Formal Pervasive Verification of a Paging Mechanism

Eyad Alkassar[⋆], Norbert Schirmer [⋆⋆], and Artem Starostin [⋆⋆⋆]

Computer Science Department - Saarland University
{eyad,nschirmer,starostin}@wjpserver.cs.uni-sb.de

Abstract. Memory virtualization by means of demand paging is a crucial component of every modern operating system. The formal verification is challenging since reasoning about the page fault handler has to cover two concurrent computational sources: the processor and the hard disk. We accurately model the interleaved executions of devices and the page fault handler, which is written in a high-level programming language with inline assembler portions. We describe how to combine results from sequential Hoare logic style reasoning about the page fault handler on the low-level concurrent machine model. To the best of our knowledge this is the first example of pervasive formal verification of software communicating with devices.

1 Introduction

With a comparably small code base of only some thousand lines of code, and implementing important safety and security abstractions as process isolation, microkernels seem to offer themselves as perfect candidates for a feasible approach to formal verification. The most challenging part in microkernel verification is memory virtualization, i.e., to ensure that each user process has the notion of an own, large and isolated memory. User processes access memory by virtual addresses, which are then translated to physical ones. Modern computer systems implement virtual memory by means of paging: small consecutive chunks of data, called pages, are either stored in a fast but small physical memory or in a large but slower auxiliary memory (usually a hard disk), called swap memory. The page table, a data structure both accessed by the processor and by software, maintains whether a page is in the swap or the physical memory. Whenever the process accesses a page located in the swap memory, either by a store/load instruction or by an instruction fetch, the processor signals a page fault interrupt. On the hardware side, the memory management unit (MMU) triggers the interrupt and translates from virtual to physical page addresses. On the software side, the *page fault handler* reacts to a page fault interrupt by moving the requested page to the physical memory. In case the physical memory is full, some other page is swapped out (cf. Fig. 1).

The aim of the Verisoft project[1] is a pervasive formal correctness result. The grand challenge is to integrate various levels of abstraction and computational models. The

[⋆] Work was supported by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems'.

[⋆⋆] Work was supported by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project.

[⋆⋆⋆] Work was supported by the International Max Planck Research School for Computer Science.

[1] http://www.verisoft.de

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 109–123, 2008.

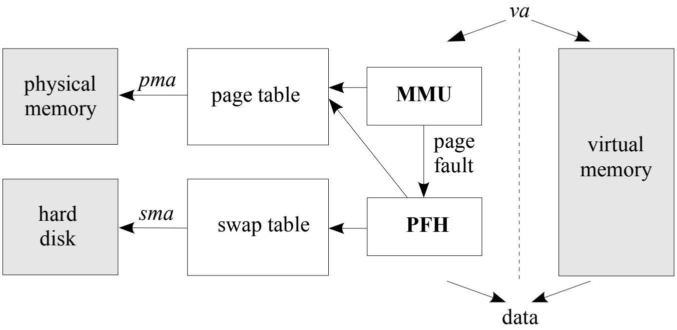

Fig. 1. Concept of paging

physical machine on the lower end, which comprises the concurrent computation of at least the processor and the devices, in the middle, portions of assembler code to implement the device drivers, and on the upper end a high-level sequential programming language, in our case C0, a subset of C. The decisive contribution of this paper is the integration of devices into a pervasive system verification methodology. This comprises dealing with interleaved I/O devices and integration of inline assembler code even in the high-level Hoare logics. All computational models and abstraction layers and almost all proofs are mechanized in the theorem prover Isabelle/HOL, which gives us the highest possible assurance that all parts fit together. At submission of the paper not finished proofs include the read case of the driver.

Related Work. Hillebrand [7] presents paper and pencil formalisations and proofs for memory virtualisation. First attempts to use theorem provers to specify and even prove correct operating systems were made as early as the seventies in PSOS [11] and UCLA Secure Unix [16]. However a missing or to a large extend underdeveloped tool environment made mechanized verification futile. With the CLI stack [2], a new pioneering approach for pervasive system verification was undertaken.

Most notably the simple kernel KIT was developed and its machine code implementation was proven. Compared to modern kernels KIT was very limited, in particular it lacked the interaction with devices. The project L4.verified [6] focuses on the verification of an efficient microkernel, rather than on formal pervasiveness, as no compiler correctness or an accurate device interaction is considered. The microkernel is implemented in a larger subset of C, including pointer arithmetic and an explicit low-level memory model [15]. However with inline assembler code we gain an even more expressive semantics as machine registers become visible if necessary. So far only exemplary portions of kernel code were reported to be verified, the virtual memory subsystem uses no demand paging [14]. For code verification L4.verified relies on Verisoft's Hoare environment [13]. In the FLINT project, an assembly code verification framework is developed and code for context switching on a x86 architecture is formally proven [12]. Although a verification logic for assembler code is presented, no integration of results into high-level programming languages is undertaken. The VFiasco project [9] aims at the verification of the microkernel Fiasco implemented in a subset of C++. Code verification is performed in a embedding of C++ in PVS and there is no attempt to map the results down to the machine level.

Overview. Sect. 2 elaborates on the virtualization problem, gives an overview of our general approach and states the required page fault handler correctness property. In Sect. 3 a page fault handler implementation, specification and its code verification is presented. In Sect. 4 we integrate the code verification into the Verisoft system stack to obtain the desired correctness result. Finally we conclude in Sect. 5.

