Formal Verification of Demand Paging

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Artem Starostin

starostin@wjpserver.cs.uni-saarland.de

Saarbrücken, März 2010

Tag des Kolloquiums:	16. März 2010
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Philipp Slusallek
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Reinhard Wilhelm (Vertreter für Dr. Gerwin Klein)
Akademischer Mitarbeiter:	Dr. Eyad Alkassar

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im November 2009

Acknowledgments

First and foremost I express my gratitude to Prof. Dr. Wolfgang J. Paul for giving me an opportunity to join possibly the world's leading research team in the field of formal verification.

I am indebted to my colleagues Dr. Norbert Schirmer, Dr. Eyad Alkassar, Dr. Mark Hillebrand, Dr. Dirk Leinenbach, Alexandra Tsyban, and Cosmin Condea. Their scientific results laid a solid foundation for the experimental part of my work and their comprehensive theses dramatically facilitated creation of this document.

I am deeply grateful to Dr. Norbert Schirmer for his guidance and collaboration on the topic of semantics stack. Every request I made to adapt the stack's definitions and proofs for the needs of this thesis led to his immediate reaction. I feel fortunate to have had a chance to work with such a responsive colleague.

This thesis work was founded by the German Federal Ministry of Education and Research (BMBF) in the frame of the Verisoft project under grant 01 IS C38 and by the International Max Planck Research School for Computer Science (IMPRS-CS).

Abstract

This thesis presents the formal pervasive verification of demand paging.

Memory virtualization by means of demand paging is a crucial component of every modern operating system. The formal verification is challenging because the reasoning about the page-fault handler (i) has to cover two concurrent computational sources: the processor and the hard disk, and (ii) involves different kinds of semantics for high- and low-level programming languages.

In order to tackle the challenge we applied a stack of semantics [Sch06, AHL+09] for a high-level C-dialect [Lei07] and low-level assembly code. It can handle mixed-language implementations and concurrently operating devices, and permits the transferral of properties to the target architecture while obeying its resource restrictions. We use a formally verified microprocessor VAMP [BJK+06] with devices [Alk09] as a target architecture to run the demand paging implementation.

The main result of this work is a mechanically checked formal proof that the page-fault handler maintains memory virtualization of user processes running on top of an operating-system microkernel: each user process is provided with the notion of an own, large and isolated memory.

This work is a part of the Verisoft project, a large scale effort bringing together industrial and academic partners to push the state-of-the-art in formal verification for realistic computer systems comprising hard- and software.

Zusammenfassung

In dieser Dissertation wird die Formale Verifikation des Demand Paging Mechanismus vorgestellt.

Speichervirtualisierung mittels Demand Paging spielt eine entscheidende Rolle bei allen modernen Betriebssystemen. Die Formale Verifikation dieses Machanismus stellt eine Herausforderung dar, da sie erstens die Beweisführung zwei nebenläufiger Berechnungsresourcen berücksichtigen muss: den Prozessor und die Festplatte, und zweitens, sie enthält verschiedene Semantiken für High- und Low-Level Programmiersprachen.

Beim Angehen dieser Herausforderung wurde ein Semantikstack [Sch06, AHL+09] für einen High-Level C-Dialekt [Lei07] und für einen Low-Level Assembler verwendet. Dieser Semantikstack kann mit Implementierungen in gemischten Programmiersprachen und mit nebenläufigen Berechnungsresourcen umgehen. Sie erlaubt eine Übermittlung der Eigenschaften zur Zielarchitektur mit Berücksichtigung der Einschränkungen dieser Architektur. Wir benutzen einen formal verifizierten Mikroprozessor VAMP [BJK+06] mit Geräten [Alk09] als Zielarchitektur, um den Demand Paging Mechanismus auszufhren.

Das wichtigste Ergebnis dieser Arbeit ist ein maschinell verifizierter Beweis über den Page-Fault Handler, welcher die Speichervirtualisierung der Benutzerprozesse verwaltet: jedem Benutzer wird sein eigener, grosser und isolierter Speicher imitiert.

Diese Arbeit ist Teil des langfristig angelegten Forschungsprojektes Verisoft. Das Projekt bringt die industriellen und akademischen Partner zusammen, um die Technologie der formalen Verifikation für realistische Computersysteme weiter zu entwickeln.

Contents

CHAPTER 1

Introduction

> A proof is a proof. What kind of a proof? It's a proof. A proof is a proof. And when you have a good proof, it's because it's proven.
>
> – *Jean Chrétien*

Computer systems nowadays are universal and omnipresent. They are crucial to the functionality of vast numbers of systems with examples ranging from electronic banking to medical technology. Our confidence in computer systems makes their reliability of large social importance. This makes correctness guarantees for computer systems a hot research topic.

For the time being, the only way to guarantee *absence* of errors in a computer system is to exploit rigorous formal methods of mathematics for specifying system's intended behavior and *proving* that the actual system's implementation meets the desired specification. The latter is known as *formal verification*. To prevent possible human errors the proof is either completely conducted or — in case it is not possible with the state-of-the-art technology — just checked by computer-aided verification tools.

With a comparably small code base of only some thousand lines of code, and implementing important safety and security abstractions as process isolation, operating-system *microkernels* seem to offer themselves as perfect candidates for a feasible approach to formal verification. Possibly 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 *demand 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. Whenever the process accesses a page located in the swap memory, a *page-fault handler* reacts by moving the requested page to the physical memory. In case the physical mem-

ory is full, some other page is swapped out.

The goal of this work is to prove correctness of exemplary demand paging software in the interactive theorem prover Isabelle/HOL [NPW02].

Related Work

The related work for this thesis could be grouped in three categories: (i) demand paging correctness, (ii) microkernel verification, and (iii) the *Verisoft* research project which hosts the present work.

Demand paging correctness. The method of storage allocation known as demand paging was first introduced in the early 1960s by the designers of the Atlas computer [KHPS61, KELS62]. Due to its conceptual simplicity the method fast became wide-spread [KR68] and subsequently classical [Tan97]. The bibliography [Smi78] suggests many sources on paging and virtual memory. However, they focus mostly on design and evaluation of page-replacement algorithms and virtual memory systems rather than on correctness.

Paper-and-pencil formalizations of demand paging which constitute the theoretical basis for this thesis were conducted by Hillebrand [Hil05]. The page-fault handler considered in this thesis was originally implemented by Condea [Con06] who also provided paper-and-pencil specifications of the implementation.

Microkernel verification. An in-depth retrospective of the microkernel verification projects is presented by Klein [Kle09]. Below we outline the key players in the field from both past and present.

First attempts to use theorem provers for formal specification and correctness proofs of operating systems date back to the mid 1970s in PSOS and UCLA DSU projects. Provably Secure Operating System (PSOS) [FN79, NBF$^+$80, NF03] was designed at SRI International as a general-purpose operating system with provable security properties. Simple illustrative proofs were carried out to demonstrate how system's properties could be formally proven. UCLA Data Security Unix (DSU) [WKP79], a kernel-structured operating system, was developed at the University of California at Los Angeles (UCLA) in the late 1970s in order to demonstrate that program verification methods could be applied to prove an operating system secure. The kernel supported processes, capabilities, pages, and non-modeled devices via kernel calls. A proof that specifications on different levels of abstraction are consistent with each other was undertaken but not completed for all portions of the kernel.

A substantial progress in microkernel verification has been achieved in the late 1980s with the KIT kernel [Bev87, Bev89] of the CLI stack [BHMY89a, BHMY89b, Moo03] developed at the University of Texas at Austin. Kernel for Isolated Tasks (KIT) was a small operating-system kernel implementing services for process scheduling, error handling, single-word message passing, and character I/O to non-modeled devices. KIT lacked dynamic creation of processes as well as shared memory and demand paging. A top-level correctness property was process isolation: execution of one process must not interfere with the other in unintended ways. The project pioneered an innovative approach for *pervasive systems verification*: not only the kernel program was verified

but also its execution environment of a processor, assembler, and compiler. The project appeared to be the first example of an accomplished mechanically checked proof of the correct implementation of a complete microkernel.

The VFiasco project [HT05, End05, HT03] held at the Technical University of Dresden aims at the formal verification of a small L4-compatible operating-system microkernel Fiasco. The Fiasco kernel is implemented with less than 15K lines of source code. Verification was undertaken in the theorem prover PVS, but the achieved results are however vague.

The Coyotos team has defined a new low-level programming language BitC with precise formal semantics in order to carry out verification of a general-purpose operating system Coyotos [JSM04], a successor of EROS [SW00]. As yet, the status of formal verification is unclear.

The L4.verified project carried out at the National ICT Australia (NICTA) is aimed at construction and verification of seL4 (secure embedded L4) microkernel [HEK+07, CKS08, EKE08, KDE09, KEH+09]. From its prototype designed in Haskel both formal model and high-performance C implementation are generated. The verification goal is to show in Isabelle/HOL that the produced implementation conforms with an abstract model via a refinement between three levels of abstraction: from C implementation — through executable specification — to the abstract kernel model. As of today, the project has completed with a result of a 200000-line formal correctness proof of the seL4 implementation.

The FLINT group at the Yale University focuses on studying separate problems in operating-system verification rather than proving a complete OS correct. XCAP [NS06] — a theoretical verification framework implemented in the Coq theorem prover — was applied to certify a realistic x86 assembly implementation of machine-context management procedures [NYS07]. Recently, results on verification of low-level programs with hardware interrupts and preemptive threads were reported [FSDG08]. The work provides a solid foundation for reasoning about preemptive kernels and hypervisors.

The Verisoft project. The context of this thesis is the Verisoft project, a large scale effort bringing together industrial and academic partners to push the state-of-the-art in formal verification for realistic computer systems, comprising hard- and software. This thesis deals with the Verisofts *academic system*, a computer system prototype for writing, signing, and sending emails. As it covers all implementation layers from the gate-level hardware up to communicating user processes it is a representative of a vertical slice of a general-purpose computer system.

The academic system comprises the following layers, which are connected by respective simulation theorems: (i) a gate-level design of the VAMP [BJK+03, BJK+06, Tve09], a RISC processor with out-of-order execution, caches [Bey05], and memory-management units [DHP05, Dal06], (ii) an operational semantics for the VAMP instruction set architecture based on the DLX ISA [MP00], (iii) an operational semantics for the VAMP assembly language, (iv) an operational semantics for the C0 programming language [LPP05, LP08, Lei07, Pet07], a slightly restricted dialect of C, (v) a framework for microkernel programmers, called Communication Virtual Machines (CVM) [GHLP05, IdRT08, Tsy09], which implements low-level microkernel functionality including a page-fault handler [Con06, ASS08], context switch [ST08b], communication [ST08a]

and memory-management primitives [Con06], (vi) an L4-inspired microkernel VAMOS [DDB08, DDSW08, DDW09], which provides support for priority-based scheduling [DDW09], user-mode device drivers, and interprocess communication (IPC) (vii) a user-level Simple Operating System (SOS) [Bog08] featuring TCP/IP communication protocol and a file system, (viii) a number of useful user applications like remote procedure calls (RPC) [Sha06, ABP09, Alk09], SMTP and signature servers, and an email client [BHW06].

A decisive novelty of the project is the integration of concurrent devices [AHK+07, HIP05, Kna08], including a hard disk, through the whole stack. The project developed a C0 semantics stack [AHL+08, AHL+09] which is orthogonal to the system stack described above. The semantics stack establishes a convenient Hoare logic [Sch05, Sch06] to reason about the sequential parts of C0 programs and simultaneously provides the means to compose the results to deal with assembly code and to integrate devices [Alk09].

All work is conducted in Isabelle/HOL [NPW02]. To support management of formal theories a repository of verification environments [HP07] is built.

Outline

This chapter ends with an introduction of notations used throughout the thesis. The remainder of the thesis is organized in seven chapters.

- In Chapter 2 we formalize the problem of virtual memory simulation: how one can provide to user processes an illusion of own memory which exceeds the physical memory. We introduce models of the VAMP processor with devices, formalize the C0 programming language, and exploiting it define CVM, a model of low-level microkernel functionality. We discuss how CVM implements demand paging in order to support memory virtualization and outline our approach to prove it correct.
- Chapter 3 is devoted to a so-called *C0 semantics stack*, a methodology for proving correctness of systems code on convenient abstract semantics level in the Hoare logics and transferring the obtained results to the level of C0 small-step semantics.
- In Chapter 4 we introduce abstract page-fault handler configurations and operations which we use to specify the intended behavior of the demand paging implementation. We introduce all invariants necessary to establish correctness of demand paging and prove them to be preserved under the defined operations.
- In Chapter 5 we specify functions of the demand paging implementation and prove their correctness against these specifications using the Hoare logics.
- Chapter 6 elaborates on transferring the results obtained in the previous chapter down to the level of C0 small-step semantics.
- In Chapter 7 we import results on the correctness of hard-disk drivers which we use — together with correctness on the small-step semantics level — to show the top-level correctness of demand paging.
- In Chapter 8 we conclude and discuss directions for the future work.

Notation

We denote the set of natural numbers including zero by $\mathbb{N}$, the set of integers by $\mathbb{Z}$, the set of boolean values $\{\mathsf{T}, \mathsf{F}\}$ by $\mathbb{B}$, and the set of identifiers, e.g., variable names, by $\mathbb{S}$. We write $pow(t)$ for the power set of t. We introduce the constant A to denote an undefined (arbitrary) value.

Lists. We denote the type of an abstract list with elements if a type t by t^*. We write $[\,]$ for an empty list and by $[x, y, \ldots]$ we construct a list of particular elements. For numbers a and b we construct a list of consecutive elements ranging from a to b by $[a, \ldots, b]$. To obtain a list of a given length n where each element is equal to x we use the function $rep(x, n)$. For concatenation of two lists l and l' we write $l \circ l'$. The head of a list l, denoted by $hd(l)$, is the first element of the list. The tail of a list l, denoted by $tl(l)$, is the part of the list besides its first element. We reverse a list l by means of the function $rev(l)$. The function $|l|$ returns the length (number of elements) of the list l. To filter a list l, i.e., keep only those elements in the list which satisfy a given predicate $P?$, we use the notation $filter(P?, l)$. We apply a function f to each element of a given list l by writing $map(f, l)$. Sometimes we interpret lists as sets (of their elements) and allow set operation like intersection of inclusion for lists.

Bit vectors. We represent bit vectors with the type $bv\text{-}t = \{0, 1\}^*$. The leftmost bit is the most significant bit and the rightmost bit is the least significant bit. Hence, we index bit vectors from right to left. For a bit vector w and two natural numbers $a > b$ we support the notation $w[a : b]$ to denote the part of the bit vector from the bit b up to the bit a. For a bit vector w we denote by $\langle w \rangle$ the conversion to the natural number with the binary representation w. For a natural number n we denote by $bin(n)$ the conversion to the binary representation of n.

Option type. Besides specific abstract data types to be defined further in this thesis, we introduce a general *option* type. It is used to extend an existing type with some error value $\bot$. For a particular type t we write $t_\bot = t \cup \{\bot\}$ to define such an extended type. All non-error values $x \in t$ will be written as $\lfloor x \rfloor$. We obtain the value of an element x of an option type by writing $[\![x]\!] = y$, provided that x is not an error value, i.e., $x = \lfloor y \rfloor$.

Pairs. We denote the first and the second elements of a pair $x = (a, b)$ by $fst(x) = a$ and $snd(x) = b$, respectively. We convert a list of pairs l into a function f by writing $map\text{-}of(l) = f$. The function f is defined as $f(x) = \lfloor y \rfloor$ in case $(x, y) \in l$, and as $f(x) = \bot$, otherwise.

We supplement predicates with a question mark, i.e., $P?$. Predicates which describe validity of some concept are supplemented with a tick, i.e., $pfh\surd$.

Through the thesis we abbreviate "SS" and "BS" for small-step and big-step semantics, respectively, whereas "H" refers to the Simpl Hoare logic. "PFH" is an abbreviation for the page-fault handler meaning anything which refers to the demand paging specification. We use a typewriter font to denote names of constant, e.g., `TOT_PHYS_PGS`, and to refer to names of variable from the implementation, e.g., `cup`.

CHAPTER 2

Virtual Memory Simulation

The virtual memory simulation problem is to introduce by hardware and software means a notion of own, large, and isolated memory to a number of user processes. In this chapter we formalize a system which provides virtual memory to user processes running on top of it. We start by introducing the processor VAMP which features address translation to support memory virtualization. Next, we present a general theory of devices and show how one can couple devices and a processor. We need devices, in particular, a hard disk to store exceeding portions of user virtual memory. Further, we introduce a formal model of C0, a slightly restricted dialect of C, featuring inline assembly, which is essential for communicating with devices. We state simulation theorems between the C0 and VAMP with devices. Then, we introduce the formal model of CVM which makes possible to run several user processes with virtual memory on top of VAMP. CVM is also implemented in C0 as a framework featuring, among others, demand paging which is an essential component to implement virtual memory. We conclude by discussing the demand paging implementation in CVM.

2.1 VAMP: Verified Architecture Microprocessor

In this section we introduce two models of the VAMP microprocessor: VAMP ISA (instruction set architecture) and VAMP assembly. While describing them we follow the thesis of Tsyban [Tsy09].

2.1.1 VAMP Instruction Set Architecture

VAMP ISA is based on but not limited to DLX ISA, a RISC processor architecture designed by Hennessy and Patterson [HP96]. Essentially, DLX is a cleaned and simplified 32-bit MIPS architecture [KH92]. VAMP features beyond the classical DLX model include mechanisms for operating-system support: user and system mode, and address translation in user mode for virtual memory implementation. Formal specification of VAMP ISA and its correctness proof towards the gate-level implementation have been originally conducted in PVS by Beyer [Bey05] and Dalinger [Dal06]. Subsequently, Tverdyshev [Tve09] translated the model into Isabelle and re-proved its correctness using to a large extend automated verification techniques.

Data representation. Data types used for representation of VAMP ISA components are summarized in Table 2.1. Registers are modeled as bit vectors. Thirty two registers form a register file which we model as a mapping from bit vectors to bit vectors. Due to compatibility with double-word floating-point instructions memories are modeled as mappings from bit vectors to pairs of bit vectors. For a thirty two bit address a we retrieve a single word from a double-word addressable memory m using the following notation.

DEFINITION 2.1 ►
Read word from ISA memory

$$m_{\text{word}}(a) \;=\; \begin{cases} \mathit{fst}(m(a[31:3])) & \text{if } a[2] = 1 \\ \mathit{snd}(m(a[31:3])) & \text{otherwise} \end{cases}.$$

Table 2.1: Data representation of VAMP ISA

Entity	Type	Validity
Register	$bv\text{-}t$	$bv\text{-}t_n\surd(w) \;=\; \lvert w\rvert = n$
Register file	$\mathit{Regf}_{\text{ISA}} =$ $bv\text{-}t \mapsto bv\text{-}t$	$\mathit{Regf}_{\text{ISA}}\surd(r) =$ $\forall\, i : bv\text{-}t_5\surd(i) \longrightarrow bv\text{-}t_{32}\surd(r(i))$
Memory	$\mathit{Mem}_{\text{ISA}} =$ $bv\text{-}t \mapsto (bv\text{-}t \times bv\text{-}t)$	$\mathit{Mem}_{\text{ISA}}\surd(m) = \forall\, a : bv\text{-}t_{29}\surd(a) \longrightarrow$ $bv\text{-}t_{32}\surd(\mathit{fst}(m(a))) \;\wedge\; bv\text{-}t_{32}\surd(\mathit{snd}(m(a)))$

Configurations. VAMP ISA configurations c_{ISA} are modeled by the record C_{ISA} with the following fields.

$c_{\text{ISA}}.pc$, $c_{\text{ISA}}.dpc$	Registers for normal and delayed program counters.
$c_{\text{ISA}}.gpr$, $c_{\text{ISA}}.spr$	General and special purpose register files.
$c_{\text{ISA}}.m$	Memory.

According to the DLX computational model the general purpose register zero always contains the zero value. The special purpose register file contains registers needed to process interrupts and registers used for virtual memory support. In the specification of ISA considered in this thesis not all of the thirty

two special purpose registers are used. Some of the special registers are just reserved for possible extensions, e.g., an integration of a floating point unit. Table 2.2 defines the set $sprs_{\mathrm{ISA}} :: pow(bv\text{-}t)$ of used special purpose register binary indices.

Validity of a VAMP ISA configuration is stated by the predicate

$$\begin{aligned} isa\surd(c_{\mathrm{ISA}}) \quad &= \quad bv\text{-}t_{32}\surd(c_{\mathrm{ISA}}.pc) \wedge bv\text{-}t_{32}\surd(c_{\mathrm{ISA}}.dpc) \\ &\wedge \quad Regf_{\mathrm{ISA}}\surd(c_{\mathrm{ISA}}.gpr) \wedge Regf_{\mathrm{ISA}}\surd(c_{\mathrm{ISA}}.gpr) \\ &\wedge \quad Mem_{\mathrm{ISA}}\surd(c_{\mathrm{ISA}}.m). \end{aligned}$$

◀ DEFINITION 2.2
Valid VAMP ISA

Table 2.2: VAMP special purpose registers

Bin. index	Dec. index	Alias	Name
00000	0	*sr*	Status register
00001	1	*esr*	Exceptional status register
00010	2	*eca*	Exceptional cause
00011	3	*epc*	Exceptional program counter
00100	4	*edpc*	Exceptional delayed program counter
00101	5	*edata*	Exceptional data
01001	9	*pto*	Page table origin
01010	10	*ptl*	Page table length
01011	11	*emode*	Exceptional mode
10000	16	*mode*	Mode

Instructions. VAMP supports a variety of instructions for (i) memory, data transfer and control operations, (ii) arithmetic, logical, test, set and shift operations, and (iii) special operations for systems calls and return from exception.

Semantics. The semantics of execution without interrupts is given by the transition function $\delta^{\mathrm{woi}}_{\mathrm{ISA}} :: C_{\mathrm{ISA}} \mapsto C_{\mathrm{ISA}}$ which yields for a configuration c_{ISA} the next state $c'_{\mathrm{ISA}} = \delta^{\mathrm{woi}}_{\mathrm{ISA}}(c_{\mathrm{ISA}})$. The definition of $\delta^{\mathrm{woi}}_{\mathrm{ISA}}$ splits cases depending on the instruction to be executed. The semantics distinguishes two execution modes: *system* denoted by $sys\text{-}mode_{\mathrm{ISA}}?(c_{\mathrm{ISA}})$, and *user* denoted by $user\text{-}mode_{\mathrm{ISA}}?(c_{\mathrm{ISA}})$. User mode restricts access to special purpose registers and execution of some instructions, e.g., a return from exception. Most notably, memory accesses are subject to *address translation* in user mode.

Address translation. In user mode all addresses for instruction fetch as well as for load/store operations are translated with the help of special purpose registers *pto* and *ptl*. Virtual address space is divided into pages of size $\mathtt{PG\_SZ} = 2^{12}$ bytes. Each virtual address *va* is split into *virtual page index* $va[31:12]$ and *byte index* $va[11:0]$ which is an offset within the page.

The main data structure for address translation is the *page table* which resides in the processor memory. Page table origin register and virtual page index specify one *page table entry*:

$$\begin{aligned} &pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va) = \\ &\qquad c_{\mathrm{ISA}}.m_{\mathrm{word}}(c_{\mathrm{ISA}}.spr(pto)[19:0] \circ rep(0,12) +_{32} va[31:12] \circ [0,0]). \end{aligned}$$

◀ DEFINITION 2.3
Page table entry

Above, $+_{32}$ is the addition of bit vectors modulo 2^{32}.

Each page table entry *pte* contains a *physical page index* $pte[31:12]$ and (i) a *valid* bit $pte[11]$ which denotes whether the page resides in the physical memory, (ii) a *protection* bit $pte[10]$ which signals whether the page is allowed to be written, and (iii) an *execution* bit $pte[9]$ which is on when the page contains executable code.

A physical page index combined with a byte index yields the complete *physical memory address*:

DEFINITION 2.4 ► Physical memory address

$$pma_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va) \quad = \quad pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[31:12] \circ va[11:0].$$

However, this computation does not always succeed and might result in raising one of the *page fault* interrupts which are introduced later in this section. Definitions of page faults share a *translation exception* predicate which we define below.

The page table length register is used to specify the amount of allocated virtual memory. In case the virtual address *va* does not belong to the user memory address translation results in a *page table length exception*:

DEFINITION 2.5 ► PTL exception

$$ptlexcp_{\mathrm{ISA}}?(c_{\mathrm{ISA}}, va) \quad = \quad \langle va[31:12] \rangle > \langle c_{\mathrm{ISA}}.spr(ptl)[19:0] \rangle.$$

Failures in address translation also occur if invalid data was used for the translation, or the processor must forbid the attempted operation at the specified address. This happens if the memory which stores an instruction is not tagged as executable, or protected memory is accessed for writing, or even the page containing this address is not present in the physical memory. Altogether, failures in address translation cause a translation exception:

DEFINITION 2.6 ► Translation exception

$$\begin{aligned} translexcp_{\mathrm{ISA}}?(c_{\mathrm{ISA}}, va, mw?, fetch?) \quad = \quad & ptlexcp_{\mathrm{ISA}}?(c_{\mathrm{ISA}}, va) \\ \vee \quad & fetch? \wedge pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[9] = 0 \\ \vee \quad & mw? \wedge pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[10] = 1 \\ \vee \quad & pte_{\mathrm{ISA}}(c_{\mathrm{ISA}}, va)[11] = 0. \end{aligned}$$

Above, the flag *mw*? indicates memory write and the flag *fetch*? denotes instruction fetch.

Interrupts. VAMP ISA computations could be broken by interrupt signals numbered with indices from zero to thirty one. Interrupts are classified according to the following criteria: (i) maskable or not maskable, (ii) internal or external, and (iii) of repeat, continue, or abort type. Maskable interrupts can be ignored under software control. If an interrupt signal arrives during execution of some instruction i and it is of repeat type then the instruction i is repeated when the program execution is resumed. If the interrupt is a continue interrupt then the instruction that follows i in the program is executed after the interrupt handling. In the remaining case the program execution is aborted.

Table 2.3 depicts interrupts supported by the VAMP ISA model. Page fault interrupts are of particular importance to this thesis. In order to define them formally, let us first introduce the *misaligned access* exception. It is raised if the memory-access width does not match (i) the delayed program counter in case of instruction fetch (instruction misalignment *imal*?), or (ii) the low-order

bits of the *effective address* $ea(c_{\mathrm{ISA}})$ (data misalignment *dmal?*):

$$imal?(c_{\mathrm{ISA}}) \;=\; c_{\mathrm{ISA}}.dpc[1] = 1 \;\vee\; c_{\mathrm{ISA}}.dpc[0] = 1,$$

$$\begin{aligned} dmal?(c_{\mathrm{ISA}}) \;=\; iw\text{-}mem?(c_{\mathrm{ISA}}) \;\wedge\; (\; &\neg iw\text{-}byte?(c_{\mathrm{ISA}}) \;\wedge\; ea(c_{\mathrm{ISA}})[0] = 1 \\ &\vee\; iw\text{-}word?(c_{\mathrm{ISA}}) \;\wedge\; ea(c_{\mathrm{ISA}})[1] = 1). \end{aligned}$$

Above, the predicate *iw-mem*? denotes that a memory access takes place whereas the predicates *iw-byte*? and *iw-word*? denote the corresponding width of that access. The effective address $ea(c_{\mathrm{ISA}})$ is computed as the sum modulo thirty two of the content of the general purpose register specified by the *rs1* instruction word field and the immediate constant.

The *page fault on fetch* interrupt is raised in the user mode whenever translation of the fetch address cannot be done:

◀ DEFINITION 2.7
Page fault on fetch

$$\begin{aligned} pff?(c_{\mathrm{ISA}}) \;=\; &\neg imal?(c_{\mathrm{ISA}}) \\ \wedge\; &user\text{-}mode_{\mathrm{ISA}}?(c_{\mathrm{ISA}}) \\ \wedge\; &translexcp_{\mathrm{ISA}}?(c_{\mathrm{ISA}}, c_{\mathrm{ISA}}.dpc, \mathsf{F}, \mathsf{T}). \end{aligned}$$

The *page fault on load/store* is similar to the page fault on fetch, but here the *effective address* of the load/store instruction is examined:

◀ DEFINITION 2.8
Page fault on load/store

$$\begin{aligned} pfls?(c_{\mathrm{ISA}}) \;=\; &\neg dmal?(c_{\mathrm{ISA}}) \\ \wedge\; &user\text{-}mode_{\mathrm{ISA}}?(c_{\mathrm{ISA}}) \\ \wedge\; &iw\text{-}mem?(c_{\mathrm{ISA}}) \\ \wedge\; &translexcp_{\mathrm{ISA}}?(c_{\mathrm{ISA}}, ea(c_{\mathrm{ISA}}), iw\text{-}write?(c_{\mathrm{ISA}}), \mathsf{F}). \end{aligned}$$

Above, the predicate *iw-write*? denotes that a memory-write accesses takes place.

Table 2.3: Interrupts of VAMP ISA

i	Name	Meaning	Mask.	Ext.	Type
0	*reset?*	Reset	No	Yes	Abort
1	*ill?*	Illegal instruction	No	No	Abort
2	$imal? \vee dmal?$	Instr. / data misalignment	No	No	Abort
3	*pff?*	Page fault on fetch	No	No	Repeat
4	*pfls?*	Page fault on load/store	No	No	Repeat
5	*trap?*	Trap / System call	No	No	Cont.
6	*ovf?*	Overflow	Yes	No	Cont.
≥ 12	$eev_i?$	Device interrupts	Yes	Yes	Cont.

It is defined only by the configuration c_{ISA} which of the internal interrupts occur in the current configuration. Conversely, external interrupts are modeled as external input *eev* :: *bv-t* of length nineteen which is a parameter to the transition function of VAMP ISA:

$$\begin{gathered} \delta_{\mathrm{ISA}} :: C_{\mathrm{ISA}} \times bv\text{-}t \mapsto C_{\mathrm{ISA}}, \\ c'_{\mathrm{ISA}} = \delta_{\mathrm{ISA}}(c_{\mathrm{ISA}}, eev). \end{gathered}$$

Interrupt signals raised in the configuration c_{ISA} are collected in the *cause* function $ca(c_{\mathrm{ISA}}, eev)$. The *masked cause* vector $mca(c_{\mathrm{ISA}}, eev)$ is computed as

a bitwise conjunction of $ca(c_{\mathrm{ISA}}, eev)$ with the mask stored in status register $c_{\mathrm{ISA}}.spr(sr)$: if interrupt i is maskable and $c_{\mathrm{ISA}}.spr(sr)[i] = 0$ then bit i is masked out. If at least one bit of $mca(c_{\mathrm{ISA}}, eev)$ is on the *jump to interrupt service routine* (JISR) signal $jisr(c_{\mathrm{ISA}}, eev)$ is activated. In case JISR is not activated we continue with uninterrupted transition: $c'_{\mathrm{ISA}} = \delta^{\mathrm{woi}}_{\mathrm{ISA}}(c_{\mathrm{ISA}})$. Otherwise, if the lowest raised interrupt is of continue type then the instruction is executed, which leads to state $\hat{c}_{\mathrm{ISA}}$:

$$\hat{c}_{\mathrm{ISA}} = \begin{cases} \delta^{\mathrm{woi}}_{\mathrm{ISA}}(c_{\mathrm{ISA}}) & \text{if } \textit{continue} \text{ interrupt} \\ c_{\mathrm{ISA}} & \text{otherwise} \end{cases}.$$

Afterwards, the program counters are set to the start address of the interrupt service routine. The exceptional versions of program counters and the status and mode registers are assigned their normal versions. Register *eca* is set to the masked interrupt cause. Register *edata* stores the data needed for interrupt handling. Finally, the mode and the status registers are set to zero. Execution of an interrupt service routine ends with the *return from exception* instruction. According to its semantics the normal versions of registers *pc*, *dpc*, *sr*, and *mode* are assigned their exceptional versions.

2.1.2 VAMP Assembly

Experience of Verisoft shows that reasoning about VAMP programs on the ISA level is unnecessarily hard for a number of reasons: (i) bit-vector encoding of instructions, (ii) bit-vector representation of operands and program counters, (iii) unwanted interrupts, and (iv) low-level specification of functional units close to their implementations. As a response to this problem a convenient abstraction, the VAMP assembly model [Tsy09, Section 3.2], was introduced.

Data representation. Data types used for representation of VAMP ISA components are summarized in Table 2.4. We represent register contents on the assembly level with thirty-two-bit natural and integer numbers. An observation shows that in Verisoft programs use integers more frequently than natural numbers. Therefore, it was decided to represent all registers except the program counters with integers. Register files are represented therefore as lists of integers. We represent memories as mappings from naturals to integers. We suppose the memory to be word-addressable and thus each memory cell must be represented with a thirty-two-bit integer.

Table 2.4: Data representation of VAMP assembly

Entity	Type	Validity
Register	$\mathbb{N}$	$\mathbb{N}_{32}\surd(x) = x < 2^{32}$
	$\mathbb{Z}$	$\mathbb{Z}_{32}\surd(x) = -2^{31} \leq x < 2^{31}$
Register file	$Regf_{\mathrm{ASM}} = \mathbb{Z}^*$	$Regf_{\mathrm{ASM}}\surd(r) =$ $\lvert r\rvert = 32 \wedge \forall i < 32 : \mathbb{Z}_{32}\surd(r[i])$
Memory	$Mem_{\mathrm{ASM}} = \mathbb{N} \mapsto \mathbb{Z}$	$Mem_{\mathrm{ASM}}\surd(m) =$ $\forall a : \mathbb{N}_{32}\surd(a) \longrightarrow \mathbb{Z}_{32}\surd(a/4)$

Configurations. VAMP assembly configurations c_{ASM} are modeled with the record C_{ASM} which has the following fields.

$c_{\mathrm{ASM}}.pc$, $c_{\mathrm{ASM}}.dpc$	Registers for normal and delayed program counters.
$c_{\mathrm{ASM}}.gpr$, $c_{\mathrm{ASM}}.spr$	General and special purpose register files.
$c_{\mathrm{ASM}}.m$	Memory.

Table 2.2 defines the set $sprs_{\mathrm{ASM}} :: pow(\mathbb{N})$ of special purpose register decimal indices which we use in the VAMP assembly model.

Validity of a VAMP assembly configuration is stated by the predicate

◀ DEFINITION 2.9
Valid VAMP assembly

$$\begin{aligned} asm\surd(c_{\mathrm{ASM}}) &= \mathbb{N}_{32}\surd(c_{\mathrm{ASM}}.pc) \wedge \mathbb{N}_{32}\surd(c_{\mathrm{ASM}}.dpc) \\ &\wedge\ Regf_{\mathrm{ASM}}\surd(c_{\mathrm{ASM}}.gpr) \wedge Regf_{\mathrm{ASM}}\surd(c_{\mathrm{ASM}}.spr) \\ &\wedge\ Mem_{\mathrm{ASM}}\surd(c_{\mathrm{ASM}}.m). \end{aligned}$$

Instructions. In contrast to bit-vector instruction representation of VAMP ISA instructions on the assembly level are modeled with inductive data type *Instr*. Its constructors have names of instruction mnemonics [Tsy09, Table 3.2].

Semantics. The effect of a single instruction execution on the assembly configuration is defined by the function $exec_{\mathrm{instr}} :: C_{\mathrm{ASM}} \times Instr \mapsto C_{\mathrm{ASM}}$ [Tsy09, Section 3.2.6]. Executions of the assembly model are specified by the transition function $\delta_{\mathrm{ASM}} :: C_{\mathrm{ASM}} \mapsto C_{\mathrm{ASM}}$ which executes current instruction $instr(c_{\mathrm{ASM}})$ with function $exec_{\mathrm{instr}}$:

$$\delta_{\mathrm{ASM}}(c_{\mathrm{ASM}}) = exec_{\mathrm{instr}}(c_{\mathrm{ASM}}, instr(c_{\mathrm{ASM}})).$$

Several assembly steps are done by the function δ^n_{ASM} defined by induction on n:

$$\begin{aligned} \delta^0_{\mathrm{ASM}}(c_{\mathrm{ASM}}) &= c_{\mathrm{ASM}}, \\ \delta^{n+1}_{\mathrm{ASM}}(c_{\mathrm{ASM}}) &= \delta^n_{\mathrm{ASM}}(\delta_{\mathrm{ASM}}(c_{\mathrm{ASM}})). \end{aligned}$$

In contrast to ISA in the assembly model interrupts are not visible and there are neither execution modes nor address translation.

2.2 Devices Framework

This section introduces the devices model used in the thesis. We start by sketching a generic device model and later illustrate by the hard disk example how this model can be instantiated with concrete devices. We show how several devices are organized in a devices system. The reader should consult [Kna08] for additional information on the devices model in general, and the doctoral thesis of Alkassar [Alk09] for the hard disk. Note that in Isabelle devices have outputs to the external environment. Since they are irrelevant to the current work, we omit them.

2.2.1 Devices

Device model. A device of type x is modeled as a finite transition system with configurations $c_x :: C_x$ and a transition function δ_x. The step function takes the current state of the device $c_x :: C_x$, a memory interface input $mifi_t :: mifi\text{-}t_t$ from the processor, and an external interface input $eifi_x :: eifi\text{-}t_x$ from a non-modeled external environment. It returns a devices' updated state $c'_x :: C_x$

and a memory interface output $\mathit{mifo}_t :: \mathit{mifo}\text{-}t_t$ to the processor. Thus, the transition function has the following signature:

$$\delta_x :: C_x \times \mathit{mifi}\text{-}t_t \times \mathit{eifi}\text{-}t_x \mapsto C_x \times \mathit{mifo}\text{-}t_t.$$

The sets of inputs $\mathit{eifi}\text{-}t_x$ from the external environment are device-specific, whereas the memory interfaces $\mathit{mifi}\text{-}t_t$ and $\mathit{mifo}\text{-}t_t$ depend only on the type t they are represented with. Devices are accessed via word-sized ports which occupy the 1024 highest word addresses of a processor memory.

Memory interface. A memory interface between a processor and devices is specified by the memory interface inputs mifi_t and the outputs mifo_t. The inputs are given by a processor to a device, and the outputs are produced by a device for a processor. The memory interface inputs mifi_t are represented by records of the type $\mathit{mifi}\text{-}t_t$ with the following fields whose representation depends on type t of the memory interface.

$\mathit{mifi}_t.\mathit{rd}$	Read flag which indicates a read operation on a device.
$\mathit{mifi}_t.\mathit{wr}$	Write flag which indicates a write operation on a device.
$\mathit{mifi}_t.a$	Access address.
$\mathit{mifi}_t.\mathit{din}$	Word-sized data input used for write accesses to a device.

We denote the idle memory interface input by $\varepsilon\text{-}\mathit{mifi}_t$. The memory interface output is a singleton $\mathit{mifo}_t :: \mathit{mifo}\text{-}t_t$ containing a word-sized data output. We denote the idle memory interface input by $\varepsilon\text{-}\mathit{mifo}_t$.

Hard disk. A device model can be instantiated with particular devices. Next, we sketch details of the hard disk model relevant for the thesis. We use the model of the disk based on the ATA/ATAPI protocol. The hard disk is parameterized over the number of sectors it has. Each sector has a size of `WORDS_PER_SECTOR` = 128 words. 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 read/written from/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 modeled by the type $\mathit{eifi}\text{-}t_{\mathrm{HD}} = \{1, 0\}$. The value $\mathit{eifi}_{hd} = 1$ indicates the end of the transfer.

Hard disk configurations c_{HD} are represented by records of the type C_{HD} . The record comprises fields for modeling hard-disk internal functionality as well as contents stored on the hard disk. For this thesis we are interested only in the following fields of the hard-disk record.

$c_{\mathrm{HD}}.s :: \mathbb{N}$	Number of sectors; has to be $\leq$ `MAX_SECTORS` $= 2^{28}$.
$c_{\mathrm{HD}}.\mathit{buf} :: \mathbb{N}^*$	The sector word buffer.
$c_{\mathrm{HD}}.\mathit{bp} :: \mathbb{N}$	The word buffer pointer.
$c_{\mathrm{HD}}.\mathit{sm} :: \mathbb{N}^*$	The swap memory content.
$c_{\mathrm{HD}}.\mathit{int} :: \mathbb{B}$	The pending interrupt flag.
$c_{\mathrm{HD}}.\mathit{cs}$	The control state; can be `IDLE`, `BRD`, `BWR`, `PRD`, `PWR`, or `ERR`.

In state `IDLE` the disk is ready to process new commands. Reading from the disk starts in state `BRD` by filling the disk buffer which is then read by the

processor when the disk is in state PRD. Similarly, write commands visit states PWR and BWR. In case an invalid command is issued, the disk transits to the error state ERR.

We restrict the set of disk states to the valid ones, which are at the points between the complete hard disk operations. These restrictions are: (i) the hard disk is in idle state, (ii) it has no pending interrupt, (iii) the size of the hard disk is sufficient, (iv) the buffer size is fixed, and (v) the buffer pointer points to the beginning of the buffer.

◀ DEFINITION 2.10
Valid hard disk

$$\begin{aligned} hd'\surd(c_{\mathrm{HD}}) \quad &= \quad c_{\mathrm{HD}}.cs = \texttt{IDLE} \\ &\wedge \quad \neg c_{\mathrm{HD}}.int \\ &\wedge \quad 8 \cdot (\texttt{BOOT\_PGS} + \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG}) \leq c_{\mathrm{HD}}.s < 2^{28} \\ &\wedge \quad |c_{\mathrm{HD}}.sm| = c_{\mathrm{HD}}.s \cdot \texttt{WORDS\_PER\_SECTOR} \\ &\wedge \quad |c_{\mathrm{HD}}.buf| = \texttt{WORDS\_PER\_SECTOR} \\ &\wedge \quad c_{\mathrm{HD}}.bp = 0 \end{aligned}$$

The meanings of the constants BOOT_PGS = 150, TOT_BIG_PGS = 1152, and PGS_PER_BIG_PG = 1024 are explained in Section 2.5.

Generalized devices. The concept of generalized devices allows us to deal with devices in a generic fashion, i.e., without having knowledge about particular kinds of devices. Generalized devices are represented by inductive data types, where each inductive constructor corresponds to a certain device. We consider only the hard disk in the thesis, however the concept is easily expendable with further devices.

A generalized device configuration c_{GD} is an instance of the inductive data type

$$C_{\mathrm{GD}} \quad = \quad \textit{dev-hd}(C_{\mathrm{HD}}) \mid \cdots \mid \textit{ill-dev}.$$

The constructor *ill-dev* is used to model an illegal device access. In order to determine whether generalized device configuration c_{GD} corresponds to a particular device predicates of the form $\textit{dev-hd?}(c_{\mathrm{GD}}) = \exists c_{\mathrm{HD}} : c_{\mathrm{GD}} = \textit{dev-hd}(c_{\mathrm{HD}})$ are used.

Generalized inputs from the external environment $\textit{eifi}_{\mathrm{GD}}$ are modeled by inductive data type

$$\textit{eifi-t}_{\mathrm{GD}} \quad = \quad \textit{eifi-hd}(\textit{eifi-t}_{\mathrm{HD}}) \mid \cdots \mid \textit{ill-eifi}.$$

The constructor *ill-eifi* is used to model illegal device inputs. In order to test whether a generalized input $\textit{eifi}_{\mathrm{GD}}$ correspond to a particular device input we use predicates like $\textit{eifi-hd?}(\textit{eifi}_{\mathrm{GD}}) = \exists c_{\mathrm{HD}} : \textit{eifi}_{\mathrm{GD}} = \textit{eifi-hd}(c_{\mathrm{HD}})$. We test whether generalized input $\textit{eifi}_{\mathrm{GD}}$ is compatible with generalized device c_{GD} by means of predicate

$$\begin{aligned} \textit{eifi-match-dev?}(c_{\mathrm{GD}}, \textit{eifi}_{\mathrm{GD}}) \quad &= \quad \textit{dev-hd?}(c_{\mathrm{GD}}) \ \wedge \ \textit{eifi-hd?}(\textit{eifi}_{\mathrm{GD}}) \\ &\vee \quad \cdots \\ &\vee \quad \textit{ill-dev?}(c_{\mathrm{GD}}) \ \wedge \ \textit{ill-eifi?}(\textit{eifi}_{\mathrm{GD}}). \end{aligned}$$

The transition function δ_{GD} of a generalized device takes a generalized device configuration c_{GD}, a memory interface input $\textit{mifi}_{\mathbb{N}}$ represented with natural numbers, and a generalized external input $\textit{eifi}_{\mathrm{DEV}}$. It returns an updated generalized device configuration c'_{GD} and a memory interface output $\textit{mifo}_{\mathbb{N}}$ represented with natural numbers. We define the generalized transition function

inductively on generalized external inputs and a generalized device configuration. In case the input is compatible with the device, we apply the step function for that device. Otherwise, the idle device is returned:

$$\delta_{\mathrm{GD}}(c_{\mathrm{GD}}, \mathit{mifi}, \mathit{eifi}_{\mathrm{GD}}) = \begin{cases} (\mathit{dev}\text{-}x(c'_x), \mathit{mifo}) & \text{if } c_{\mathrm{GD}} = \mathit{dev}\text{-}x(c_x) \\ & \wedge\ \mathit{eifi}_{\mathrm{GD}} = \mathit{eifi}\text{-}x(\mathit{eifi}_x) \\ & \wedge\ \delta_x(c_x, \mathit{mifi}, \mathit{eifi}_x) = (c'_x, \mathit{mifo}) \\ (\mathit{ill}\text{-}\mathit{dev}, \varepsilon\text{-}\mathit{mifo}_t) & \text{otherwise} \end{cases}.$$

Devices Systems. Several devices can be organized in a devices system. Let $\mathit{devnum}\text{-}t = \{1, \ldots, \texttt{MAX\_DEV}\}$ be the set of possible device identifiers. Devices systems c_{DS} are modeled as mappings from device identifiers to generalized device configurations $C_{\mathrm{DS}} = \mathit{devnum}\text{-}t \mapsto C_{\mathrm{GD}}$. We distinguish two possible kinds of transitions a device may take in a system: *internal*, which are taken as a reaction to a memory-interface input from the processor, and *external*, which process inputs from the external environment.

In an internal step the memory-interface port address *mifi.a* defines identifier *did* of the device supposed to make a transition. The internal step of a device in a system is defined by function

$$\delta^{\mathrm{INT}}_{\mathrm{DS}} :: C_{\mathrm{DS}} \times \mathit{mifi}\text{-}t_t \mapsto C_{\mathrm{DS}} \times \mathit{mifo}\text{-}t_t.$$

Device *did* performs a step with idle external inputs ε-*eifi*:

$$\delta_{\mathrm{GD}}(c_{\mathrm{DS}}(\mathit{did}), \mathit{mifi}, \varepsilon\text{-}\mathit{eifi}) = (c_{\mathrm{GD}}, \mathit{mifo}),$$

and the resulting devices system is $\delta^{\mathrm{INT}}_{\mathrm{DS}}(c_{\mathrm{DS}}, \mathit{mifi}) = (c'_{\mathrm{DS}}, \mathit{mifo})$ with

$$c'_{\mathrm{DS}}(i) \;=\; \begin{cases} c_{\mathrm{GD}} & \text{if } i = \mathit{did} \\ c_{\mathrm{DS}}(i) & \text{otherwise} \end{cases}.$$

For the external step an explicit input of an identifier *did* of the device which is supposed to make a step is necessary:

$$\delta^{\mathrm{EXT}}_{\mathrm{DS}} :: C_{\mathrm{DS}} \times \mathit{devnum}\text{-}t \times \mathit{eifi}\text{-}t_{\mathrm{GD}} \mapsto C_{\mathrm{DS}},$$

$$\delta^{\mathrm{EXT}}_{\mathrm{DS}}(c_{\mathrm{DS}}, \mathit{did}, \mathit{eifi}) \;=\; c'_{\mathrm{DS}},$$

where

$$c'_{\mathrm{DS}}(i) \;=\; \begin{cases} \mathit{fst}(\delta_{\mathrm{GD}}(c_{\mathrm{DS}}(\mathit{did}), \varepsilon\text{-}\mathit{mifi}_t, \mathit{eifi})) & \text{if } i = \mathit{did} \\ c_{\mathrm{DS}}(i) & \text{otherwise} \end{cases}.$$

Finally, we define a version of validity predicate for a hard disk which states that a valid hard disk (Definition 2.10) is present is a devices system at position `SWAP_DID`.

DEFINITION 2.11 ▶ Valid hard disk (in devices system)

$$\mathit{hd}\surd(c_{\mathrm{DS}}) \quad = \quad \exists c_{\mathrm{HD}} : c_{\mathrm{DS}}(\texttt{SWAP\_DID}) = \mathit{dev}\text{-}\mathit{hd}(c_{\mathrm{HD}}) \wedge \mathit{hd}'\surd(c_{\mathrm{HD}})$$

2.2.2 Combined Systems

We introduce a notion of a *combined system*, a processor model coupled with a devices system. Computations of such systems are guided by an external oracle, called an *execution sequence*, which defines for each point of time which of the computational sources, either the processor or some device, makes a step.

Whenever a device makes a step the external oracle additionally provides a device input. We define two particular kinds of combined systems: (i) VAMP ISA with devices, and (ii) VAMP assembly with devices.

Execution sequences and external inputs. An execution sequence is an external oracle which parametrizes runs of a combined system. A sequence element s denotes whether the processor or a device with a particular number and particular input makes a step and is modeled by the data type

$$\textit{seqel-t} \quad = \quad \textit{Proc} \mid \textit{Dev}(\mathbb{N} \times \textit{eifi-t}_{\mathrm{DEV}}).$$

A sequence *seq* is defined as a mapping from combined system's step numbers to sequence elements:

$$\textit{seq-t} \quad = \quad \mathbb{N} \mapsto \textit{seqel-t}.$$

Execution sequence *seq* is called valid, if (i) it is live with respect to the processor and all devices, i.e., for any position in the sequence there are positions further in the sequence at which the processor and devices make a step, and (ii) for any position n in the sequence where a device makes a step, an external input which matches that device type is delivered:

◀ DEFINITION 2.12
Valid execution sequence

$$\begin{aligned} \textit{seq}\surd(\textit{seq}, c_{\mathrm{DS}}) \quad = \quad & \forall\, n:\ \exists\, i:\ n < i \ \wedge\ \textit{seq}(i) = \textit{Proc} \\ \wedge \quad & \forall\, \textit{dn}, n:\ \exists\, i, \textit{eifi}_{\mathrm{GD}}:\ n < i \ \wedge\ \textit{seq}(i) = \textit{Dev}(\textit{dn}, \textit{eifi}_{\mathrm{GD}}) \\ \wedge \quad & \forall\, n:\ \textit{seq}(n) = \textit{Dev}(\textit{dn}, \textit{eifi}_{\mathrm{GD}}) \\ & \longrightarrow\ \textit{eifi-match-dev?}(c_{\mathrm{DS}}(\textit{dn}), \textit{eifi}_{\mathrm{GD}}). \end{aligned}$$

We determine the number of processor steps in a prefix of length T of execution sequence *seq* by means of the function

◀ DEFINITION 2.13
Processor step numbers

$$\textit{proc-steps}(\textit{seq}, T) = \begin{cases} 0 & \text{if } T = 0 \\ \textit{proc-steps}(\textit{seq}, T-1) & \text{if } \textit{seq}(T-1) \neq \textit{Proc} \\ \textit{proc-steps}(\textit{seq}, T-1) + 1 & \text{otherwise.} \end{cases}$$

We obtain a list of inputs corresponding to the device with number *did* from execution sequence *seq* prefix of length T by means of function

◀ DEFINITION 2.14
Devices inputs filter

$$\textit{dev-input}(\textit{seq}, \textit{did}, T) = \begin{cases} [\,] & \text{if } T = 0 \\ \textit{dev-input}(\textit{seq}, \textit{did}, T-1) \circ [\textit{eifi}_{\mathrm{GD}}] & \text{if } \textit{seq}(T-1) = \textit{Dev}(\textit{did}, \textit{eifi}_{\mathrm{GD}}) \\ \textit{dev-input}(\textit{seq}, \textit{did}, T-1) & \text{otherwise.} \end{cases}$$

Note that the length of the obtained list in the definition above is the number of steps for the particular device.

Non-interference of devices. As defined in Section 2.2.1 devices may take transitions triggered by the processor they interact with or by the external environment. As long as devices take steps triggered only by the external environment it is possible to split a computation of the combined system into two independent execution sequences of the processor and of the devices system. This allows us to reason about the processor computation separately and then exploit the result of such reasoning in order to describe the computation of the whole combined system.

We check whether a configuration of the devices system c'_{DS} is obtained from configuration c_{DS} by executing independently all devices steps contained in the prefix of sequence *seq* of length T by the predicate

DEFINITION 2.15 ► Non-interference of devices

$$\begin{aligned} &\mathit{non\text{-}interf\text{-}dev?}(c_{\mathrm{DS}}, c'_{\mathrm{DS}}, \mathit{seq}, T) = \\ &\quad \forall \mathit{did} : c'_{\mathrm{DS}}(\mathit{did}) = \delta^*_{\mathrm{GD}}(c_{\mathrm{DS}}(\mathit{did}), \\ &\qquad \mathit{rep}(\varepsilon\text{-}\mathit{mifi}_t, |\mathit{dev\text{-}input}(\mathit{seq}, \mathit{did}, T)|), \\ &\qquad \mathit{fst}(\mathit{dev\text{-}input}(\mathit{seq}, \mathit{did}, T))). \end{aligned}$$

Memory mapping for devices. Semantics of combined systems distinguishes whether the processor accesses some device by executing a load or store instruction with an effective address which belongs to devices ports. We define constant `DEVICES_BORDER` $= \langle 1^{17}0^{15} \rangle$ which partitions the VAMP memory into two parts: normal memory and devices ports.

VAMP ISA with devices. Configurations $c_{\mathrm{ISA+DS}}$ of VAMP ISA with devices are modeled by the record $C_{\mathrm{ISA+DS}}$ which has two fields.

$c_{\mathrm{ISA+DS}}.\mathit{cpu} :: C_{\mathrm{ISA}}$	Processor.
$c_{\mathrm{ISA+DS}}.\mathit{devs} :: C_{\mathrm{DS}}$	Devices system.

As for semantics, first of all we need to adapt the interrupts definitions. Since we have two memory parts, it has to be guaranteed that page tables and the fetched instruction do not lie behind `DEVICES_BORDER`. We extend the page fault on fetch predicate (Definition 2.7) with a condition that an interrupt occurs if the fetch address or, in case of user mode, the corresponding page table entry address, belongs to the devices range. For load/store we allow the accessed address to point to a device port only if the memory operation has the width of a word. The transition function of the VAMP ISA with devices model

$$\delta_{\mathrm{ISA+DS}} :: \mathbb{N} \times C_{\mathrm{ISA+DS}} \times \mathit{seq}\text{-}t \mapsto C_{\mathrm{ISA+DS}}$$

takes as arguments the number of steps T to be executed, ISA with devices configuration $c_{\mathrm{ISA+DS}}$, and execution sequence *seq* together with external inputs. The step function is defined by induction on the step numbers T. For $T = 0$ we have:

$$\delta^0_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) \;=\; c_{\mathrm{ISA+DS}}.$$

In the definition for the step $T + 1$, first we perform T steps of system:

$$(c^T_{\mathrm{ISA}}, c^T_{\mathrm{DS}}) \;=\; \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}),$$

then, depending on the current sequence element $s = \mathit{seq}(T)$ the definition for the step $T + 1$ distinguishes three cases:

- the processor makes a device access: the new states c^{T+1}_{ISA} and c^{T+1}_{DS} of both the processor and the devices system (internal step) are computed,
- the processor makes a step without a device access: only the processor new state c^{T+1}_{ISA} is generated, and
- the devices system makes a step triggered by the external environment (external step): only the new state c^{T+1}_{DS} of the devices system is generated.

VAMP assembly with devices. Configurations $c_{\text{ASM+DS}}$ of VAMP ISA with devices are modeled by the record $C_{\text{ASM+DS}}$ which has two fields.

$$\begin{array}{ll} c_{\text{ASM+DS}}.cpu :: C_{\text{ASM}} & \text{Processor.} \\ c_{\text{ASM+DS}}.devs :: C_{\text{DS}} & \text{Devices system.} \end{array}$$

Semantics of the VAMP assembly with devices model distinguishes whether a processor access devices or not in the same fashion as VAMP ISA combined systems do. The transition function of VAMP assembly with devices

$$\delta_{\text{ASM+DS}} :: \mathbb{N} \times C_{\text{ASM+DS}} \times \mathit{seq}\text{-}t \mapsto C_{\text{ASM+DS}}$$

performs n steps guided by an execution sequence seq starting from a given configuration $c_{\text{ASM+DS}}$ and yields an updated configuration of the VAMP assembly with devices. It is defined inductively over the step number. For $n = 0$ we have:

$$\delta^0_{\text{ASM+DS}}(c_{\text{ASM+DS}}, \mathit{seq}) \;=\; c_{\text{ASM+DS}}.$$

In the definition for the step $n + 1$, first we perform n steps of the system:

$$(c^n_{\text{ASM}}, c^n_{\text{DS}}) \;=\; \delta^n_{\text{ASM+DS}}(c_{\text{ASM+DS}}, \mathit{seq}).$$

Then, depending on $s = \mathit{seq}(n + 1)$ as well as whether a device access takes place the function distinguishes the three following cases:

- the processor attempts a device access, and hence new configurations c^{n+1}_{ASM} and c^{n+1}_{DS} of the processor and the devices systems, respectively, are computed,
- the process makes a step which does not access any device: only the new configuration c^{n+1}_{ASM} of the processor is computed, and
- some device performs a transition, therefore only the updated configuration c^{n+1}_{DS} is obtained.

2.3 C0 and Small-Step Semantics

C0 [Lei07] is a type-safe garbage-collected dialect of C without pointer arithmetic. C0 was chosen in Verisoft as a compromise between two orthogonal issues: the language has to be powerful enough to allow implementation of systems software while having clean formal semantics making verification of this software feasible. As systems software like a page-fault handler may access hardware resources beyond visibility of the C0 language, e.g., devices ports, C0 was extended with support of inline assembly statements.

2.3.1 Syntax

The C0 concrete syntax and visibility rules for variables are very similar to standard C. Operational semantics, though, is similar to Pascal. The major restrictions are: (i) no initialization during declarations, except for a constant declaration, (ii) no side-effects inside expressions, (iii) no function calls inside expressions, (iv) the size of arrays is fixed at the compile time, (v) no variable declarations in functions after the first statement, (vi) only one return statement which is the last control command in each function, (vii) no pointer

arithmetic, (viii) no pointers to local variables, (ix) no pointers to functions, (x) no void pointers, i.e. all pointers are typed.

Types. C0 is a typed language. C0 types are defined by the data type *ty-t* with the following constructors:

$BooleanT$,	$PtrT(tn)$, where $tn :: \mathbb{S}$,
$IntegerT$ for signed integers,	$NullT$,
$UnsgndT$ for unsigned integers,	$ArrT(n, T)$, where $n :: \mathbb{N}$, $T ::$ *ty-t*,
$CharT$,	$StructT(fs)$, where $fs :: \mathbb{S} \times$ *ty-t**.

Expressions. C0 supports the following expressions of the data type *expr-t* .

$Lit(v)$	Literal values, where v is a constant.
$VarAcc(vn)$	Variable access, where $vn :: \mathbb{S}$.
$ArrAcc(e, i)$	Access of array e with index i, where $e, i ::$ *expr-t*.
$StructAcc(e, n)$	Access of structure $e ::$ *expr-t* with field $n :: \mathbb{S}$.
$Deref(e)$	Dereferencing, where $e ::$ *expr-t*.
$UnOp(uop, e)$	Unary operations, where $uop ::$ *unop* and $e ::$ *expr-t*.
$BinOp(bop, e_1, e_2)$	Binary operations; $bop ::$ *binop* and $e_1, e_2 ::$ *expr-t*.
$LzBinOp(bop, e_1, e_2)$	Lazy binary operations; $bop ::$ *lzbinop*, $e_1, e_2 ::$ *expr-t*.

Data types for unary *unop*, binary *binop*, and lazy binary *lzbinop* operations are inductive with constructors of the form *unary-minus*, *plus*, *logical-and*, etc. Also note that constants passed as parameters to literal values are represented with the type *val-t*. Since it is also used to represent values in the big-step semantics, we define it formally later in Section 3.3 where we discuss the big-step semantics.

Statements. C0 supports the following statements modeled by the data type *stmt-t* :

Skip	The empty statement.
$Ass(e_1, e_2)$	Assignment of expressions $e_2 ::$ *expr-t* to $e_1 ::$ *expr-t*.
$PAlloc(e, tn)$	Allocation of a pointer to new element of type $tn :: \mathbb{S}$ with assignment of the address to expression $e ::$ *expr-t*.
$Comp(c_1, c_2)$	Sequential composition, where $c_1, c_2 ::$ *stmt-t*.
$Ifte(e, c_1, c_2)$	Conditional execution; $e ::$ *expr-t* and $c_1, c_2 ::$ *stmt-t*.
$Loop(e, c)$	While loop, where $e ::$ *expr-t* and $c ::$ *stmt-t*.
$SCall(vn, pn, ps)$	Call of procedure $pn :: \mathbb{S}$ with parameters $ps ::$ *expr-t** and return value assigned to variable $vn :: \mathbb{S}$.
$XCall(pn, ps, rs)$	Extended call of procedure pn with parameters ps and result expressions rs, where $pn :: \mathbb{S}$ and $ps, rs ::$ *expr-t**.
$ESCall(pn, ps, rs)$	External call to an only declared procedure of name pn.
$Return(e)$	Return from procedure.
$Asm(l)$	Inline assembly statement; $l ::$ *Instr**.

All statements in a C0 program are tagged with unique numerical identifiers which are hidden in the present thesis.

We compute the number of return statements in the statement (tree) s by the function $\#ret(s)$ [Lei07, Definition 4.3]. We convert a statement (tree) s into a list of statements by the function $s2l(s)$ [Lei07, Definition 4.5].

Programs. A type environment maps type names to types:

$$\textit{tenv-t} \quad = \quad (\mathbb{S} \times \textit{ty-t})^*.$$

A function table stores information about the functions. A single function f is described by data type *func-t* which has four components.

$f.body :: \textit{stmt-t}$	The body of the function.
$f.params :: (\mathbb{S} \times \textit{ty-t})^*$	The list of function parameters.
$f.lvars :: (\mathbb{S} \times \textit{ty-t})^*$	The list of local variables.
$f.rtype :: \textit{ty-t}$	The return type of the function.

Parameters and local variables are represented by their names and types. We call a list of pairs of names and type a *symbol table*. A complete function table is a list of pairs of function names and descriptions:

$$\textit{functable-t} \quad = \quad (\mathbb{S} \times \textit{func-t})^*.$$

Finally, a program is modeled by type *prog-t* which consists of a type environment, a function table, and a global symbol table:

$$\textit{prog-t} \quad = \quad \textit{tenv-t} \times \textit{functable-t} \times (\mathbb{S} \times \textit{ty-t})^*.$$

A program $\Pi :: \textit{prog-t}$ is called *translatable* [Lei07, Definition 7.41], denoted by $\Pi \in \textit{xltbl}_{\text{prog}}$, if it fulfills constraints on the size of immediate operands an the number of required VAMP assembly registers to evaluate expressions.

2.3.2 Small-Step Semantics

Different kinds of semantics for C0 we developed in Verisoft (cf. Chapter 3). In this section we consider the lowest-level version, the C0 small-step semantics which describes each single computation step that transform configurations.

Values. Variables and sub-variables are modeled by data type *gvar-t* which has the following constructors.

$\textit{gvar}_{\text{gm}}(vn)$	Global variable of name $vn :: \mathbb{S}$.
$\textit{gvar}_{\text{lm}}(n, vn)$	Local variable of n-th frame with name $vn :: \mathbb{S}$; $n :: \mathbb{N}$.
$\textit{gvar}_{\text{hm}}(n)$	n-th heap variable; $n :: \mathbb{N}$.
$\textit{gvar}_{\text{arr}}(g, n)$	n-th array element of variable $g ::$ *gvar-t*; $n :: \mathbb{N}$.
$\textit{gvar}_{\text{str}}(g, fn)$	Field of structure $g ::$ *gvar-t* with name $fn :: \mathbb{S}$.

In the small-steps semantics all values are flattened. We have only elementary values that could occupy one memory cell. Values of aggregate types are stored consecutively as sequences of memory cells. We model memory cells by data type *mcell-t* with the following constructors.

$mcell_{\text{bool}}(b)$	A Boolean $b :: \mathbb{B}$.
$mcell_{\text{int}}(i)$	A (signed) integer $i :: \mathbb{Z}$.
$mcell_{\text{nat}}(n)$	An unsigned integer $n :: \mathbb{N}$.
$mcell_{\text{char}}(c)$	A character $c :: \mathbb{Z}$.
$mcell_{\text{ptr}}(p)$	A pointer to $p ::$ *gvar-t*$_\perp$; $\perp$ models the null pointer.

State. Configurations c_{SS} of C0 small-step semantics are modeled by the data type C_{SS} which has two components.

$c_{\text{SS}}.mem$	The memory configuration of the type *memconf-t*.
$c_{\text{SS}}.prog$	The program rest of type *stmt-t*.

A memory configuration $c_{\text{SS}}.mem$ is a record of the type *memconf-t* with following three components.

$c_{\text{SS}}.mem.gm$	Memory frame for global variables.
$c_{\text{SS}}.mem.hm$	Heap memory frame.
$c_{\text{SS}}.mem.lm$	Stack of pairs of local memory frames and return destinations.

Each memory frame m consists of its symbol table $m.st :: (\mathbb{S} \times ty\text{-}t)^*$, the set of initialized variables $m.init :: pow(\mathbb{S})$, and the content $m.ct :: \mathbb{N} \mapsto mcell\text{-}t$.

We abbreviate global and heap symbol tables of a memory configuration mem as $gst(mem)$ and $hst(mem)$, respectively. The top-most local memory frame in the memory configuration mem is denoted by $lm_{\text{top}}(mem)$ whereas the top-most return destination is denoted by $res_{\text{top}}(mem)$. The symbol table of the top-local memory frame is denoted by $lst_{\text{top}}(mem)$.

Expression evaluation. Evaluation of expression e in memory configuration $c_{\text{SS}}.mem$ and with respect to type environment te is done by means of function $eval_{\text{SS}}(te, c_{\text{SS}}.mem, e)$ and is defined inductively over an expression tree in Section 4.3.4 of Leinenbach's thesis [Lei07]. In case e is a variable access the content from corresponding memory frame is obtained, otherwise, equivalent abstract operations are applied.

Transition function. The core of the small-step semantics is its transition function [Lei07, Section 4.4.3]. The transition function

$$\delta_{\text{SS}} :: tenv\text{-}t \times functable\text{-}t \times C_{\text{SS}} \mapsto C_{\text{SS}\perp}$$

gets a type environment te and a function table ft and calculates the transition from configuration c_{SS}:

$$\delta_{\text{SS}}(te, ft, c_{\text{SS}}) = \begin{cases} \lfloor c'_{\text{SS}} \rfloor & \text{if no fault occurs} \\ \perp & \text{otherwise} \end{cases}.$$

We execute n steps by δ^n_{SS}.

Valid configurations. A configuration c_{SS} is in the set of valid configurations $C0\surd(te, ft)$ [Lei07, Section 5.5] for a particular type environment and a function table if basic well-formedness and typing constraints hold for the configuration and the components, in particular:

- unique identifiers within the different name spaces of type names, global variable names, local variable names within the different procedures, and procedure names,
- procedure bodies are well-typed and contain a single return statement at the end,
- all memory frames are well-typed and contain only valid pointers, i.e., pointers which point to existing variables,
- the program rest conforms with the function tables, i.e., all statements in the program rest are from one of the procedures and their order follows certain rules, and
- the number of return statements in the program rest is not greater than the number of stack frames.

A version of the C0 validity $C0'\surd(te, ft)$ which is often used in program verification strengthens the last condition by requiring the number of return statements in the program rest to be equal to the recursion depth minus one.

2.3.3 Compiling C0 to VAMP

So far we have defined in this chapter three models for reasoning about programs: VAMP ISA and assembly, and C0 small-step semantics. Most of the software in Verisoft is implemented and verified at the C0 level. However the key theorem of this thesis, the top-level correctness of a page-fault handler, has to describe, among others, how C0 code is executed on the target ISA machine. In order to define this, C0 code needs to be translated — via assembly code — to the object code which is executable on the hardware. For this a simple non-optimizing compiler [Lei07, Pet07] has been developed and verified in Verisoft. The correctness theorem of the compiler specification [Lei07, Theorem 8.3] is a simulation theorem between a C0 program executed by the C0 small-step semantics and the generated assembly code executed by the VAMP assembly model. The gap towards VAMP ISA was closed by Tsyban: a simulation theorem between VAMP ISA and assembly [Tsy09, Theorem 4.8] was proven. Moreover, these two theorems were transitively combined in a verified ministack from C0 to VAMP ISA [Tsy09, Section 4.3]. We present its correctness statement below.

Abstraction relation from VAMP ISA towards assembly. The major difference between the VAMP ISA and assembly models is data representation. ISA registers are defined as bit vectors whereas assembly registers are natural numbers. ISA memory is defined as a mapping from bit vectors of length twenty nine to pairs of bit vectors of length thirty two whereas assembly memory is a mapping of thirty-bit natural numbers to thirty-two-bit integers. The abstraction relation $\mathit{isa\text{-}sim\text{-}asm?}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}})$ between VAMP ISA and assembly [Tsy09, Definition 4.6] basically defines how components of an ISA configuration are converted to their assembly equivalents.

Abstraction relation from VAMP assembly towards C0. An *allocation function* maps variables g to pairs $alloc(g) = (b, s)$ of the allocated base address b for g and the allocated size s of g's type. The simulation relation $consis?(te, ft, c_{\mathrm{SS}}, alloc, c_{\mathrm{ASM}})$ [Lei07, Definition 8.11] defines whether a C0 small-step semantics configuration and an assembly configuration are equivalent with respect to a given allocation function. It is defined as a conjunction of different individual consistency properties including those for code, control, allocation, values, pointers, registers, and frame headers.

Abstraction relation from VAMP ISA towards C0. Transitive composition of abstraction relations introduced in this section define the ministack abstraction relation from ISA towards C0:

$$\begin{aligned} \mathit{C0\text{-}sim\text{-}isa?}(te, ft, c_{\mathrm{SS}}, c_{\mathrm{ISA}}) \;=\; \exists\, c_{\mathrm{ASM}}, alloc:\;& \mathit{asm}\surd(c_{\mathrm{ASM}}) \\ \wedge\;& \mathit{consis?}(te, ft, c_{\mathrm{SS}}, alloc, c_{\mathrm{ASM}}) \\ \wedge\;& \mathit{isa\text{-}sim\text{-}asm?}(c_{\mathrm{ASM}}, c_{\mathrm{ISA}}). \end{aligned}$$

Preconditions for simulation. The predicate $dyn\text{-}C0\text{-}props?(te, ft, c_{\mathrm{SS}}, n)$ denotes necessary preconditions [Tsy09, Section 4.2.4] for simulation of C0 by VAMP ISA which have to hold at every of n steps. The preconditions are the absence of assembly statements and the "sufficient memory" requirement. The latter states that the topmost stack frame ends below the heap base and that the allocated heap size is appropriately bounded.

We denote that an ISA machine is running in system mode with all interrupts masked out by

$$\mathit{sys\text{-}exec}_{\mathrm{ISA}}?(c_{\mathrm{ISA}}) \quad = \quad \mathit{sys\text{-}mode}_{\mathrm{ISA}}?(c_{\mathrm{ISA}}) \wedge \langle c_{\mathrm{ISA}}.\mathit{spr}(\mathit{sr})\rangle = 0.$$

Additional conclusions. In order to effectively apply the simulation theorem between C0 and VAMP ISA in the context of systems verification the theorem's conclusion is enriched with additional properties which state that certain components of the system stay unchanged during the C0 execution. The predicate $no\text{-}mod\text{-}spr?(spr, spr')$ states that no special registers are modified. The predicate $only\text{-}mod\text{-}mem?(m, m', a_{\mathrm{begin}}, a_{\mathrm{end}})$ states that only the part of the memory between the addresses a_{begin} and a_{end} is modified. Both predicates are formally defined in Section 4.3.2 of Tsyban's thesis [Tsy09].

THEOREM 2.16 ▶
C0 simulates ISA

Assume that (i) the initial C0 program is translatable, (ii) the C0-ISA relation holds for the current valid C0 configuration c_{SS} and some valid VAMP ISA machine $c_{\mathrm{ISA+DS}}.cpu$ being in the system mode, (iii) the execution sequence is valid, (iv) the C0 computation starting from this configuration, does not produce a $\perp$-configuration up to the step n, (v) during these n steps we execute only non-assembly statements having enough stack and heap memory, (vi) the last C0 address `PROGEND` does not lie in the devices range, then there exists a number of ISA with devices steps T during which the ISA combined system transits to a valid resulting state $c'_{\mathrm{ISA+DS}}$. Moreover, for it and a corresponding valid C0 configuration c'_{SS} the following holds: (i) the C0-ISA relation is preserved, (ii) the special purpose registers are unchanged, (iii) only the memory, which belongs to the C0 program is possibly changed, and (iv) the devices

non-interference holds. Formally:

$$
\begin{array}{ll}
 & (te, ft, gst(c_{\mathrm{SS}}.mem)) \in xltbl_{\mathrm{prog}} \\
\wedge & \mathit{C0\text{-}sim\text{-}isa?}(te, ft, c_{\mathrm{SS}}, c_{\mathrm{ISA+DS}}.cpu) \\
\wedge & \mathit{isa}\surd(c_{\mathrm{ISA+DS}}.cpu) \wedge \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.devs) \wedge c_{\mathrm{SS}} \in \mathit{C0'}\surd(te, ft) \\
\wedge & \mathit{sys\text{-}exec}_{\mathrm{ISA}}?(c_{\mathrm{ISA+DS}}.cpu) \\
\wedge & \delta^n_{\mathrm{SS}}(te, ft, c_{\mathrm{SS}}) = \lfloor c'_{\mathrm{SS}} \rfloor \\
\wedge & \texttt{PROGEND} < \texttt{DEVICES\_BORDER} \\
\wedge & \mathit{dyn\text{-}C0\text{-}props?}(te, ft, c_{\mathrm{SS}}, n) \\
\longrightarrow & \exists\, T, c'_{\mathrm{ISA+DS}} : \\
 & \quad \delta^T_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}} \\
 & \wedge \quad \mathit{isa}\surd(c'_{\mathrm{ISA+DS}}.cpu) \wedge c'_{\mathrm{SS}} \in \mathit{C0'}\surd(te, ft) \\
 & \wedge \quad \mathit{C0\text{-}sim\text{-}isa?}(te, ft, c'_{\mathrm{SS}}, c'_{\mathrm{ISA+DS}}.cpu) \\
 & \wedge \quad \mathit{no\text{-}mod\text{-}spr?}(c'_{\mathrm{ISA+DS}}.\mathit{cpu.spr}, c_{\mathrm{ISA+DS}}.\mathit{cpu.spr}) \\
 & \wedge \quad \mathit{only\text{-}mod\text{-}mem?}(c'_{\mathrm{ISA+DS}}.\mathit{cpu.m}, c_{\mathrm{ISA+DS}}.\mathit{cpu.m}, \\
 & \qquad\qquad\qquad \mathit{abase}_{\mathrm{gm}}(te, ft, gst(c_{\mathrm{SS}}.mem)), \texttt{PROGEND}) \\
 & \wedge \quad \mathit{non\text{-}interf\text{-}dev?}(c_{\mathrm{ISA+DS}}.devs, c'_{\mathrm{ISA+DS}}.devs, \mathit{seq}, T).
\end{array}
$$

Note that above the function $\mathit{abase}_{\mathrm{gm}}(te, ft, gst)$ [Lei07, Definition 7.15] computes the base address of the global memory.

2.4 CVM: Communicating Virtual Machines

One of the most challenging verification objectives of Verisoft is to prove that a VAMP ISA machine with devices correctly implements memory virtualization for user processes: the physical memory and the swap space of the hard disk are organized by the kernel to provide separate uniform linear memories for user processes. Such organization is described by the model *communicating virtual machines* (CVM) [GHLP05, IdRT08].

CVM [Tsy09] is a computational model for concurrent user processes interacting with a generic microkernel and devices. CVM is implemented in C0 with inline assembly as a framework featuring virtual memory, demand paging [ASS08], memory management, and low-level inter-process and devices communications. The demand paging implementation includes a page-fault handler and its initialization code. Most other features are implemented in the form of so called microkernel primitives [ST08a], functions with inline assembly parts realizing basic operations which constitute the kernel's functionality. The framework can be linked on the source code level with an abstract kernel [In 09], an interface to users, in order to obtain a concrete kernel, a program that can run on a target machine, e.g., a VAMP processor.

2.4.1 Semantics

Configurations. We denote the number of processes including the kernel that are allowed to run in our system by the constant $\texttt{MAX\_PID} = 128$. For identifiers of user processes we introduce data type $\mathit{procnum\text{-}t} = \{1, \ldots, \texttt{MAX\_PID}-1\}$. Altogether user processes of the system are modeled as a mapping from process identifiers to VAMP assembly configurations. We use data type $\mathit{userprocs\text{-}t} = \mathit{procnum\text{-}t} \mapsto C_{\mathrm{ASM}}$ for that. CVM configurations c_{CVM} are modeled by record C_{CVM} which has the following components.

$c_{\mathrm{CVM}}.ak :: C_{\mathrm{C0}}^{\mathrm{mono}}$	Configuration of the abstract kernel.
$c_{\mathrm{CVM}}.ups :: \mathit{userprocs\text{-}t}$	Mapping of user processes.
$c_{\mathrm{CVM}}.ds :: C_{\mathrm{DS}}$	Configuration of the devices system.
$c_{\mathrm{CVM}}.cup :: \mathit{procnum\text{-}t}_{\perp}$	Current-process identifier.
$c_{\mathrm{CVM}}.sr :: \mathbb{N}$	Status-register used as a mask for interrupts.

The abstract kernel is modeled by the *monolithic* C0 small-step semantics configuration which (additionally) includes a type environment and a function table. Each user process is modeled by the VAMP assembly semantics. Configurations of the CVM model are by no means restricted to any particular instantiation of the devices system component. However, we will instantiate the devices-system component with the devices system of the underlying physical combined system from which the swap hard disk is removed as swapping is transparent to the CVM model. The identifier of a current process is modeled by option type $\mathit{procnum\text{-}t}_{\perp}$: the value $\perp$ corresponds to the kernel whereas any value *pid* which belongs to the set *procnum-t* corresponds to the process with number *pid*. The status register component represents the interrupt mask shared between the user processes. CVM executions are parametrized with abstract kernel code $\Pi_{\mathrm{AK}} :: \mathit{prog\text{-}t}$. Initial configuration of the CVM model for devices system configuration c_{DS} is denoted by $\mathit{cvm}_{\mathrm{init}}(\Pi_{\mathrm{AK}}, c_{\mathrm{DS}})$.

Computations. The parameters of the CVM transition function

$$\delta_{\mathrm{CVM}} :: C_{\mathrm{CVM}} \times \mathit{seqel\text{-}t} \mapsto C_{\mathrm{CVM}\perp}$$

are a CVM model configuration $c_{\mathrm{CVM}} :: C_{\mathrm{CVM}}$, and an execution-sequence element $s :: \mathit{seqel\text{-}t}$. In case the sequence element corresponds to a processor step either the kernel or some user makes progress. Otherwise, δ_{CVM} boils down to a step of some device. The CVM transition function yields either an error constant $\perp$ or an updated configuration of the CVM model.

Depending on the execution sequence element s and the current-process identifier $c_{\mathrm{CVM}}.\mathit{cup}$ the CVM transition function $\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, s)$ distinguishes the following cases.

Devices step:	s corresponds to some device.
User step:	s is a processor step and $c_{\mathrm{CVM}}.\mathit{cup}$ is a user-process identifier.
Kernel step:	s is a processor step and $c_{\mathrm{CVM}}.\mathit{cup}$ corresponds to the kernel.

Formally, the CVM transition function is

$$\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, s) = \begin{cases} \lfloor \mathit{step}_{\mathrm{user}}(c_{\mathrm{CVM}}) \rfloor & \text{if } s = \mathit{Proc} \wedge c_{\mathrm{CVM}}.\mathit{cup} = \lfloor \mathit{pid} \rfloor \\ \mathit{step}_{\mathrm{kernel}}(c_{\mathrm{CVM}}) & \text{if } s = \mathit{Proc} \wedge c_{\mathrm{CVM}}.\mathit{cup} = \perp \\ \lfloor \mathit{step}_{\mathrm{devs}}(c_{\mathrm{CVM}}, \mathit{did}, \mathit{eifi}) \rfloor & \text{if } s = \mathit{Dev}(\mathit{did}, \mathit{eifi}) \end{cases}.$$

For formal definitions of the functions $\mathit{step}_{\mathrm{devs}}$, $\mathit{step}_{\mathrm{user}}$, and $\mathit{step}_{\mathrm{kernel}}$ consult the thesis of Tsyban [Tsy09].

A device step $\mathit{step}_{\mathrm{devs}}$ [Tsy09, Definition 5.14] is taken as a response to an input from the external environment and boils down to an application of external device step function $\delta_{\mathrm{DS}}^{\mathrm{EXT}}$.

User steps distinguish three cases: (i) a user step without interrupts, (ii) a user step with an interrupt that aborts the user execution (illegal, misalignment or PTL exception), and (iii) a user step with an interrupt which allows

us to take a step (external interrupts, trap, and overflow). User steps without interrupts boil down to an application of the VAMP assembly transition function to the user process which is making a step. Steps with interrupts results in setting the current-process identifier to the kernel value and the abstract-kernel invocation. Semantics of user processes update depends on the kind of interrupt. In case (ii) the user is not changed, otherwise it makes a step as in case (i). Altogether, the function $step_{\text{user}}$ [Tsy09, Definition 5.15] updates the CVM-state components for user processes, the current-process identifier, and the abstract kernel.

We distinguish three cases of a kernel step $step_{\text{kernel}}$ [Tsy09, Definition 5.21]: (i) waiting for interrupts from devices, (ii) finishing the kernel execution, and (iii) a step of the abstract kernel. The first case models a situation when there is not even a single user-process in the system to be resumed. In this case, the kernel has no jobs to accomplish, and hence its configuration remains the same. We say that the kernel is *waiting for interrupts* in this situation. If an interrupt occurs, the kernel will be restarted. The second case handles a switch from the kernel execution either to an execution of the next scheduled user process, or to the idle state defined in the first case. The last case models a step of the abstract kernel. This step is either a simple C0 step of the abstract kernel or an execution of some CVM primitive.

The multiple-step function of the CVM $\delta^{\text{n}}_{\text{CVM}}$ is defined by induction on the step number n:

$$\begin{aligned} \delta^{0}_{\text{CVM}}(c_{\text{CVM}}, seq) &= \lfloor c_{\text{CVM}} \rfloor \\ \delta^{\text{n}+1}_{\text{CVM}}(c_{\text{CVM}}, seq) &= \begin{cases} \bot & \text{if } \delta^{\text{n}}_{\text{CVM}}(c_{\text{CVM}}, seq) = \bot \\ \delta_{\text{CVM}}(c^{n}_{\text{CVM}}, seq(n)) & \text{if } \delta^{\text{n}}_{\text{CVM}}(c_{\text{CVM}}, seq) = \lfloor c^{n}_{\text{CVM}} \rfloor \end{cases} . \end{aligned}$$

Implementation. The CVM model is implemented in C0 with inline assembly as a framework [Tsy09, Section 6.1] which consists of (i) process-context switch procedures saving and restoring contexts of user processes, (ii) *demand paging* implementation: a page-fault handler with its initialization code as well as elementary device drivers, (iii) an elementary dispatcher which decides whether an invocation of a page-fault handler is needed and calls the dispatcher of an abstract kernel, and (iv) fourteen primitives for different operations for user processes.

2.4.2 Correctness

Linker. As mentioned before a concrete kernel, a complete kernel program that can run on a computer, is obtained by linking an abstract kernel with the CVM framework. Next we follow Chapter 6 of Tsyban's thesis [Tsy09] and sketch how the linking operator is formally defined. Let the program of the CVM framework be $\Pi_{\text{CVM}} = (te_{\text{CVM}}, ft_{\text{CVM}}, gst_{\text{CVM}})$.

Linking of two type environments te and te' is done by means of the function $link_{\text{te}}(te, te')$ [Tsy09, Definition 6.1]. The result is a type environment which contains all types which constitute both original type environments.

Linking of two function tables ft and ft' is done by means of the function $link_{\text{ft}}(ft, ft')$ [Tsy09, Definition 6.6]. First, the function removes all external functions from ft', deletes entries of external functions in ft which are defined in ft', and replaces all external function call statements in ft by ordinary calls

in case a callee is defined in ft'. Next, the same is done for ft with respect to ft'. Finally, the modified function tables are subject to concatenation. Here one peculiarity has to be considered. The C0 small-step semantics requires that each statement in a program is uniquely tagged with a statement identifier. The concatenation mentioned above violates the uniqueness of statements identifiers property. Therefore, the statements of the two function tables are renumbered by means of the functions $renum^{\text{odd}}_{\text{fun}}$ and $renum^{\text{even}}_{\text{fun}}$ [Tsy09, Definition 6.5] to regain the statement identifier uniqueness and only afterwards concatenated.

Linking of two symbol tables st and st' is done by means of the function $link_{\text{st}}(st, st')$ [Tsy09, Definition 6.7]. It concatenates the first symbol table with the part of the second symbol table from which all entries that occur in the first symbol table are removed.

Linking of two C0 programs Π and Π' is done by means of the function $link_{\Pi}(\Pi, \Pi') = \Pi''$ [Tsy09, Definition 6.8] which uses the above defined functions to link the program components:

$$\begin{aligned} \Pi''.te &= link_{\text{te}}(\Pi.te, \Pi'.te), \\ \Pi''.ft &= link_{\text{ft}}(\Pi.ft, \Pi'.ft), \\ \Pi''.gst &= link_{\text{st}}(\Pi.gst, \Pi'.gst). \end{aligned}$$

The C0 program of the concrete kernel Π_{CK} and its individual components are defined below. All of them are parametrized by the program of the abstract kernel Π_{AK}.

DEFINITION 2.17 ► Program of the concrete kernel

$$\begin{aligned} \Pi_{\text{CK}}(\Pi_{\text{AK}}) &= link_{\Pi}(\Pi_{\text{CVM}}, \Pi_{\text{AK}}) \\ te_{\text{CK}}(\Pi_{\text{AK}}) &= \Pi_{\text{CK}}(\Pi_{\text{AK}}).te \\ ft_{\text{CK}}(\Pi_{\text{AK}}) &= \Pi_{\text{CK}}(\Pi_{\text{AK}}).ft \\ gst_{\text{CK}}(\Pi_{\text{AK}}) &= \Pi_{\text{CK}}(\Pi_{\text{AK}}).gst \end{aligned}$$

Correctness of linking is justified by showing the C0 validity of the individual components of the linked program. In order to succeed with that a number of assumptions on the structure of the abstract kernel's program Π_{AK} have to be imposed. These assumptions are listed in Section 6.4 of Tsyban's thesis [Tsy09] and are collected in the predicate

$$\textit{abs-kernel-props}(\Pi_{\text{AK}}).$$

Abstraction relation. Essentially, correctness criteria of CVM are formulated as an abstraction relation from a VAMP ISA with devices configurations towards the CVM model state. The abstraction relation denoted by

$$\textit{cvm-sim}?(\Pi_{\text{AK}}, c_{\text{CVM}}, c_{\text{SS}}, c_{\text{ISA+DS}})$$

holds if the CVM configuration c_{CVM} encodes both the states of VAMP ISA with devices $c_{\text{ISA+DS}}$ and the concrete kernel c_{SS} with respect to the abstract kernel program Π_{AK}. The relation distinguishes user and kernel executions within CVM and is formally defined in Section 7.1 of Tsyban's thesis [Tsy09].

Since demand paging executions belong to the kernel we discuss below only essential terms of the abstraction relation during the kernel runs. The latter comprises (i) the kernel relation [Tsy09, Definition 7.10] which defines how a concrete kernel is related to the abstract one, (ii) the devices relation [Tsy09, Definition 7.1] which claims that the device states in CVM $c_{\text{CVM}}.ds$ and VAMP ISA $c_{\text{ISA+DS}}.devs$ configurations are equal except for the swap hard disk which is an idle device in CVM, (iii) the relation for user processes [Tsy09, Definition

7.3] which states that the user processes configurations $c_{\mathrm{CVM}}.ups$ are represented in the configuration $c_{\mathrm{ISA+DS}}$ of the VAMP ISA machine, and (iv) the equality between the status register value retrieved from the memory of VAMP ISA and the corresponding CVM component $c_{\mathrm{CVM}}.sr$.

The relation for user processes is of particular importance to the present work since it can be maintained only with a correct page-fault handler. The registers of suspended user processes are implemented by the *process control blocks* (PCBs), a data structure maintained by the CVM framework which we describe in details later in Section 2.5. An active user process, i.e., the one that is currently running on the processor, is represented by the contents of hardware registers. In both cases memories of user processes are stored in the physical memory and on the hard disk. The cases are distinguished by analyzing the value of the current process identifier and the execution mode of the ISA machine.

The function $vm(c_{\mathrm{ISA+DS}}, p)$ performs the mentioned case distinction and reconstructs a virtual assembly machine for a given process identifier p.

◀ DEFINITION 2.18
Reconstruction of user processes

$$vm(c_{\mathrm{ISA+DS}}, p) = \begin{cases} vm_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}, p) & \text{if } \neg \mathit{sys\text{-}mode}_{\mathrm{ISA}}?(c_{\mathrm{ISA+DS}}.cpu) \wedge val_{cup}(c_{\mathrm{ISA+DS}}) = p \\ vm_{\mathrm{direct}}(c_{\mathrm{ISA+DS}}, p) & \text{otherwise} \end{cases}$$

In the first case the process p is suspended and its virtual machine is reconstructed from variables (PCBs) by means of the function vm_{vars}. The second case corresponds to a situation when the process p is active and its virtual machine is constructed directly from the hardware registers with the help of the function vm_{direct}. The reader can find the definitions of both functions in Section 7.1.3 of Tsyban's thesis [Tsy09]. By $val_{cup}(c_{\mathrm{ISA+DS}})$ we denote the value of the current process identifier on the side of CVM's implementation — the variable `cup` — read from the memory of $c_{\mathrm{ISA+DS}}$. For the reconstruction of user memory both vm_{vars} and vm_{direct} use the function $mem(c_{\mathrm{ISA+DS}}, p)$ which we introduce next. Its definition uses functions $nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad)$, $int_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad)$, and $int_{\mathrm{swap}}(c_{\mathrm{DS}}, ad)$ which retrieve natural or integer numbers from the physical or swap memory at a given address ad.

For each memory address, the decision where the data can be found is taken according to the valid bit of the respective page table entry. On the input we have a process identifier p and a virtual address va. Below we formally describe how address translation mechanism is defined in the CVM formal theories[1].

The page index and the byte index of a virtual address va are defined as follows.

◀ DEFINITION 2.19
Page index and byte index

$$\begin{aligned} px(va) &= va/\texttt{PG\_SZ}, \\ bx(va) &= va \bmod \texttt{PG\_SZ}. \end{aligned}$$

The page table origin and length for a process p are obtained from the memory of ISA by reading the registers pto and ptl from the process control block at addresses ad^p_{pto} and ad^p_{ptl}, respectively.

◀ DEFINITION 2.20
Page table origin and length

$$pto_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p) = nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad^p_{pto})$$

[1] Note that the hardware specification of this mechanism was already given in Section 2.1.1. Moreover Chapter 4 will define the same mechanism in terms of the abstract configurations of the page-fault handler. All three representations of the address translation mechanism are shown to be equivalent during the proofs of the CVM and page-fault handler top-level correctness theorems.

$$ptl_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p) \quad = \quad int_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad^p_{ptl})$$

The i-th page table entry of a process p is delivered by reading the i-th word in the physical memory starting from the respective page table origin.

DEFINITION 2.21 ▶ Page table entry

$$pte_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p, i) \quad = \quad nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, pto_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p) \cdot \texttt{PG\_SZ} + i \cdot 4)$$

The page index of a page table entry has a meaning of a physical page index. Combining it with a byte index of the virtual address we get the physical memory address.

DEFINITION 2.22 ▶ Physical memory address

$$pma_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p, va) \quad = \quad px(pte_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p, px(va))) \cdot \texttt{PG\_SZ} + bx(va)$$

We use big pages of size `BIG_PG_SZ` bytes. Therefore, big-page and big-byte indices of a virtual address va represented as a natural number are defined as:

DEFINITION 2.23 ▶ Big-page index and big-byte index

$$\begin{array}{rcl} bpx(va) & = & va/\texttt{BIG\_PG\_SZ}, \\ bbx(va) & = & va \mod \texttt{BIG\_PG\_SZ}. \end{array}$$

We obtain the big-page table origin for a process p by reading a natural number from the memory of ISA with devices at address ad^p_{bpto}.

DEFINITION 2.24 ▶ Big-page table origin

$$bpto_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p) \quad = \quad nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad^p_{bpto})$$

Note, that big-page table origins do not store absolute values of addresses, but only indices within the *bpt* array. To obtain the absolute address we need to add the start address of this array ad_{bpt}. The i-th big-page table entry of a process p is defined as the i-th word read from the physical memory starting from the corresponding big-page table origin.

DEFINITION 2.25 ▶ Big-page table entry

$$bpte_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p, i) \quad = \quad nat_{\mathrm{vars}}(c_{\mathrm{ISA}}, ad_{bpt} + bpto_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p) \cdot 4 + i \cdot 4)$$

The swap memory address is defined as a combination of the respective big-page table entry and the big-byte index.

DEFINITION 2.26 ▶ Swap memory address

$$sma_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p, va) \quad = \quad bpte_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p, bpx(va)) \cdot \texttt{BIG\_PG\_SZ} + bbx(va)$$

From a page table entry *pte* we extract the valid bit at bit position eleven.

DEFINITION 2.27 ▶ Valid bit

$$valid_{\mathrm{CVM}}(pte) = pte/\texttt{PG\_SZ} \mod 2$$

The value of this bit signals whether the page we are considering resides in the main or swap memory. We define the function which makes this decision and creates assembly memory components.

DEFINITION 2.28 ▶ Assembly memory reconstruction

$$mem(c_{\mathrm{ISA+DS}}, p)(va) = \begin{cases} int_{\mathrm{vars}}(c_{\mathrm{ISA+DS}}.cpu, pma_{\mathrm{CVM}}(c_{\mathrm{ISA+DS}}.cpu, p, va \cdot 4)) & \text{if } valid? \\ int_{\mathrm{swap}}(c_{\mathrm{ISA+DS}}.devs, sma_{\mathrm{CVM}}(c_{\mathrm{ISA+DS}}.cpu, p, va \cdot 4)) & \text{otherwise} \end{cases}$$

Above, $valid? = valid_{\mathrm{CVM}}(pte_{\mathrm{CVM}}(c_{\mathrm{ISA+DS}}.cpu, p, px(va \cdot 4))) = 1$.

Now we are able to define a correctness criterion for user processes. Its idea is to reconstruct virtual machines for every user process and check whether each of these machines match the one specified by the CVM user process component *ups*. For this we define an equality check operator for assembly machines. The operator is parametrized *vm-size*, an upper bound for memory addresses to be read and checked.

◀ DEFINITION 2.29
Assembly configurations equality

$$\begin{aligned} \textit{asm-equal?}(c^1_{\mathrm{ASM}}, c^2_{\mathrm{ASM}}, \textit{vm-size}) = \\ & c^1_{\mathrm{ASM}}.\textit{dpc} = c^2_{\mathrm{ASM}}.\textit{dpc} \\ \wedge\ & c^1_{\mathrm{ASM}}.\textit{pc} = c^2_{\mathrm{ASM}}.\textit{pc} \\ \wedge\ & \textit{tl}(c^1_{\mathrm{ASM}}.\textit{gpr}) = \textit{tl}(c^2_{\mathrm{ASM}}.\textit{gpr}) \\ \wedge\ & \textit{stored-spr}(c^1_{\mathrm{ASM}}.\textit{spr}) = \textit{stored-spr}(c^2_{\mathrm{ASM}}.\textit{spr}) \\ \wedge\ & \forall\ a < \textit{vm-size} :\ c^1_{\mathrm{ASM}}.m(a/4) = c^2_{\mathrm{ASM}}.m(a/4) \end{aligned}$$

Above, $\textit{stored-spr}(\textit{regs}) = [\textit{regs}[\textit{sr}], \textit{regs}[\textit{pto}], \textit{regs}[\textit{ptl}], \textit{regs}[\textit{mode}]]$ is the lists of those special purpose registers that are stored in process control blocks or other kernel variables.

The only thing that remains before we can state the desired correctness relation for user processes is the right choice of the parameter *vm-size*. For each process p we should compare only those virtual memory parts that have been allocated. The size of the allocated memory measured in pages is stored in the page table length register and is expresses by the formula $\textit{ptl}_{\mathrm{CVM}}(c_{\mathrm{ISA}}, p)+1$.

Ultimately, the relation for user processes, denoted by $\mathcal{B}(\textit{ups}, c_{\mathrm{ISA+DS}})$, is nothing but an equality of the reconstructed assembly machines and all user processes parametrized with the right amount of virtual memory.

◀ DEFINITION 2.30
Relation for user processes

$$\begin{aligned} \mathcal{B}(\textit{ups}, c_{\mathrm{ISA+DS}}) \ =\ & \forall\ 0 < p < \texttt{MAX\_PID} : \\ & \textit{asm-equal?}(\textit{vm}(c_{\mathrm{ISA+DS}}, p), \textit{ups}(p), (\textit{ptl}_{\mathrm{CVM}}(c_{\mathrm{ISA+DS}}.\textit{cpu}, p) + 1) \cdot \texttt{PG\_SZ}) \end{aligned}$$

Implementation invariants. To prove that the CVM abstraction relation holds throughout CVM executions a number of invariants over the concrete kernel's C0 machines as well as the underlying ISA machine with device has to hold. These *implementation invariants* are formally defined in Section 7.1.6 in thesis of Tsyban [Tsy09]. All implementation invariants are collected in the predicate

$$\textit{impl-inv?}(\Pi_{\mathrm{AK}}, c_{\mathrm{CVM}}, c_{\mathrm{SS}}, c_{\mathrm{ISA+DS}}).$$

The formal definition distinguishes cases of user and kernel executions. The implementation invariants which hold during kernel executions include (i) the validity of the ISA machine, ISA code invariants $\textit{code-inv?}(\Pi_{\mathrm{AK}}, c_{\mathrm{ISA}})$, and the zero-filled page condition $\textit{zfp-cond?}(c_{\mathrm{ISA}})$, (ii) a requirement to the ISA machine to operate in system mode, (iii) the validity of the concrete kernel C0 configuration, (iv) memory structure invariants of the concrete kernel, and (v) the C0-ISA ministack relation $\textit{C0-sim-isa?}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}, c_{\mathrm{ISA}})$ discussed in Section 2.3.3.

The ISA code invariants [Tsy09, Definition 7.11] $\textit{code-inv?}(\Pi_{\mathrm{AK}}, c_{\mathrm{ISA}})$ state that (i) the instruction stored in the ISA memory at address zero must be a jump instruction to the beginning of the kernel code, and (ii) the concrete kernel code translated by the C0 compiler resides in the ISA memory starting from the address `PROGBASE`.

◀ DEFINITION 2.31
ISA code invariants

The zero-filled page condition $\textit{zfp-cond?}(c_{\mathrm{ISA}})$ states that a page filled with zeros resides at the address `ZFP` in the memory of the ISA machine c_{ISA}.

◀ DEFINITION 2.32
Zero-filled page condition

We discuss the zero filled page in details further in this chapter.

CVM correctness theorem. Before we state the correctness theorem of CVM we need to mention a number of additional definitions. All of them are introduced in Section 7.2 of Tsyban's thesis [Tsy09]. The predicate *init-isa*?(Π_{AK}, c_{ISA}) claims that c_{ISA} is the initial ISA configuration which is the first valid configuration after reset. $\texttt{HEAP-SIZE}_{\text{AK}}$ is the upper bound on the heap memory of the abstract kernel. The function $asize_{\text{heap}}(hst)$ [Lei07, Definition 7.12] computes the allocated size of the heap for a heap symbol table *hst*. Finally, for two generalized device configurations d_1 and d_2 we denote that both devices are of the same type by

$$d_1 \overset{\text{type}}{\sim} d_2.$$

THEOREM 2.33 ► CVM correctness theorem

Assume that (i) the abstract kernel properties hold for Π_{AK}, (ii) the processor component $c_{\text{ISA+DS}}.cpu$ of ISA with devices is in its initial state, (iii) the swap hard disk properties hold for the devices system $c_{\text{ISA+DS}}.devs$, (iv) the ISA execution sequence seq_{ISA} is valid, then there exists a valid CVM execution sequence seq_{CVM} and for any finite number n of kernel steps $\leq N$ there exists a number of CVM model steps T_{CVM}, such that by executing this number of steps starting from initial CVM configuration the CVM model transits to a non-error state c'_{CVM} with no heap boundaries violation. Moreover, there exists a number of ISA with devices steps T_{ISA} during which the ISA machine transits to a resulting state $c'_{\text{ISA+DS}}$ and a corresponding configuration c'_{SS} of the concrete kernel, such that (i) the devices typing and the swap hard disk properties hold for the updated devices system, (ii) the implementation invariants hold, and (iii) the CVM simulation relation holds. Formally:

$$
\begin{array}{lll}
 & \textit{abs-kernel-props}(\Pi_{\text{AK}}) & \\
\wedge & \textit{init-isa}?(\Pi_{\text{AK}}, c_{\text{ISA+DS}}.cpu) & \\
\wedge & hd\surd(c_{\text{ISA+DS}}.devs) & \\
\wedge & seq\surd(seq_{\text{ISA}}, c_{\text{ISA+DS}}.devs) & \\
\longrightarrow & (\exists\, seq_{\text{CVM}} : \forall\, n \leq N : \exists\, T_{\text{CVM}} : & \\
 & \quad \textit{proc-steps}(seq_{\text{CVM}}, T_{\text{CVM}}) = n & \\
 & \wedge \quad seq\surd(seq_{\text{CVM}}, ds_{\text{init}}(c_{\text{ISA+DS}}.devs)) & \\
 & \wedge \quad (\forall\, c'_{\text{CVM}} : & \\
 & \qquad \delta_{\text{CVM}}^{T_{\text{CVM}}}(cvm_{\text{init}}(\Pi_{\text{AK}}, c_{\text{ISA+DS}}.devs), seq_{\text{CVM}}) = \lfloor c'_{\text{CVM}} \rfloor & \\
 & \qquad \wedge \quad asize_{\text{heap}}(hst(c'_{\text{CVM}}.ak.mem)) \leq \texttt{HEAP-SIZE}_{\text{AK}} & \\
 & \qquad \longrightarrow (\exists\, T_{\text{ISA}}, c'_{\text{ISA+DS}}, c'_{\text{SS}} : & \\
 & \qquad\quad \delta_{\text{ISA+DS}}^{T_{\text{ISA}}}(c_{\text{ISA+DS}}, seq_{\text{ISA}}) = c'_{\text{ISA+DS}} & \\
 & \qquad\quad \wedge \quad \forall\, did :\ c'_{\text{ISA+DS}}.devs(did) \overset{\text{type}}{\sim} c_{\text{ISA+DS}}.devs(did) & \\
 & \qquad\quad \wedge \quad hd\surd(c'_{\text{ISA+DS}}.devs) & \\
 & \qquad\quad \wedge \quad \textit{impl-inv}?(\Pi_{\text{AK}}, c'_{\text{CVM}}, c'_{\text{SS}}, c'_{\text{ISA+DS}}) & \\
 & \qquad\quad \wedge \quad \textit{cvm-sim}?(\Pi_{\text{AK}}, c'_{\text{CVM}}, c'_{\text{SS}}, c'_{\text{ISA+DS}}))))).
\end{array}
$$

Above, we have presented the formal model of CVM and its correctness theorem: a simulation theorem between a physical ISA machine and virtual machines of user processes. Most crucially the correctness of the simulation theorem depends on the correctness of demand paging as the page-fault handler takes care about maintaining the relation for user processes. In the next section we focus on demand paging *implementation* in CVM whereas the remainder of

the thesis will be concerned about its *correctness*.

2.5 Demand Paging

User processes are modeled as virtual assembly machines in the CVM model. This means that user processes have an illusion of their own, large, and isolated memory. This is implemented via demand paging based on the VAMP's address translation mechanism and a swap hard disk. Memory virtualization is transparent to user processes: within the CVM model page faults that might occur during a user step are handled silently by the page-fault handler such that the user can continue its run. The *page table space*, a data structure both accessed by the processor and by software, maintains information whether a certain page is in the swap or the main memory. Whenever a user processes accesses a page that is currently swapped out, a page-fault interrupt is triggered by the processor (cf. Definitions 2.7 and 2.8). In response, the page-fault handler is invoked, which copies the requested page back to the main memory. To copy pages between the swap disk and the main memory the page-fault handler relies on a hard-disk driver [Alk09]. Both the handler and the driver are part of the CVM framework implementation.

In this section we follow the master thesis of Condea [Con06] and describe which data structures are maintained by the demand paging implementation and outline execution scenarios of the page-fault handler.

2.5.1 Data Structures

Constants. The implementation of demand paging uses the following constants which define the memory map of the kernel.

- Our system configuration, and implicitly the page fault handler, is parameterized with two constants `PG_SZ` and `BIG_PG_SZ` holding the page and, respectively the big-page size. The former, as aforementioned, is set to 4KB whereas the latter is 4MB. The ratio between big-page and page size is denoted by `PGS_PER_BIG_PG` $= 1024$. Moreover, page size measured in words is denoted by `PG_SZ_WD` $= 1024$.

- The user memory region is placed to the high end of the physical memory which is `TOT_PHYS_PGS` $= 8192$ pages large whereas everything below it belongs to the system memory. Thus, we keep the index of the first user memory page or, equivalently, the number of pages given to the kernel in `KERNEL_PGS` $= 6390$ constant. Pages from 0 to `KERNEL_PGS`-1 are reserved for the kernel whereas the remaining `USER_PGS` $= 8192 - 6390 = 1802$ are used for user processes. We devote the larger part of the available physical memory to the kernel because in order to show competitive performance it has to completely reside in the physical memory.

- The last page of the system memory which resides at address `ZFP` $=$ `KERNEL_PGS` $- 1$ is permanently filled with zeros to support the *copy-on-write* principle.

- The swap memory stores `TOT_BIG_PGS` = 1152 big-pages. The first `BOOT_PGS` = 150 pages are reserved for the boot region which stores an operating-system image.
- Constant `PT_START` = 4194304 stores the absolute address where the page table space begins. It must be known in order to appropriately set page table origins.
- The total amount of virtual memory is kept in the `TOT_PGS` constant and set in our system to 4GB.

Process control blocks. In order to manage the interleaved execution of user programs and to distribute the systems resources among the tasks, the user-visible and the system information is kept in a table of *process control blocks* (PCBs). The table has a fixed size, equal to the maximum number of supported processes `MAX_PID`. The data structure for a process control block of a process *pid* is declared in Listing 2.1 and has the following components.

Listing 2.1: Process control block and page descriptor

```
1 struct pcb_t {                                  struct pd_t {
2   int          ef[EF_DIM];                        unsigned int pid;
3   unsigned int ihd[IHD_DIM];                      unsigned int vpx;
4   int          bpto;                               unsigned int ppx;
5   int          bpt1;                               struct pd_t* next;
6   int          empty[46u];                         struct pd_t* prev;
7 };                                              };
```

Exception frame. This is the main save area within the process control block. For any *pid* different from the identifier of the current process or when the machine runs in system mode and register save has already completed, the exception frame array holds the values of visible CPU registers. These are thirty one general purpose registers and ten special purpose registers, which — together with thirty two items reserved for floating-point registers — makes in total `EF_DIM` = 73 registers. For each register to be stored, there is a corresponding variable in the PCB. The following registers are not stored in the exception frame:

- Register *gpr*[0] since it permanently has value zero.
- The special purpose *mode* register, because all the processes except the kernel run in user mode of the processor.
- The *emode* register since it is only needed to hold a copy of the *mode* register before a context switch from system to user mode of the CPU. Since in CVM there is no interrupt nesting and thereby it is a binary decision between the two aforementioned modes of the processor.
- The *sr* register, because as soon as an interrupt occurs, the hardware saves the status register *sr* into the exception status register *esr*. From there it is afterwards restored. Keep in mind that in CVM there is no interrupt nesting. During the system mode execution all maskable interrupts are masked by setting *sr* to zero.

- The *esr* register since its the kernel has a variable `SR` to store the shared status-register value.

Interrupt handlers. An array of `IHD_DIM` = 6 unsigned integers is reserved for handlers of internal interrupts / user-defined signals.

Big-page table origin. It stores the index in the big-page table space indicating the start of the big-page table for process *pid*.

Big-page table length. It stores the length of the big-page table for process *pid*.

Empty array. The empty array at the end of a PCB is used to push the PCB size to half a megabyte. By that we make starting addresses of processes inside the PCB array to be powers of two, which allows efficient address computations with logical shift operations.

Of particular relevance for the page fault handling and memory management from all the data contained in the process control block are the exception frame elements at indices `PTO` = 72 and `PTL` = 73 as well as `bpto` and `bptl` fields. An important remark concerning the page table length register is the fact that it also distinguishes between an active and an inactive task. A value of -1 denotes the task is inactive whereas any non-negative value denotes an active process.

Page table space and big-page table space. The page tables of the active tasks are stored in a region called *page table space*. It is important that they must not overlap and they must be page-aligned. In our system, the total virtual memory `TOT_PGS` is limited by 4GB and the page size is 4KB. For each allocated virtual page there must exist a corresponding page table entry in the page table space. Without wasting any space, we would need to store 2^{20} page table entries, each 4-byte long. This will take up $4 \cdot 2^{20}/2^{12} = 1024$ pages of physical memory. However, due to the page alignment of the page tables, there is a maximum waste of one page per process. Since our system supports `MAX_PID` processes, the total waste is at most 128 pages. As a result, the accumulated size, including the potential waste, of the page table space is $1024 + 128 = 1152$ pages of physical memory.

With the above considerations, we declare the page table space as a two-dimensional array on the heap `pt`[`TOT_PGS_PT`][`PTES_PER_PG`] with dimensions:

- `TOT_PGS_PT` = 1152 which represents the upper bound on the number of pages occupied by the page table space, and
- `PTES_PER_PG` = 1024 which represents the number of page table entries contained in a page.

Let us now estimate the memory requirements for the big-page tables. The big-page tables of active tasks are stored in a region called *big-page table space* and they must also not overlap.

Generally speaking, for every big-page of virtual memory there must exist a corresponding big-page table entry in the big-page table space. With a total virtual memory `TOT_PGS` limited by 4GB and a big-page size of 4MB, we would consequently need $2^{32}/2^{22} = 1024$ big-page table entries, each 4-byte long.

However, there is a difference with respect to the granularity of the virtual memory and swap memory: while the first is partitioned in pages, the latter is partitioned in big pages. Therefore, to store the virtual memory on the swap file, we would waste a maximum one swap memory big-page per process. Since our system supports `MAX_PID` processes, the total waste cumulates to at most 128 swap memory big-pages. As a result, we set the total number of swap memory big-pages `TOT_BIG_PGS` to 1024+128 = 1152 and define big-page table space as one-dimensional array of this size: `bpt`[`TOT_BIG_PGS`].

User memory management. Starting from the page with index `KERNEL_PGS`, the user memory continues to the upper end of the physical memory. We define simple structures, called *page descriptors*, in order to manage the user pages (see 2.1). One such page descriptor `pd` corresponds to exactly one physical page index of the user memory and has the following elements:

- a process identifier `pid` and a virtual page index `vpx` to identify to which process and to which virtual page a physical page belongs to if it is used,
- a physical page index `ppx` which points to the user page in the physical memory, and
- pointers `next` and `prev` to organize doubly-linked lists of page descriptors.

Each page descriptor is associated with a physical page belonging to the user memory. These pages fall into two categories.

- User memory pages that store a virtual page. These pages, as well as their associated page descriptors, are called *active*.
- *Free* pages of user memory. Their page descriptors are also called *free*.

Both categories are managed using lists of page descriptors. The active pages are maintained in a so-called *active list*, whereas the free pages are maintained in a *free list*. The implementation variables for these lists are `active` and `free`, respectively.

Swap memory management. The swap memory is allocated to processes in big pages of size 4MB. To accommodate a maximum total virtual memory `TOT_PGS` of 4GB, there must be, as previously pointed out, `TOT_BIG_PGS` big-pages of swap memory.

To manage the swap memory, we use a stack of free big-pages `bpfree`. This is realized by means of a statically defined array of size `TOT_BIG_PGS` which is always accessed at index `bpages_free`−1, the free big page stack pointer. The stack consists in fact only of the array entries with index lower or equal than the stack pointer. Each entry on the stack will contain the (unique) index of a free swap memory big-page.

Initially, all the swap big-pages are free, so the stack `bpfree` will hold all the swap big page indices from 0 to `TOT_BIG_PGS`. At allocation, when an extra swap memory big-page is needed to store a virtual big page, the big page table entry corresponding to the virtual big page is assigned the value of the topmost entry in `bpfree` and then the stack pointer is decremented. Conversely, at deallocation, the stack pointer is incremented and then the swap memory big

page index which is stored in the big-page table entry of the released virtual big page is assigned to the topmost entry of `bpfree`.

Please keep in mind that the stack pointer will always be non-negative. This is ensured by setting `TOT_BIG_PGS` sufficiently big with respect to the total virtual memory size of our system.

Reverse lookup array. When a process voluntarily releases (portions of) its virtual memory, the user memory pages that possibly used to store the released virtual pages should also be freed. Therefore, we must remove all the physical page descriptors associated with these user pages from the active list. Normally, we would have to search the entire list for a page descriptor with a specific physical page index, which is inefficient. In order to avoid the search time we introduce an array storing pointers to all page descriptors. This array will be used as a reverse lookup table, i.e., for associating a physical page index to a page descriptor. We declare array `struct pd* ppx2pd[TOT_PHYS_PGS]` which for all user pages with indices $ppx \in \{\texttt{KERNEL\_PGS}, \ldots, \texttt{TOT\_PHYS\_PGS}\}$ maintains that `ppx2pd`[ppx] points to a page descriptor with the physical page index field of value ppx.

Note that we will only need to manipulate page descriptors associated to user pages. Therefore, the first `KERNEL_PGS` elements of the `ppx2pd` array do not store valid pointers to page descriptors. In fact, they are never initialized.

Miscellaneous variables. The system maintains also the following variables: the first two are used by the page-fault handler and memory management primitives whereas the remaining three are used only by other parts of the CVM code.

`pages_used`	Number of currently used virtual pages.
`pages_free`	Number of currently free physical pages.
`SR`	Devices interrupts and overflow mask.
`cup`	Current user process identifier.
`kheap`	Kernel heap pointer.

2.5.2 Execution Scenarios

As mentioned above, two page descriptor lists are used by the page-fault handler, namely the active list and the free list. Elements of the free list correspond to physical pages in user memory that are currently not used by any process. At startup, the free list contains all user memory pages. Complementary, elements of the active list correspond to allocated user pages. Every element of the active list is associated with physical page index `ppx`, process identifier `pid`, and virtual page index `vpx`. For such an element in the active list, the page table entry of `vpx` which belongs to task `pid` is valid and points to `ppx`. The order of the elements in the active list is significant, in fact it is used as a queue: elements are created at its tail on swap in, and they are taken away from its head on swap out. