy	copies data between virtual and the physical memory	A
get_vm_word	reads a word from the virtual memory of a process	A
set_vm_word	writes a word to the virtual memory of a process	A
out_word	writes a word to a device	AD
in_word	reads a word from a device	AD
virt_io	copies data between a device and a process	AD
phys_io	copies data between a device and the physical memory	AD
phys_io_range	I/O operations on port ranges	AD
reset	initializes the memory and the registers of a process	
get_vm_gpr	reads a register of a process	
set_vm_gpr	writes a register of a process	
alloc	gives additional memory to a process	
free	releases a given amount of the memory of a process	
clone	clones a process	
set_mask	mask external interrupts	

Table 1
List of primitives of the microkernel

discuss the correctness of primitives. They are implemented in the C0 programming language [7], a slightly restricted C, with inline assembler parts. In brief, the limitations of C0 compared to standard C are as follows. Prefix and postfix arithmetic expressions, e.g., $i{+}{+}$, are forbidden, as well as function calls as a part of expressions. Pointers are typed and do not point to local variables or to functions. Void pointers and pointer arithmetic are not supported. The size of arrays has to be statically defined.

2.2 Primitives

The academic microkernel contains 16 primitives described in Table 1. The comment 'A' denotes that a primitive has an inline assembler portion. The comment 'D' designates that a primitive accesses devices. Thus, the primitives can be divided into three groups: (i) 7 primitives implemented in pure C0, (ii) 4 primitives which have assembler portions, and (iii) 5 primitives which have assembler portions and access devices. In this paper we give the methodology for verification of code written in C0 with inline assembler. It is applicable to all the primitives. However, we have verified so far primitives from the second group.

2.3 A Formal Model

The CVM model defines a parallel execution of the kernel and N user processes on an underlying physical machine with a hard disk. According to CVM, the C0 language semantics is used to model the computation of the kernel, and semantics of virtual machines models the computation of user processes. In the following, we outline the necessary concepts of the model: (i) physical and virtual machines [3], (ii) a hard disk [5], and (iii) C0 machines [7]. Having them, we sketch the CVM semantics and give its correctness criteria. For details cf. [6]. Memories of physical and virtual machines are conceptually organized in pages of P machine words.

2.3.1 Physical Machines

Physical machines are the sequential programming model of the VAMP hardware [2] as seen by a system software programmer. They are parameterized by (i) the set $SAP \subseteq \mathbb{B}^5$ of *s*pecial purpose register *a*ddresses visible to *p*hysical machines, and (ii) the number TPP of *t*otal *p*hysical memory *p*ages which defines the set $PMA = \{a \mid 0 \leq \langle a \rangle < TPP \cdot P\} \subseteq \mathbb{B}^{30}$ of accessible *p*hysical *m*emory *a*ddresses. The machines are records $pm = (pc, dpc, gpr, spr, m)$ with the following components: (i) the normal $pm.pc \in \mathbb{B}^{32}$ and the delayed $pm.dpc \in \mathbb{B}^{32}$ program counters used to implement the delayed branch mechanism, (ii) the general purpose $pm.gpr \in \mathbb{B}^5 \mapsto \mathbb{B}^{32}$ and the special purpose $pm.spr \in SAP \mapsto \mathbb{B}^{32}$ register files, and (iii) the word addressable physical memory $pm.m \in PMA \mapsto \mathbb{B}^{32}$.

The computation is possible in two modes: user and system. In user mode a memory access to a virtual address va is subject to address translation. It either redirects to the translated 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 $v(pm, va)$ maintained by the memory management unit of the physical machine. When on, it signals that the page storing the virtual address va resides in the main memory, elsewise, it is on a hard disk.

The semantics of an uninterrupted execution is defined by the underlying *i*nstruction *s*et *a*rchitecture (ISA). On an interrupt signal, which could be internal or external, the machines switches to the system mode and invokes a special piece of software—an interrupt handler. Within the paper, we are interested in two particular kinds of interrupts: (i) page faults, and (ii) system call exceptions. A page fault is treated by the *page fault handler*, a routine which translates addresses and loads missing pages from a hard disk into the physical memory. Its implementation servers several purposes. For instance, it could be used to handle a page fault and to guarantee that no page fault will occur within a certain period in the future. The latter property is needed for the primitives, thus, they heavily call the handler (for details cf. Sect. 2.5). System call exceptions occur due to a special instruction, called the *trap*. It is used by an assembler programmer in order to invoke one of the system calls provided by the operating system microkernel. System calls, viewed from a simplified perspective, are just the wrappers around the primitives.

2.3.2 Virtual Machines

Virtual machines are the hardware model visible for user processes. They give user an illusion of an address space exceeding the physical memory. No address translation is required, hence page faults are invisible. The virtual machine's parameters are: (i) the number TVP of *t*otal *v*irtual memory *p*ages which defines the set of accessible *v*irtual *m*emory word *a*ddresses $VMA = \{a \mid 0 \leq \langle a \rangle < TVP \cdot P\} \subseteq \mathbb{B}^{30}$, and (ii) the set $SAV \subseteq SAP$ of *s*pecial purpose registers *a*ddresses visible to *v*irtual machines. Their configuration, formally, is a record $vm = (pc, dpc, gpr, spr, m)$ where only $vm.spr \in SAV \mapsto \mathbb{B}^{32}$ and $vm.m \in VMA \mapsto \mathbb{B}^{32}$ differ from the physical machines. Semantics is completely specified by the ISA with the following exception. Due to safety reasons we split the set SAV into two parts: (i) the set SAV_R of *read only* register addresses, and (ii) the set SAV_W of addresses of registers that could be *completely* accessed by a user. A write attempt to a register $vm.spr[x]$ with $x \in SAV_R$ has no effect. The set SAV_R contains the register *ptl* (page table length). It stores the amount of virtual memory allocated to the process measured in pages. We abbreviate $vm.spr[ptl] = vm.ptl$.

2.3.3 Integrating a Hard Disk

We use the formal model of a hard disk based on the ATA/ATAPI protocol. We denote the configuration of the hard disk by *hd*. Only the component $hd.sm \in \mathbb{N}_{2^{30}} \mapsto \mathbb{N}_{2^{32}}$ which models the disk content as a word-addressable memory is of our interest. A step of the system (pm, hd) comprising the physical machine and the hard disk is denoted by the function $\delta(pm, hd) = (pm', hd')$. If no write instruction to the disk is executed only the physical machine is updated according to its semantics. Otherwise, both *pm* and *hd* are changed.

2.3.4 C0 Machines

A C0 machine is a record $c = (pr, tt, ft, rd, lms, hm)$. Its components are: (i) the program rest $c.pr$, a sequence of statements which still has to be executed, (ii) the typetable $c.tt$ which collects information about types used in the program, (iii) the function table $c.ft$ storing information about functions of the program, (iv) the recursion depth $c.rd$, (v) the local memory stack $c.lms$ mapping numbers $i \leq c.rd$ to memory frames which implement a relatively low-level memory model and comprise components for the number of variables in a frame, their names, types, and contents, and (vi) the heap memory $c.hm$ which is a memory frame as well.

The global memory of a C0 machine c is $c.lms(0)$. The top local memory frame is denoted by $top(c) = c.lms(c.rd)$. A memory frame first includes the parameters of the corresponding function. A variable of a machine c is a pair (m, i), where m is a memory frame and i is the number of the variable in the frame. By $va(c, i) = (top(c), i)$ we denote the i-th variable of the current function context.

2.3.5 Communicating Virtual Machines

The CVM configuration is formally a record $cvm = (up, ak, cp)$ with the following components: (i) the list of N user processes $cvm.up \in C_{vm}^N$ represented by virtual machines, (ii) the abstract kernel $cvm.ak \in C_c$ modeled by a C0 machine, and (iii) the current process identifier $cvm.cp \in \mathbb{N}_N$, where $cvm.cp = 0$ stands for the kernel. The CVM semantics distinguishes user and kernel computations. In case $cvm.cp \neq 0$ the user process $pid = cvm.cp$ is intended to make a step. In case no interrupt occurs it boils down to the step of the virtual machine $cvm.up[pid]$. Otherwise, the kernel dispatcher is invoked and the kernel computation starts. The kernel dispatcher handles possible page faults and determines whether a primitive f is meant to be executed. In case it is, the parameters of the primitive p_f are extracted by means of the system call mechanism. The specification f_S is applied to the user processes $cvm'.up = f_S(cvm.up, p_f)$. Next, the user computation resumes.

2.4 Correctness Criteria

Microkernel correctness requirements have to relate: (i) the implementation of kernel lower layers, encoded by the C0 machine c, (ii) the CVM model cvm, and (iii) the physical machine with the hard disk (pm, hd).

The implementation c is related to the CVM model by means of linking. We use the formal specification of the linking operator $link(cvm.ak, c) = k$. It takes two C0 machines encoding the abstract kernel and the implementation of its lower layers, respectively, and produces the concrete kernel k, also a C0 machine. We state that the concrete kernel k correctly runs on the physical machine pm by means of the C0 compiler consistency relation (cf. Sect. 3.2).

The correctness criteria for the user processes is hidden inside the memory virtualization relation. This simulation relation, called the $\mathcal{B}$-relation, specifies a parallel execution of the user processes $cvm.up$ on one physical machine pm. In order to specify the $\mathcal{B}$-relation, let us first give a notion of a *process control block (PCB)*. The PCBs are C0 data structures permanently residing in the memory of the underlying physical machine. They store the information about visible registers of all user processes. Thus, we are able to reconstruct user virtual machines from the contexts stored in the PCBs. The function $virt(pid, pm, hd) = vm$ yields the virtual machine for the process pid by taking the register values from the corresponding PCB fields. The memory component $vm.m$ of the built virtual machine is constructed out of the physical memory and the data on the hard disk depending on where a certain memory page lies:

$$vm.m[a] = \begin{cases} pm.m[pma(pid, a)] & \text{if } v(pm, a) \\ hd.sm[sma(pid, a)] & \text{otherwise} \end{cases} .$$

The physical memory address is computed by the function $pma(pid, a)$ while the swap memory address is yielded by the function $sma(pid, a)$ (for the definitions cf. [1,6]). Then, the $\mathcal{B}$-relation is defined formally as follows:

$$\mathcal{B}(cvm.up, pm, hd) = \forall pid \in \mathbb{N}_N \ : \ virt(pid, pm, hd) = cvm.up[pid].$$

There is a number of additional correctness demands omitted due to the space limitations.

2.5 A Page Fault Handler

The $\mathcal{B}$-relation can only be maintained with an appropriate page fault handler. The page fault handler is a routine which serves two purposes. Called for a virtual address *va* and a process identifier *pid* it (i) yields to the caller the translated physical memory address *pma*(*pid*, *va*), and (ii) guarantees that the page storing *pma*(*pid*, *va*) resides in the physical memory of the machine running the handler.

Possibly called twice in a primitive in order to translate addresses for different processes, it must respect the following. An appropriate page fault handler must not swap out the memory page that was swapped in during a previous call to it. In order to guarantee this a proper page eviction strategy must be used. We support two lists, called *active* and *free*, for the page management. Together they describe all pages of physical memory accessible to a user. Items of the free list describe the pages that immediately could be given to user, i.e., without swapping out a page to the hard disk. Active list describes physical pages that store a virtual page. When all physical memory is occupied, a page from the active list is evicted and replaced by the one loaded from the hard disk according to the FIFO strategy. For formal details and correctness issues cf. [1].

3 Verification Approach

There are several possibilities to argue about the correctness of kernel lower layers, and in particular of primitives. One might have an idea to reason about their object code in the machine language semantics. Due to the huge size of the target code—the kernel lower layers translated by the C0 compiler are 11K lines long—this approach seems to be unfeasible for the interactive verification. Running to extremes, one can try to verify system code on a very high-level of abstraction, e.g., by means of a generic verification environment for imperative programs [11], and then transfer the results down to the necessary level introducing refinement theorems. However, techniques that allow reasoning about mixture of C and assembler code in such environments were only recently invented (the approach is used in [1]). They basically aim at big C programs with assembler portions, isolated in separate functions. Since this is not the case for the primitives—they are relatively small C functions which can have several inline assembler parts—we decided to do the formal verification in a mixture of C0 small step and inline assembler semantics.

3.1 Verification Environment

We use Isabelle/HOL theorem prover as the basis for the verification environment. All the concepts and their semantics listed in Sect. 2.3 were formalized in Isabelle by the colleagues in the Verisoft project. The source code of the primitives is automatically translated by a tool into the C0 small step semantics in Isabelle.

3.2 C with Inline Assembler Semantics

A C0 configuration *c* is related to the underlying physical machine *pm* by the compiler simulation relation *consis*(*alloc*)(*c*, *pm*) parameterized over an allocation function *alloc* which maps C0 variables to the physical memory cells. Essentially, the relation is a conjunction of the following facts: (i) *value* consistency: the respective

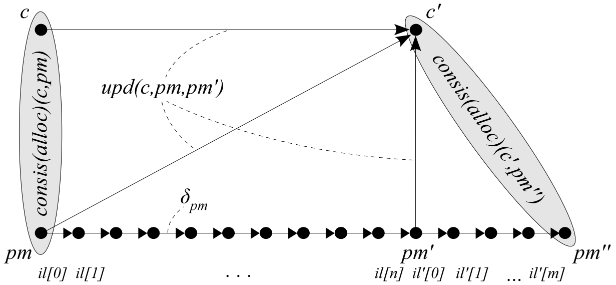

Figure 1. Switching C and assembler semantics

variables of c and pm have the same values and the reachable portions of the heaps in $c.hm$ and $pm.m$ are isomorphic, (ii) *control* consistency: the delayed program counter $pm.dpc$ points to the start of the translated code of the first statement of $c.pr$ and $pm.pc = pm.dpc + 4$, (iii) *code* consistency: the compiled code lies at the correct address in the memory $pm.m$, and (iv) *stack* consistency: the heap resp. stack pointers which reside in the registers $pm.gpr[29]$ resp. $pm.gpr[30]$ point to the first free address of $c.hm$ resp. to the beginning of $top(c)$. For details cf. [7].

An assembler instruction list il can be integrated by a special statement $asm(il)$ into the C0 code. As long as no such statement occurs the C0 semantics is applied. The former approach to deal with verification of an assembler statement is to maintain the compiler consistency relation with execution of every single instruction from il (cf. Sect. 4.3 of [3]). This method turned out to be inconvenient due to excessive complexity of formal proofs, therefore a new one was developed and used.

In brief, the novel approach is as follows. On an assembler statement the execution is switched to the consistent underlying physical machine and continues directly there. When the assembler instructions have been executed we switch back to the C0 level. For this we have to update the C0 machine possibly affected by the assembler instructions. The allocation function *alloc* makes it possible to determine which variables of the C0 machine 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 w.r.t. the allocation function *alloc*, i.e., $consis(alloc)(c, pm)$. From the consistency relation we have that the program counters of pm point to the address of the assembler statement: $pm.dpc = ad(asm(il))$ and $pm.pc = pm.dpc + 4$, where $ad(s)$ yields for a statement s its address in the memory of pm. This allows us to start reasoning about the correctness of the assembler 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 machine 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' (cf. Fig. 1). A number of restrictions are imposed on the changes in the physical machine, which guarantee that the C0 machine is not destroyed by the assembler 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 memory region where the compiled code is stored stays the same, i.e., we forbid self-modifying code, (iii) the stack and heap pointers are unchanged: $pm'.gpr[x] = pm.gpr[x]$ for $x \in \{29, 30\}$, (iv) the memory occupied be the local memory frames remains the same except for $top(c)$, and (v) pointers change is forbidden except setting them to *null.* We formally prove that we deal with assembler portions which meet these restrictions.

The program rest is updated straightforwardly—the assembler statement is removed, i.e., $c'.pr = r$. The memory update proceeds separately for the global, the top local, and the heap memories. For each of them the respective memory cells of the physical machines configurations pm and pm' are compared. In case a memory cell at an address a is changed, the value of the variable x, s.t. $alloc(c', x) = a$ is updated with $pm'.m[a]$. However, the compiler correctness relation does not necessarily hold between the C0 configuration c' and the physical machine pm'. The control consistency will be broken if the assembler statement $asm(il)$ is either (i) the last statement of a loop body, or (ii) the last statement of the 'then' part of a conditional statement. The translation of these statements to the target code results in a list of assembler instructions il' which has to be executed by the machine pm' in order to reach a consistent to c' state. Note that il' contains only control instructions, and, hence does not affect any C0 variable. Executing il' we transit from pm' to pm'' updating the program counters and regain consistency $consis(alloc)(c', pm'')$.

4 Correctness of a Primitive

Since primitives are parts of the microkernel, their correctness is closely related to the correctness of the whole kernel. Execution of a primitive is one of the induction cases of the overall kernel correctness theorem [6]. We distinguish two main theorems for each primitive: (i) the primitives functional correctness, and (ii) the top-level correctness of a primitive. The latter is used to prove the induction case of the overall kernel correctness theorem and, therefore, claims correctness criteria needed for the integration, for instance that the abstract kernel data is not corrupted. The former is used as an auxiliary theorem to prove the latter. It states the correctness of the input-output relation of a primitive call. Such modularization increases the robustness of formal theories to the possible code changes, e.g., due to the errors disclosed during the verification. In this case, one has to adapt the proofs only of the first theorem which is much simpler than the second one. Next, we present the general idea behind these theorems and discuss their formal proofs.

4.1 Functional Correctness

The functional correctness justifies the input/output relation of a primitive. We start in some C0 state k encoding the concrete kernel and consistent to the underlying physical machine pm and claim the requirements $pre_f(k, pm)$ to a primitive f caller. We end in the resulting state obtaining the desired values $post_f(k', pm')$ of C0 variables and memory cells of the physical machine. Note that both pre- and postconditions, in general, speak not only about values of C0 variables, but also about the memory parts of the underlying machine which are not accessible via variables but are subject to change by inline assembler code. The straightforward idea of the functional correctness is reflected in the next theorem.

Theorem 4.1 (Functional Correctness of a Primitive) Let k be the concrete kernel calling the primitive f with the parameters p_f: $k.pr = f(p_f); r$. Let (pm,hd) be the configuration of the underlying physical machine with the hard disk, s.t. it is consistent to the concrete kernel: $consis(alloc)(k, pm)$. Assume that the precondition $pre_f(k, pm)$ to the primitive is satisfied, then there exist (i) a number of steps T of the physical machine with the hard disk, s.t. $(pm', hd') = \delta^T(pm, hd)$, and (ii) a configuration of the concrete kernel k' with an appropriate allocation function $alloc'$, s.t. they are consistent to the physical machine: $consis(alloc')(k', pm')$, and the desired postcondition $post_f(k', pm')$ holds.

In our experience it is inconvenient to prove such theorems directly. We rather create several separate lemmas of the same form but speaking about the code in *different* semantics. For example, if a primitive contains a number of assembler instructions wrapped both from the beginning and the end in C0 code, we create three lemmas: one for the C0 part before assembler, next for the assembler portion, and, finally, for the remaining C0 part. This simple idea is easily scalable to arbitrary combinations of C and assembler. We prove such lemmas by applying C0 and inline assembler semantics. The crucial point is the construction of a consistent C0 machine after the assembler part execution. We proceed as described in Sect. 3.2.

The verification proceeds, certainly, with respect to the total correctness criteria, i.e., we show the termination and the absence of run-time errors. The machinery for this is hidden inside the C0 small step semantics. The set C_c of all possible C0 configurations is represented formally in Isabelle by the *option* type which extends C_c with an additional *error* state. The semantics is constructed in a way that the computation ends in a non-error state only in the case that no run-time errors occur. We formally show that the resulting configuration of a primitive call is not in the faulting state. We do this in an iterative fashion, i.e., we show that the execution of every single statement brings some sensible configuration. This could happen only in case all expressions of the statements are correctly evaluated. The correctness demands to the expression evaluation forces us to show formally that neither null-pointer dereference nor out-of-boundary array access happens. This also proves the termination of single statements. In order to guarantee the termination of a whole program provided that statements terminate we have to show that neither infinite loops nor infinite recursive calls occur. Since we do not use recursion in the kernel implementation, we pay attention only to loops. Their termination is closely related to the way loops are verified in the C0 semantics. The correctness of a loop is established by an inductive lemma. We formally specify the number of the loop iterations by a ranking function over the variables modified in the loop. Since we proceed by induction on the result yielded by the ranking function, the termination follows. We give details in the example (cf. Sect. 5.3). The absence of run-time errors in assembler portions boils down to the absence of interrupts conditions which are required to the proven by the inline assembler semantics. The

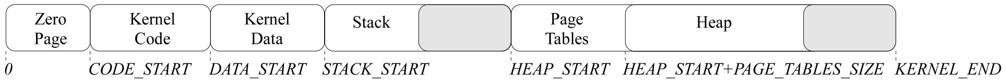

Figure 2. Memory structure of the microkernel claimed by the kernel invariant

termination of assembler loops is proven analogously to C0 loops.

4.2 Correctness in the Context of the Kernel

The correctness criteria needed for the integration are, basically, split into two parts. They are: (i) the kernel correctness requirements stated in Sect. 2.4, and (ii) the *kernel invariant* which turns out to be necessary to be proven first. The kernel invariant $inv(cvm, k, pm, hd)$ is the conjunction of the following statements: (i) the memory map properties, (ii) the page fault handler invariants, (iii) the validity of the C0 machine encoding the concrete kernel, and (iv) the hard disk properties and liveness requirements for the system 'hard disk - physical machine'.

4.2.1 Memory Map Properties

The kernel code has a particular alignment in the memory. Its data structures lie both in the global and the heap memories. For safety reasons we must know these regions, and know which of their parts could be changed with every step of the kernel, for instance with the execution of a primitive. Fig. 2 depicts the memory structure which we describe formally.

4.2.2 Page Fault Handler Invariants

As mentioned in Sect. 2.5, the page fault handler is (heavily) called by the kernel. The handler maintains a variety of global data structures, in particular lists for page management. Therefore, we must claim that no functions besides the page fault handler are allowed to modify its data structures. Due to the complexity of the page fault handler its verification is attempted by means of the refinement technique which connects its representation on several semantical layers. In order to support that approach, we formally preserve: (i) the mapping between the implementation of the kernel lower layers c which contains the handler and the PFH abstraction, and (ii) the validity properties over the handler abstraction. The page fault handler properties relevant for the primitives correctness comprise: (i) for distinct pairs $(pid_1, va_1) \neq (pid_2, va_2)$ the translated physical addresses are distinct: $pma(pid_1, va_1) \neq pma(pid_2, va_2)$, (ii) every physical address is associated exactly with one pair (pid, va), and (iii) all translated addresses lie outside the kernel range: $\forall (pid, va) \; : \; pma(pid, va) \notin [0 : KERNEL\_END)$.

Next, we present the top-level correctness theorem of a primitive execution. It turns out that its proof requires several static properties $pro(cvm.ak, c)$ over the abstract kernel and the implementation of the kernel lower layers. They are the necessary preconditions to a correct linking and state, not exclusively, the following: (i) the function tables $cvm.ak.ft$ and $c.ft$ encode the same function signatures, (ii) all external function declarations in $cvm.ak.ft$ have an implementation in $c.ft$ and vice

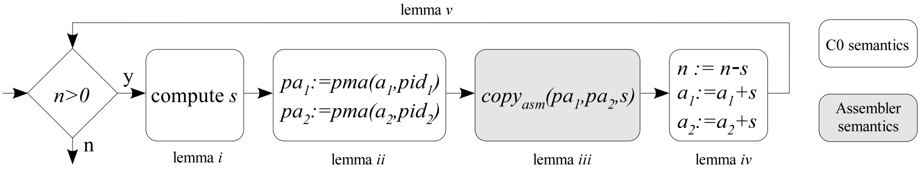

Figure 3. Algorithm and structure of auxiliary lemmas for the *copy* primitive

versa, and (iii) the type tables $cvm.ak.tt$ and $c.tt$ encode the same types.

Theorem 4.2 (Top-level Correctness of a Primitive) Let k be the concrete kernel calling the primitive f with the parameters p_f: $k.pr = f(p_f); r$. Let cvm be the configuration of the CVM model, and (pm,hd) be the configuration of the underlying physical machine with the hard disk. Assume that (i) the concrete kernel is consistent to the physical machine: $consis(alloc)(k, pm)$, (ii) the relation $\mathcal{B}(cvm.up, pm, hd)$ holds, (iii) the preconditions $pre_f(k, pm)$ to the primitive are satisfied, (iv) the kernel invariant $inv(cvm, k, pm, hd)$ holds, and (v) the kernel static properties $pro(cvm.ak, c)$ are satisfied, then there exists a number of steps T of the physical machine with the hard disk, s.t. $(pm', hd') = \delta^T(pm, hd)$ after which (i) the CVM model executes the primitive and the relation $\mathcal{B}(f_S(cvm.up, p_f), pm', hd')$ still holds, (ii) the concrete kernel executes the primitive and is still consistent to the physical machine: $\exists k', alloc' \ : \ consis(alloc')(k', pm') \wedge k'.pr = r$, and (iii) the kernel invariant is preserved: $inv(cvm', k', pm', hd')$.

5 Case Study: Copying Data Between Processes

As an application of the developed approach we show how we establish the correctness of the *copy* primitive [4]. It is intended to copy n words from a process pid_1 at address a_1 to a process pid_2 at address a_2. In the context of an operating system it is widely used to implement process management routines, as well as IPC. The correctness is justified by the instances of Theorems 4.1 and 4.2, where $f = copy$, and $p_f = p_{copy} = pid_1, pid_2, a_1, a_2, n$.

5.1 Algorithm

Let $copy_{asm}(pa_1, pa_2, s)$ be an assembler fragment that copies s words in the memory from a physical address pa_1 to pa_2. The algorithm behind the *copy* primitive is as follows. In a loop until n words are processed, we compute the size s of portions to be copied respecting the page borders of both processes. The crucial observation is that both pages, from and to which we copy, must be present in the physical memory. This is achieved by two consecutive calls to the page fault handler which compute physical addresses $pa_1 = pma(pid_1, a_1)$ and $pa_2 = pma(pid_2, a_2)$ and guarantee that both pages containing pa_1 and pa_2 reside in the main memory. We proceed with the copying by executing $copy_{asm}(pa_1, pa_2, s)$. The idea is depicted in Fig. 3.

[4] Implementation and Isabelle/HOL theories containing proofs are available from www.verisoft.de.

5.2 Specification

The specification of the primitive has to reflect the changes on (i) the user processes *cvm.up* of the model, (ii) the concrete kernel *k*, and (iii) the underlying physical machine *pm*. Of course we do not specify just the same conditions in terms of each modified machine because they are interconnected through the simulation relations. We define the desired result on a sufficient level of abstraction. Nevertheless, there is a number of necessary properties, mostly of technical nature, that could be expressed only in terms of the implementation machines *k* and *pm*.

Effects on the Model

Let for a memory m an access to d consecutive cells starting at address a is abbreviated as $m_d[a]=m[a+d-1]\circ\ldots\circ m[a]$. The effect of the primitive execution on the model is given by the function $copy_S(cvm.up, pid_1, pid_2, a_1, a_2, n) = cvm'.up$ which updates the memory of the user process pid_2, i.e., the virtual machine $cvm.up[pid_2]$:

$$cvm'.up[i].m_n[a] = \begin{cases} cvm.up[pid_1].m_n[a_1] & \text{if } i=pid_2 \ \wedge\ a=a_2 \\ cvm.up[i].m_n[a] & \text{otherwise} \end{cases}.$$

The result of $copy_S$ is welldefined only if the preconditions $pre_{copy_S}(cvm.up, pid_1, pid_2, a_1, a_2, n)$ are satisfied. Otherwise, the same trick as with C0 machines is used. The model state space C_{cvm} is extended with a single error state which signals, in particular, that the preconditions to a primitive are not justified. The validity requirements over a model run prevent error states. The predicate pre_{copy_S} encodes formally the following: (i) the amount to be copied is reasonable: $n > 0$, (ii) we copy between different processes: $pid_1 \neq pid_2$, (iii) since memories of virtual machines are word-addressable, the addresses a_1 and a_2 are divisible by 4, (iv) process identifiers pid_1 and pid_2 lie in the interval $[1, N)$, (v) virtual machines $vm_1 = cvm.up[pid_1]$ resp. $vm_2 = cvm.up[pid_2]$ have amount of virtual memory storing resp. sufficient to store the desired portion, i.e., $a_x/4 + n < vm_x.ptl \cdot P$, $x \in \{1, 2\}$.

Effects on the Implementation

The intended modifications of the physical machine pm, on top of which the concrete kernel k runs are defined by the postcondition $post_{copy}(k', pm')$. First, it claims the value of the result variable of the call. Next, it describes the changes in the physical memory of the updated machine pm'. Recall that a virtual address va of a process pid is translated to the physical one by means of the function $pma(pid, va)$. Then, the step-by-step changes over the physical memory are:

$$pm'.m_s[a] = \begin{cases} pm.m_s[pma(pid_1, a_1)] & \text{if } a=pma(pid_2, a_2) \\ pm.m_s[a] & \text{otherwise} \end{cases}.$$

The result is obtained only if the precondition is satisfied. The predicate $pre_{copy}(k)$ defines the indispensable demands to the C0 implementation. Basically, it speaks about parameters and results of the call and demands that: (i) the parameters $va(k, i)$, $0 \leq i < 5$ of the primitive are welltyped, evaluate without errors, storing valid values, and (ii) the result variable of the call is present in $top(k)$. We do not have any special demands to the physical machine before the call to *copy*. Nevertheless, it is worth to mention the preconditions $pre_{copy_{asm}}(pm)$ to the assembler portion $copy_{asm}(pa_1, pa_2, s)$ of the function. They are discharged when we verify the C0 prefix of the function and perform the semantics switch. The requirements comprise, among others, the following conditions: (i) $s > 0$, (ii) the addresses are divisible by 4 and bounded by the total amount of physical memory $pa_x/4 + s < TPP \cdot P$, $x \in \{1, 2\}$, and (iii) the memory regions between which we copy do not overlap: $[pa_1/4 : pa_1/4 + s) \not\cap [pa_2/4 : pa_2/4 + s)$.

5.3 Proving Correctness

In order to prove Theorem 4.1 we show the following separate lemmas for the correctness of: (i) the C0 code inside the loop up to the first call to the page fault handler, (ii) the two consecutive calls to the page fault handler, (iii) the inline assembler portion, (iv) the remaining C0 part of the loop body, and (v) the whole loop, which makes use of the four previous lemmas (cf. Fig. 3). We motivate such modularization as follows. We create separate lemmas for the items i, iii, and iv because they describe code portions in alternating semantics. The case ii is treated specially as it speaks about the properties derived from the page fault handler specification. They are used for lemma iii, but either invisible or not important in the other lemmas.

The proof proceeds by applying the respective semantics to the code statement by statement. In order to prove the loop, i.e., lemma v, we formally specify the ranking function $r(n, a_1, a_2) = i$, s.t. $r(0, a_1, a_2) = 0$ which counts the number i of the remaining loop iterations. The lemma has an inductive fashion with the step of the form $P(k, pm, i) \Longrightarrow P(k', pm', i - 1)$. Hence, its proof justifies the loop termination.

Lemma ii argues that after two consecutive calls to the page fault handler for the computation of physical memory addresses pa_1 and pa_2 both pages containing these addresses reside in the physical memory. A design requirement not to swap out the page that was swapped in during the previous run is respected by the page fault handler. However, this property *cannot* be stated directly in its specification. It is expressed in the specification of two successive calls to the page fault handler. We deal with the problem as follows. Let $page(pa)$ denote the physical page corresponding to the physical address pa. We have a look at the eviction algorithm stated in the page fault handlers formal specification and find out the property $ev(p)$ a page p must obey in order to be evicted during the next call to the handler. The first call to the handler yields the translated physical memory address pa_1. The swapped in page is then $page(pa_1)$. We prove that $\neg ev(page(pa_1))$ holds after the first call the page fault handler. Since there is no code between the two handler calls, this property holds in the precondition to the second call for free.

Lemma iii states that the physical addresses from (pa_1) and to (pa_2) which we copy are associated with processes pid_1 resp. pid_2 and do not belong to the kernel range. This ensures kernel safety—the assembler portion does not destroy kernel by modifying its data structures—and security, for instance we do not disclose page tables by coping them to a user process. Next, lemma states that pa_1 and pa_2 do not belong to the hard disk port address range, guaranteeing that no swap data is disclosed or modified . The proof exploits lemma ii because most of these properties are inferred from the page fault handler validity (cf. Sect. 4.2.2).

The proof of Theorem 4.2 uses the functional correctness established above. The essential proof goals are the implication from the postcondition $post_{copy}(k', pm')$ to (i) the $\mathcal{B}$-relation $\mathcal{B}(f_S(cvm.up, p_f), pm', hd')$, and (ii) the kernel invariant $inv(cvm', k', pm', hd')$. The proof of the former necessarily exploits the fact that the $\mathcal{B}$-relation is preserved under the page fault handler. Since the relation is not affected by the C0 parts of the primitive—we do not write to PCBs and cannot modify the memory region beyond C0 variables—it remains to show that the $\mathcal{B}$-relation is not destroyed by the assembler fragment. Here, since we transfer data between pages residing in the physical memory formally described in lemma iii and perform the respective memory updates on the model *cvm* and on the physical machine *pm* as stated in Sect. 5.2, the relation follows. During the kernel invariant proof we examine $post_{copy}(k', pm')$ in order to determine which C0 variables and memory parts are changed by the primitive. From this the invariant is concluded.

6 Summing Up

We have shown how the problem of formal correctness of microkernel primitives is solved exploiting a novel approach to verification of C with inline assembler programs. We conclude by giving statistics and directions for further research.

Complexity

The implementation of 16 primitives consumes about 600 lines of code. We have verified 3 primitives so far—*copy*, *get_vm_word*, and *set_vm_word*—which are implemented with 100 lines. The functional correctness of the exemplary primitive (Theorem 4.1) is established in about 2K proof steps, where proofs in C0 and in assembler semantics are related as 2:1 [5] . The integration of these results into the kernel context (Theorem 4.2) requires about 5K commands using general kernel lemmas of technical nature proven in 5K steps.

Further Work: Gaining from Automated Verification

Although we used mostly interactive verification techniques there is a room for the automation. One can gain from methods of automated verification while proving the functional correctness of source code. We used the ML code generation mechanism for the proof of the microkernel source code wellformdness properties required by the C0 compiler correctness theorem. That saved about 1K proof commands. Next possible candidate for proof automation are assembler portions. Due to the relatively simple finite memory model it might be possible to obtain the values of desired memory cells by means of model checking. In order to ease the C0

[5] Approximately the same proportion holds for the respective parts of the implementation.

part verification, one can think of a Hoare logic environment for the C0 small step semantics which will automatically generate verification conditions to be proven.

References

[1] Alkassar, E., N. Schirmer and A. Starostin, *Formal pervasive verification of a paging mechanism*, in: *14th Intl Conference, TACAS 2008, Proceedings (to appear)*, LNCS (2008).

[2] Beyer, S., 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** (2006), pp. 411–430.

[3] Gargano, M., M. Hillebrand, D. Leinenbach and W. Paul, *On the correctness of operating system kernels*, in: *Proc. of the 18th Intl Conference on TPHOLs*, 2005, pp. 1–16.

[4] Heiser, G., K. Elphinstone, I. Kuz, G. Klein and S. M. Petters, *Towards trustworthy computing systems: Taking microkernels to the next level*, SIGOPS Oper. Syst. Rev. **41** (2007), pp. 3–11.

[5] Hillebrand, M., T. In der Rieden and W. Paul, *Dealing with I/O devices in the context of pervasive system verification*, in: *ICCD '05*, 2005, pp. 309–316.

[6] In der Rieden, T. and A. Tsyban, *Cvm - a verified framework for microkernel programmers*, in: *3rd intl Workshop on Systems Software Verification (SSV08), to appear* (2008).

[7] Leinenbach, D., "Compiler Verification in the Context of Pervasive System Verification," Ph.D. thesis, Saarland University, Computer Science Department (2007).

[8] Moore, J., *A grand challenge proposal for formal methods: A verified stack*, in: *Proc. of the 10th Anniversary Colloquium of UNU/IIST*, 2003, pp. 161–172.

[9] Ni, Z., D. Yu and Z. Shao, *Using xcap to certify realistic systems code: Machine context management*, in: *Proc. 20th TPHOLs* (2007), pp. 189–206.

[10] Paul, W., *Towards a worldwide verification technology*, in: *Proc. of the Conference on Verified Software: Theories, Tools, Experiments*, Zürich, Switzerland, 2005.

[11] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technische Universität München (2006).

[12] Tews, H., *Micro hypervisor verification: possible approaches and relevant properties*, electronic version is available at http://www.cs.ru.nl/H.Tews/nluug-07/hyperveri.pdf (2007).