2 Virtual Memory Simulation Problem

One of the most challenging verification objectives of Verisoft is to prove that the physical machine correctly implements memory virtualization towards user processes: the physical memory and the swap space of the hard disk are organized by the page fault handler to provide separate uniform linear memories towards user processes. This is expressed as a simulation theorem between a physical machine and a virtual machine model and is described in this section. The other sections are concerned with its proof.

Most crucially the correctness of the simulation theorem depends on the correctness of the page fault handler. This proof ties together various key results of the Verisoft system stack. Besides the physical machine this also includes semantics for C0 and a Hoare logic. The physical machine model is quite fine grained and executes instructions and devices concurrently, whereas the page fault handler is basically a high-level sequential C0 program. We avoid conducting the whole proof on the low-level of the physical machine, by proving the page fault handler correct in a Hoare logic for C0. The soundness theorem of the Hoare logic [13] composed with the correctness theorem of the C0 compiler [10] allows to transfer the page fault handler correctness down to the physical machine. However, we have to consider one peculiarity of the page fault handler (and low-level systems code in general): there are small portions of inline assembler code that break the high-level C0 semantics. In case of the page fault handler this is the device driver code for communication with the hard disk. We encapsulate these pieces of inline assembler code into so called 'XCall's. On the level of the Hoare logic these are represented as atomic state updates (cf. Sect. 3.1) on an extended state that comprises both the C0 memory and the current configuration of the hard drive. We prove that the assembler implementation of the XCall adheres to this abstraction (cf. Sect. 4.2).

Another major step is to bridge the gap from the sequential C0 model to the concurrent execution of the processor and the devices on the physical machine. This is achieved by means of a reordering theorem (cf. Sect. 4.2). The memory virtualization theorem is a property that has to hold for all possible interleavings. The reordering theorem allows us to focus only on those execution traces where the relevant actions of the device driver and the device happen consecutively.

2.1 Basic Definitions

We denote the set of boolean values by $\mathbb{B}$ and set of natural numbers including zero by $\mathbb{N}$. For any natural x we denote the set of natural numbers less then x by $\mathbb{N}_x$. We denote the unbounded abstract list with elements of type T by T^* and the list of length n by T^n. The length of a list x is denoted by $|x|$, its element access by $x[i]$, and the tail, e.g., the part of the list without the first element, by $tl(x)$. The operator $\langle x \rangle$ yields for a bit string $x \in \mathbb{B}^*$ the natural number represented by x. The fragment of a bit list from a position a to b is denoted by $x[b:a]$. For a record x the set of all its possible configurations is defined by C_x. A memory m is modeled as a mapping from addresses a to byte values $m[a]$. An access to d consecutive memory cells starting at address a is abbreviated as $m_d[a] = m[a+d-1], \ldots, m[a]$.

We deal with an abstract model of computation, where N user processes are virtual machines that run on a single physical machine. The memories of machines are logically organized in pages of size $P = 4K$ bytes. For a virtual address $va \in \mathbb{B}^{32}$ we define by $px(va) = va[31:12]$ and $bx(va) = va[11:0]$ its page and byte indexes, respectively. Represented as natural number we get $px(va) = va/P$ and $bx(va) = va \mod P$.

2.2 Physical Machine Specification

The physical machine is the sequential programming model of the VAMP hardware [3] as seen by a system software programmer. It is parameterized by (i) the set $SAP \subseteq \mathbb{B}^5$ of special purpose register addresses visible to physical machines, and (ii) the number TPP of total physical memory pages which defines the set $PMA = \{a \mid 0 \leq \langle a \rangle < TPP \cdot P\} \subseteq \mathbb{B}^{32}$ of accessible physical memory addresses. 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 register file $pm.gpr \in \mathbb{B}^5 \mapsto \mathbb{B}^{32}$, and the special purpose register file $pm.spr \in SAP \mapsto \mathbb{B}^{32}$, and (iii) the byte addressable physical memory $pm.m \in PMA \mapsto \mathbb{B}^8$. We demand SAP to contain the following addresses: (i) *mode*, for the mode register, and (ii) *pto* and *ptl*, for the page table origin resp. length registers whose values are measured in pages. For any address $a \in SAP$ we will abbreviate $pm.spr[a] = pm.a$. A physical machine is running in *system mode* if $pm.mode = 0^{32}$ and in *user mode* if $pm.mode = 0^{31}1$.

Address Translation and Page Faults. 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 $pma(pm, va)$ or generates a page fault.

The physical memory address is computed as follows. We interpret the memory region $pm.m_{pm.ptl \cdot P}[pm.pto \cdot P]$ as the current page table. Let $ptea(pm, va) = pm.pto \cdot P + 4 \cdot px(va)$ be the page table entry address for virtual address va and $pte(pm, va) = pm.m_4[ptea(pm, va)]$ be its page table entry. The page table entry is composed of three components, the physical page index $ppx(pm, va) = pte(pm, va)[31:12]$, the valid bit $v(pm, va) = pte(pm, va)[11]$, and the protected bit $p(pm, va) = pte(pm, va)[10]$. Concatenation of the physical page index and the byte index yields the physical memory address $pma(pm, va) = ppx(pm, va) \circ bx(va)$.

In order to define page faults, let $w \in \mathbb{B}$ be active on write operations. The page fault $pf(pm, va, w)$ is raised if (i) the valid bit $v(pm, va)$ is not set, or (ii) the write flag w and the protected bit $p(pm, va)$ are active.

Semantics. The semantics of a physical machine is formally given by the transition function $\delta_{pm}(pm) = pm'$ yielding the next state configuration. If no page fault occurs the effects of a transition are defined by the underlying instruction set architecture (ISA). In case $pf(pm, va, w)$ the program counters are set to the start address of the page fault handler. We switch to system mode and the execution of the handler is triggered. After its termination the mode is changed back and the user computation resumes.

2.3 Devices

From the viewpoint of the operating system the hard disk is a device . Before describing the hard disk model, we sketch our general framework for memory-mapped devices (we do not consider DMA here). A device x is a finite transition system which communicates with (i) an unspecified external environment, and (ii) the processor. Examples for input and output from and to the environment are incoming key press events and outgoing network packages. We denote inputs from the environment with $eifi_x$ and outputs with $eifo_x$. The processor reads and writes from and to a device by load- and store-word operations to specific address regions. Both operations are signaled by an output function ω of the processor: $\omega(pm) = mifi$. In case of a read operation, the device returns requested data in form of an output called $mifo$. The transition of a device of type C_x is then given by: $\delta_{devs}(x, eifi_x, mifi) = (x', mifo, eifo_x)$.

For all modeled device types DT and a set of device identifiers DI, the configuration $pmd = (pm, devs : DI \rightarrow DT)$ describes the state of the processor and of all devices. The processor and devices are executed in an interleaved way. An oracle, called *execution sequence* (seq) determines for a given step number i whether the processor, i.e., $seq(i) = Proc$ or some device d makes a step, in which case the sequence also provides the input from the environment: $seq(i) = (d, eifi_{xd})$. The function δ_{pmd} describes the execution of the overall system. It takes as input the combined state of the processor and of all devices, a step number and an execution sequence. A detailed description of the device framework can be found in [1]. Next we instantiate one device with the hard disk.

Hard Disk Description. We model a hard disk based on the ATA/ATAPI protocol. Hard disks are parameterized over the number of sectors S. Each sector has a size of 128 words. We assume that the hard disk is large enough to store the total virtual page space of all user processes. The processor can issue read or write commands to a range of sectors, by writing the start address and the count of sectors to a special port. Each sector is then read/written word by word from/to a sector buffer. After a complete sector is written or read to the sector buffer, the hard disk needs some time to transfer data to the sector memory. This amount of time is modeled as non-determinism by an oracle input from the external environment, indicating the end of the transfer. In this case the input $eifi_{hd}$ is set to one. The hard disk can be run either in interrupt or polling mode. We chose the second type.

In the following we only need the sector memory of the hard disk. Its domain ranges over $S \cdot 128$ words: $hd = (sm : \mathbb{N}_{S \cdot 128} \mapsto \mathbb{N}_{256}, \dots)$. A hard disk is necessarily an item of the device system $pmd.devs$. We abbreviate an access to the hard disk of a physical machine with devices as $pmd.hd$. A detailed description of the hard disk, its transitions and a simple driver can be found in [8].

2.4 Virtual Machine Specification

Virtual machines are the hardware model visible for user processes. They give an user the 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 total virtual memory pages which defines the set of virtual memory addresses $VMA = \{a \mid 0 \leq \langle a \rangle < TVP \cdot P\} \subseteq \mathbb{B}^{32}$, and (ii) the set $SAV \subseteq SAP \setminus \{mode, pto, ptl\}$ of special purpose register addresses visible to virtual machines. Their configuration, formally, is a record $vm = (pc, dpc, gpr, spr, m)$ where only the $vm.spr \in SAV \mapsto \mathbb{B}^{32}$ and $vm.m \in VMA \mapsto \mathbb{B}^{8}$ differ from the physical machines. The semantics is completely specified by the ISA.

2.5 Simulation Relation

A physical machine maintaining a page fault handler with a hard disk can simulate virtual machines. The simulation relation, called the $\mathscr{B}$-relation, specifies a (pseudo-) parallel computation of N user processes $up \in C_{vm}^{N}$ modeled as virtual machines on one system *pmd* composed out of a physical machine, a hard disk, and other devices. The computation proceeds as follows.

The physical machine maintains a variable *cp* designating which of the user processes is meant to make a step. Unless a page fault occurs, the process $up[cp]$ is updated according to the semantics of virtual machines. Otherwise, the physical machine invokes the page fault handler. After its execution, the user process continues the computation. An appropriate page fault handler obeys the following rules: (i) it maintains the list of *N process control blocks (PCB)* (described below), which permanently reside in the memory of the physical machine, (ii) it is able to access page tables of processes which lie consecutively in the physical memory, (iii) it has a data structure, called the *swap table* which maps virtual addresses to swap page indexes. A swap memory address $sma(pm, a)$ is computed via an access to such a table.

Process control blocks implement the user processes. They contain fields for storing the content of *gpr* and *spr* register files of all processes. When a user process currently being run on the physical machine is interrupted by a page fault, the content of the registers is copied into the PCBs *before* the execution of the page fault handler. Accordingly, after the handler terminates the interrupted user process is restored by copying the content of appropriate PCB fields back to the registers of the physical machine.

Now we define the $\mathscr{B}$-relation. First, we must reconstruct virtual machines from the contexts stored in PCBs. The function $virt(pid, pmd) = vm$ yields the virtual machine for process *pid* by taking the register values from the corresponding PCB fields. The memory component of the built virtual machine is constructed out of physical memory and the data on the hard disk depending where a certain memory page lies:

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

Then, the $\mathscr{B}$-relation is: $\mathscr{B}(pmd, up) = \forall pid \in \mathbb{N}_N \ : \ virt(pid, pmd) = up[pid]$.

There is a small number of additional correctness relations omitted due to the lack of space. The reader should refer to [5] for them.

Proving the correctness of memory virtualization is a one-to-n-step simulation between the virtual and the physical machine. One has to show that each step of a virtual machine, can be simulated by n steps of the physical machine, while the $\mathscr{B}$-relation is preserved. The only interesting case in the proof is the occurrence of a page fault during the execution of a load or store instruction. In all other cases, the semantics of the virtual and the physical machines almost coincide. In the following we describe the crucial part of the proof: the page fault handler execution leads to a non-page faulting configuration maintaining the $\mathscr{B}$-relation.

3 Page Fault Handler Implementation and Code Verification

3.1 Extended Hoare Logic

Our page fault handler is implemented in a high-level programming language with small portions of inline assembler code for communication with the hard disk and to access portions of memory that are not mapped to program variables (e.g., memory of user processes). As programming language we use C0, which has been developed for and is extensively used within the Verisoft project. In short C0 is Pascal with C syntax, i.e., its type-system is sound and it supports references but no pointer-arithmetic. Syntax and semantics of C0 are fully formalized in Isabelle/HOL. Moreover, a compiler is implemented and formally verified [10].

We use Hoare logic as an effective means of C0 program verification. Unfortunately the inline assembler portions, that make hardware details visible towards the programming language, break the abstractions C0 provides. We deal with the low-level inline assembler parts, without breaking the Hoare logic abstraction for C0, by encapsulated them into an atomic step on an extended state, a so called *XCall*. The extended state can only be modified via a XCall.

We apply an instance of Schirmer's Hoare logic environment [13] implemented in Isabelle/HOL. He defines a set of Hoare rules, for a generic programming language *Simpl*, and formally proves the soundness of this logic. Hoare rules describe a triple $\{P\}S\{Q\}$, where precondition P defines the set of valid initial states of the C0 variables, S is the statement to execute and postcondition Q is guaranteed to hold for the states after execution of S. We prove total correctness and hence termination is guaranteed.

Additionally to the embedding of C0 into Simpl, we introduce some special treatment for XCalls and refer to the resulting system as *extended Hoare logic*. First we have to deal with the extended state space, i.e., the physical memory and the swap. In Simpl we treat them analogous to C0 variables, with the only difference that they are not restricted to C0 types. It is not necessary to introduce a new Hoare rule into Simpl to handle XCalls. Instead we can use the '*Basic*' construct of Simpl, a general assignment which can deal with an arbitrary state update function f: $\{f(s) \in Q\}$ *Basic* f $\{Q\}$. On the level of Simpl every XCall is modelled as such a state update, representing the abstract semantics of the XCall.

The specification of the page fault handler and its operations, and data types in general, are not directly formulated on the level of the extended C0 state. Instead the C0 state is lifted to a more abstract model. For example a linked pointer structure may be mapped to a HOL list. An abstraction relation *abs* relates the model *a* and the extended C0 state *xc*. The pre- and postconditions of Hoare triples then typically express that the abstraction relation is preserved by the abstract operation and the corresponding C0 implementation. When the operation on the abstract model transfers *a* to a' then the extended C0 state has to make the analogous transition. Hence we prove simulation of the abstraction and the C0 program by the following specification scheme: $\{abs(a,xc)\}S\{abs(a',xc')\}$.

In Sect. 3.2 we describe the C0 implementation of the page fault handler using XCalls. Then we continue in Sect. 3.3 with the specification of the page fault handler by an abstract model. The aforementioned Hoare logic verification ensures the simulation of the implementation and the abstract model.

3.2 Implementation

Our page fault handler implementation [4] maintains several global data structures to manage the physical and the swap memories. These data structures permanently reside in the memory of a physical machine. They are used to support mechanisms of virtual memory de- and allocation, and the page replacement strategy. The necessary data structures comprise: (i) the process control blocks, (ii) the page and swap tables, and (iii) *active* and *free* lists managing allocated and free user memory pages, respectively.

On the software level we distinguish the following page faults: (i) an *invalid access* occurring when a desired page is not present in the physical memory, and (ii) the *zero protection* page fault which signals a write access to a newly allocated page.

On the reset signal a page fault handler initialization code is executed. It brings the data structures to a state where all the user physical pages are free, and page resp. swap tables are filled with zeros. The page table lengths of all user processes are nulled, and their origins are uniformly distributed inside the page table space.

When a memory page is allocated for a virtual machine it must be filled with zeros. In order to avoid heavy swapping and zero-copying at a particular page index, we optimize the allocation process by making all of the newly allocated pages point to the *zero filled page* residing at page address ad_{zfp}. This page is always protected. Whenever one reads from such a page a zero content is provided. At a write attempt to such a page a zero protection page fault is raised. Thus, an allocation leaves the *active* and *free* lists unchanged, possibly modifying the PCBs and the page table space in case a movement of origins is needed. On the memory free, the descriptors of the released pages are moved back to the *free* list, and the corresponding entries in the page tables are invalidated.

On a page fault the handling routine is called. The free list is examined in order to find out whether any unused page resides in the physical memory and could be given to a page faulting process. If not, a page from an active list is evicted. An obtained vacant page is then either filled with the desired data loaded from the disk, or with zeros depending on the kind of page fault. The page table entry of an evicted page is invalidated while the valid bit of a loaded page is set. We use the FIFO-eviction strategy, which guarantees that the page swapped in during the previous call to the handler will not be swapped out during the current call. This property is crucial for liveness of the page fault handler since a single instruction can cause up to two page faults on the physical machine — one during the fetch phase, the other during a load/store operation.

Programming Model and Extended State. The semantics of C0 is defined as a small step transition system on configurations c. In small step semantics, C0 configurations are records $c = (pr, s)$ where $c.pr$ is the program rest and $c.s$ is the 'state'. The page fault handler manipulates and hence must have access to the following components:

- Program variables as the *free*, and *active* lists, maintained only by the page fault handler. These are ordinary C0 variables.
- The page tables which are used by the hardware for address translation. We simply map them to an ordinary array variable in C0, where we solely have to ensure that the allocation address of this array coincides with the page table origin used by the physical machine during address translation. All the C0 data structures together with the handler code consume *TSP* total system pages of physical memory.
- Physical memory of the machine running the handler. The page fault handler transfers memory pages from the hard disk to the non-system region of physical memory which consists of $TUP = TPP - TSP$ total user pages of physical memory, and hence it must be able to manipulate the region $pm.m_{TUP \cdot P}[TSP \cdot P]$. As in C0 the memory is not explicitly visible (e.g., through pointer arithmetic) we employ the extended state to manipulate physical memory.
- The swap memory of the hard disk. An elementary device driver which swaps pages from memory to the hard disk is an integral part of the page fault handler. Similar to the virtual memory, the hard disk is out of the scope of the pure C0 machine, and is handled by the extended state.

Access to the physical memory and the elementary device driver of the hard disk are both implemented as *inline assembler code* in C0. As detailed in Sect. 3.1 we encapsulate inline code by atomic primitives, the XCalls. We augment the C0 small step semantics to handle XCalls which results in a new transition system on extended configurations $xc = (c, mem \in \mathbb{N}_{TUP} \mapsto \mathbb{N}_{256}^{P}, swap \in \mathbb{N}_{TVP \cdot N} \mapsto \mathbb{N}_{256}^{P})$.

The semantics δ_{xc} of the new machine executes the small step transition function of the C0 machine in case the head of the program rest is an ordinary C0 statement. Otherwise the effects of the primitives are applied, i.e., in case the next statement is a read command: $xc.c.pr = readPage(xc, ad_{mem}, ad_{swap})$ we copy a page from the swap to the virtual memory: $xc'.mem(ad_{mem}) = xc.swap(ad_{swap})$. Whereas in case of a write primitive $xc.c.pr = writePage(xc, ad_{swap}, ad_{mem})$ we copy a page from physical memory to the swap: $xc'.swap(ad_{swap}) = xc.mem(ad_{mem})$.

The implementations of these primitives, mainly the elementary drivers, are verified separately against their specification. Note, that at this point we even abstracted interleaved device execution to one atomic step. In Sect. 4.2 we justify this abstraction by stating the correctness of our driver implementation.

3.3 Specification

Abstract Page Fault Handler. An abstract page fault handler is a high-level concept of: (i) data structures from the implementation, (ii) physical memory of the machine running a page fault handler, and (iii) hard disk of this machine. Before the formal specification of the page fault handler two auxiliary concepts are formalized. They are page descriptors and translation tables.

Page Descriptors. A page descriptor is a record $pd = (pid, vpx, ppx)$ holding the information about one user page. Its fields are: (i) $pd.pid \in \mathbb{N}_N$, a process identifier which denotes to which virtual machine an associated physical page belongs, (ii) $pd.vpx \in \mathbb{N}_{TVP}$, a virtual page index showing to which virtual page the corresponding physical page belongs, and (iii) $pd.ppx \in \mathbb{N}_{TPP}$, a physical page index which points to the user page in the physical memory.

Translation Table. We abstract the page and the swap tables to the concept of a translation table. It allows us to easily determine the address of a virtual page in physical or swap memories. A *translation table entry* is a record $tte = (ppx, spx, v)$ with $tte.ppx \in \mathbb{N}_{TPP}$, $tte.spx \in \mathbb{N}_{TVP}$ and $tte.v \in \mathbb{B}$. The components are the physical page index, the swap page index, and the valid bit, respectively.

Configuration. The abstracted configuration of a page fault handler is a record $pfh = (active, free, tt, pto, ptl, mem, swap)$. The meaning of the components is: (i) the active list $pfh.active \in C_{pd}^*$ of page descriptors associated with user memory pages that store a virtual page, (ii) the free list $pfh.free \in \mathbb{N}_{TPP}^*$ of unused physical page indexes. We demand $|pfh.active| + |pfh.free| = TUP$ since both lists describe physical memory pages potentially accessible by a user, (iii) the translation table $pfh.tt \in C_{tte}^{TVP}$ is an abstraction of the physical memory region that stores page tables and swap tables. Each entry corresponds exactly to one virtual page, (iv) the vectors of the processes' page table origins $pfh.pto \in \mathbb{N}_{TVP}^N$ and page table lengths $pfh.ptl \in \mathbb{N}_{TVP}^N$, (v) page addressable representations of the non-system part of the physical memory $pfh.mem \in \mathbb{N}_{TUP} \mapsto \mathbb{N}_{256}^P$, and the hard disk content $pfh.swap \in \mathbb{N}_{TVP \cdot N} \mapsto \mathbb{N}_{256}^P$.

The components $pfh.(mem, swap)$ are supposed to be mapped one-to-one to the corresponding components of the implementation configuration xc.

Page Faults Visible to the Software. A page fault handler guarantees that after its call a page associated with a virtual address va and a process pid will be in the physical memory. The translation table entry address for a virtual page index $px(va)$ and a process pid is defined as $ttea(pfh, pid, px(va)) = pfh.pto[pid] \cdot P + px(va)$, and the corresponding translation table entry is $tte(pfh, pid, px(va)) = pfh.tt[ttea(pfh, pid, px(va))]$. We shorten an access to an entry's component $x \in \{ppx, spx, v\}$ as $x(pfh, pid, px(va)) = tte(pfh, pid, px(va)).x$. By $pf(pfh, pid, va) = v(pfh, pid, px(va)) \vee ppx(pfh, pid, px(va)) = ad_{zfp}$ we define the page fault signal. The disjuncts denote the invalid access and the zero protection page faults, respectively.

Obtaining a Non-Page Faulting Configuration. A page fault handling routine invoked in a configuration pfh with $pf(pfh, pid, va)$ steps to a configuration pfh' with $pf(pfh', pid, va)$ according to the following algorithm. First, a *victim* page descriptor vic is selected depending on the length of the free list. Let $E(pfh)$ hold in case the free list in the configuration pfh is empty, i.e., $|pfh.free| = 0$. We set $vic = pfh.active[0]$ if $E(pfh)$, and $vic = (*, *, pfh.free[0])$ otherwise. If the free list is empty the descriptor of the page to be swapped in is removed from the head of the active list, otherwise from the head of the free list, and is appended to the active list. The vpx field of the descriptor is set to the $px(va)$ value. Formally,

$pfh'.active = tl(pfh.active) \circ (pid, px(va), vic.ppx)$ and $pfh'.free = pfh.free$ if $E(pfh)$, and $pfh'.active = pfh.active \circ (pid, px(va), vic.ppx)$ and $pfh'.free = tl(pfh.free)$ otherwise. Further, in case of the empty free list the victim page is written to the swap memory. Formally, $pfh'.swap = swap\_out(pfh, vic.pid, vic.ppx, vic.vpx)$ if $E(pfh)$, where $swap\_out(pfh, pid, ppx, vpx)$ yields the modified swap component replacing the swap page at address $spx(pfh, pid, vpx)$ by $pfh.mem[ppx]$. The (obtained) free space in the physical memory is either filled with zeros in case of the zero protection page fault or with the page loaded from the swap memory. We set $pfh'.mem = zfp(pfh, vic.ppx)$ if $ppx(pfh, pid, px(va)) = ad_{zfp}$, and $pfh'.mem = swap\_in(pfh, pid, vic.ppx, px(va))$ otherwise. The $swap\_in(pfh, pid, ppx, vpx)$ returns a memory component where a page at address ppx is updated with $pfh.swap[spx(pfh, pid, vpx)]$, and $zfp(pfh, ppx)$ yields a memory where the page ppx is filled with zeros. Finally, the translation table entry of the evicted page is invalidated while the valid bit and the page index of the swapped in page are appropriately set:

$$pfh'.tt[i].(ppx, v) = \begin{cases} (ppx(pfh, vic.pid, vic.vpx), 0) & \text{if } i = ttea(pfh, vic.pid, vic.vpx) \\ (vic.ppx, 1) & \text{if } i = ttea(pfh, pid, px(va)) \\ pfh.tt[i].(ppx, v) & \text{otherwise} \end{cases} .$$

4 Correctness of the Page Fault Handler: Integrating Results

Conceptually, there are two correctness criteria for a page fault handler. Invoked in the configuration pfh with the parameters (pid, va) it must guarantee, first of all, a page fault no longer occurs at the address va. Secondly, it must preserve the $\mathcal{B}$-relation which is established for the first time after the page fault handler initialization code. Both properties follow from the functional correctness of the page fault handler implementation.

Mapping Implementation to Abstraction. In order to state that the handler implementation respects abstraction we define the predicate $map(c, pmd, pfh)$ which is, basically, a conjunction of the three following statements: (i) the variables of the implementation C0 machine c encode the data structures $pfh.(active, free, tt, pto, ptl)$ of the abstraction, (ii) the memory $pmd.pm.m$ of the physical machine starting from the page TSP encodes the abstract memory component $pfh.mem$, and (iii) the hard disk content $pmd.hd.sm$ stores the swap pages $pfh.swap$ of the abstraction. This mapping is established for the first time with the initial configuration of the abstract page fault handler and has to be preserved under each call to the handler.

Mapping C0 to the Physical Machine. Since the overall paging mechanism correctness is stated on the level of the physical machine with devices, we relate C0 configurations to the physical machine states. Given an allocation function $alloc$ mapping variable names to memory locations, we relate a C0 configuration to its physical machine implementation. We use the compiler simulation relation $consis(alloc)(c, pm)$, which relates values of variables and pointers to memory regions and the program to some code region of the physical machine. A further condition is control-consistency, which 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 $c.pr$ of the C0 machine.

Validity of the Abstract Page Fault Handler. We demand a variety of properties to hold over the page fault handler abstraction. These properties reflect the functional correctness and are necessary for the $\mathscr{B}$-relation proof. The predicate $valid(pfh)$ claims, among others, the following: (i) all virtual addresses are translated into physical addresses outside the page fault handler code range, (ii) translation tables do not overlap, (iii) page table origins of user processes are monotonic, and $ttea(pfh,pid,px(va))$ always addresses a value inside the translation table, (iv) the active list describes only valid pages, and all the valid pages are described by the active list, (v) none of the virtual pages of a given process might be stored by two or more active pages, and (vi) all physical page indexes in active and free lists are distinct. Now we state the overall correctness theorem of a paging mechanism.

Theorem 1 (Paging Mechanism Correctness). Let c be the C0 machine calling the handler function: $c.pr = handler(pid,va);r$, and let pfh be the abstract page fault handler configuration. Let up be the user processes, and $pmd = (pm,devs)$ be the physical machine with devices. Assuming that (i) c is simulated by pm: $consis(alloc)(c,pm)$, (ii) the relation $\mathscr{B}(pmd,up)$ holds, (iii) c and pmd encode a valid configuration pfh: $map(c,pmd,pfh) \wedge valid(pfh)$, and (iv) a page fault takes place: $pf(pfh,pid,va)$, then there exists a number of steps T, s.t. $pmd' = (pm',devs') = \delta^T_{pmd}(pmd)$ after which (i) the handler function is executed and the C0 machine remains consistent with the physical one: $\exists c',alloc' \,.\, consis(alloc')(c',pm') \wedge c'.pr = r$, (ii) the relation $\mathscr{B}(pmd',up)$ is preserved, and (iii) c' and pmd' encode a valid non-page faulting configuration pfh': $map(c',pmd',pfh') \wedge valid(pfh') \wedge pf(pfh',pid,va)$.

4.1 Compiler Correctness: From C0 to the Physical Machine

The compiler correctness theorem states that the execution of a given C0 program simulates the execution of the compiled program running on the hardware. It is formally proven correct in Isabelle/HOL by Leinenbach [10].

Theorem 2. $\forall t.\, consis(alloc)(c,pm) \Longrightarrow \exists s,alloc'.\, consis(alloc')(c^t,pm^s)$

4.2 Driver Correctness

Implementation Model. The driver calls are implemented as C0 functions with inline assembler code, where the inline code portion accesses the hard disk and manipulates the physical memory. An assembler instruction list il can be integrated by a special statement $Asm(il)$ into the C0 code. The control-consistency is extended in a straightforward way to the new statement: an instruction list at the head of the program rest maps to a memory region pointed at by the delayed PC of the physical machine: $c.pr = Asm(il);r \wedge consis(alloc)(c,pm) \Longrightarrow pm.m_{4\cdot length(il)}(pm.dpc) = il$

The semantics of inline assembler is defined over the combined configuration of a C0 and a physical machine with device state (c,pmd). We execute C0 transitions as long as no assembler is met. In case of assembler instructions we switch, via the compiler consistency relation, to the physical machine, execute the assembler code and switch back to the C0 machine. For the last step, we only have to state how the C0 variables are

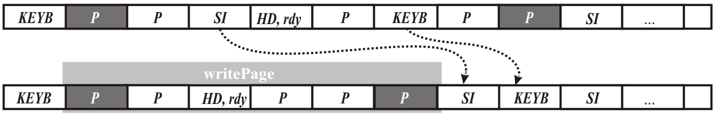

Fig. 2. Reordering of device steps, with hard disk, serial interface and keyboard

manipulated by the assembler code. This combination of assembler and C0 semantics is the driver's implementation model. Out of the two driver calls *readPage* and *writePage* the latter and more complex one was formally verified.

Simulation Relation. This paragraph outlines the correctness of the elementary hard disk driver. First we have to define a simulation relation between our abstract state *xc* and the implementation state (c, pmd). This relation is called $xConsis(xc, c, pmd)$ and it maps: (i) the abstract swap component to the sector memory defined in the hard disk: $xc.swap_{TUP}[0] = pmd.hd.sm_{TUP \cdot K}[0]$, (ii) the abstract memory component to total user pages of physical memory: $xc.mem_{TUP}[0] = pm.m_{TUP \cdot P}[TSP \cdot P]$, (iii) the program rest of *xc* to the C0 program rest by substituting XCalls by their C0 with inline assembler implementation, and (iv) machine *c* to the physical machine *pm*: $consis(alloc)(c, pm)$. Next we can state the simulation theorem, where $pmd^{t,seq}$ denotes the execution of *t* steps under the oracle *seq* of the physical machine with devices:

Theorem 3. $\forall seq.\, xConsis(xc, c, pmd) \implies \exists c',\, t.\, xConsis(\delta_{xc}(xc), c', pmd^{t,seq})$

The most challenging part of the correctness prove, is dealing with the interleaved device execution. We need a programmer's model without unnecessary interleaving, and we have to prove some atomicity of driver execution.

Non-Interference of Devices. All devices take interleaved steps triggered by the environment. However we want to verify Theorem 3, ignoring all devices except the hard disk. In the best case the proof would split in sequential assembler execution and device steps analyzed separately. In principle we want to ensure that: (i) the execution of the driver does not interfere with other devices than the hard disk, and (ii) the execution of the driver can be considered as one atomic step.

These two properties follow from a simple observation of device execution. If the processor is not accessing a device *x* and if device *x* is not triggering an interrupt we can simply swap the execution of the processor and the device without changing the final configuration. A similar lemma holds for swapping steps of two different devices.

We generalized this basic observations to a reordering theorem (see Fig. 2): if all interrupts are disabled and if the processor only accesses some device *x* during executing a given instruction list, we can move all other device steps after the time when the instruction list is executed.

4.3 Proof Sketch of the Paging Mechanism Correctness

With the described methodology we are able to show Theorem 1. In brief, the proof idea is as follows: we show the functional correctness of the page fault handler by reasoning

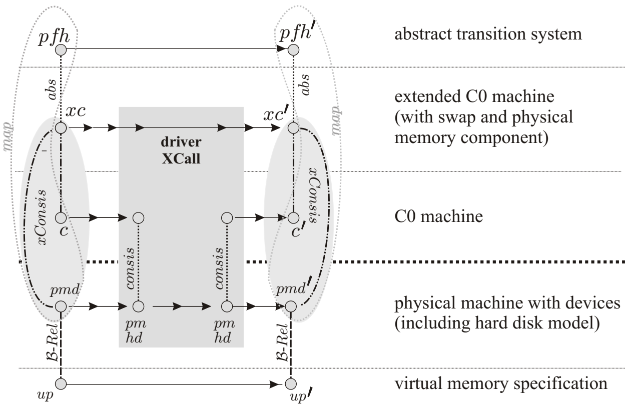

Fig. 3. Page fault handler correctness: putting it all together

in Hoare logic, lift the results down to the level of the physical machine via the soundness theorem of the Hoare logic [13] and Theorems 3 and 2, and infer the $\mathscr{B}$-relation by reasoning about the physical machine memory content. An overview is depicted in Fig. 3. The order of proof steps of the paging mechanism are as follows. 1. Show the validity of the Hoare triple $\{abs(pfh, xc)\}$ $handler(pid, va)$ $\{abs(pfh', xc')\}$. 2. Justify the implication $valid(pfh) \Longrightarrow valid(pfh')$. 3. From Theorem 3 obtain the number of steps T and the implementation machine c' in order to instantiate the existential quantifiers in the conclusion: (a) via *xConsis* and (1) obtain the user part memories' contents $pmd'.pm'.m$ and $pmd'.hd'.sm$ that respect pfh', and (b) via *xConsis* get that c' is mapped to pfh'. 4. Apply Theorem 2 in order to lift (3.c) down to the system part of the physical memory $pmd'.pm'.m_{TSP}[0]$. 5. We have mapped the user and system parts of physical memory and hard disk content to the valid non-pagefaulting configuration pfh'. From $valid(pfh')$ we are able to claim the properties about physical memory content sufficient to show $\mathscr{B}(pmd', up)$, which follows by case splitting on the page fault types.

5 Conclusion

The verification of 500 lines of C0 and 30 lines of assembler code and the reordering theorems took us about 2 person years[2]. Not classical verification problems, as finding invariants, appeared to be the major problem. Rather unexpected and tedious work as 'simple' bitvector operations turned out to be very time consuming. A lot of further effort, not elaborated in this paper, amounts to the integration of models, in particular lifting properties to the overall kernel correctness result, as well as a language stack covering different big- and small step semantics of C0.

This paper not only presents the methodology to deal with pervasive system verification, including inline assembler and driver code accessing interleaved devices, we also verify an important piece of a kernel, running on a real and verified processor.

[2] Isabelle/HOL theories are available from `http://www.verisoft.de/`

By that we give a strong argument for the feasibility of formal verification beyond its application to abstract models and toy implementations.

References

1. Alkassar, E., Hillebrand, M., Knapp, S., Rusev, R., Tverdyshev, S.: Formal device and programming model for a serial interface. In: Proc. 4th International Verification Workshop (VERIFY). CEUR-WS Workshop Proc. (2007)
2. Bevier, W., Hunt Jr., W., Moore, J.S., Young, W.: An approach to systems verification. Journal of Automated Reasoning 5(4), 411–428 (1989)
3. Beyer, S., Jacobi, C., Kroening, D., Leinenbach, D., Paul, W.: Putting it all together: Formal verification of the VAMP. International Journal on Software Tools for Technology Transfer 8(4–5), 411–430 (2006)
4. Condea, C.: Design and implementation of a page fault handler in C0. Master's thesis, Saarland University (July 2006)
5. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
6. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.: Towards trustworthy computing systems: Taking microkernels to the next level. In: Operating Systems Review (July 2007)
7. Hillebrand, M.: Address Spaces and Virtual Memory: Specification, Implementation, and Correctness. PhD thesis, Saarland University, Computer Science Department (June 2005)
8. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD 2005, pp. 309–316. IEEE Computer Society Press, Los Alamitos (2005)
9. Hohmuth, M., Tews, H., Stephens, S.: Applying source-code verification to a microkernel: the vfiasco project. In: Proc. 10th ACM SIGOPS, pp. 165–169. ACM Press, New York (2002)
10. Leinenbach, D., Petrova, E.: Pervasive compiler verification – from verified programs to verified systems. In: 3rd SSV 2008, Elsevier Science B. V (to appear, 2008)
11. Neumann, P., Feiertag, R.: PSOS revisited. In: Omondi, A.R., Sedukhin, S. (eds.) ACSAC 2003. LNCS, vol. 2823, Springer, Heidelberg (2003)
12. Shao, Z., Yu, D., Ni, Z.: Using xcap to certify realistic systems code: Machine context management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 189–206. Springer, Heidelberg (2007)
13. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (April 2006)
14. Tuch, H., Klein, G.: Verifying the L4 virtual memory subsystem. In: Proc. NICTA Formal Methods Workshop on Operating Systems Verification, NICTA, pp. 73–97 (2004)
15. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Proc. 34th POPL, pp. 97–108. ACM Press, New York (2007)
16. Walker, B., Kemmerer, R., Popek, G.: Specification and verification of the UCLA Unix security kernel. Commun. ACM 23(2), 118–131 (1980)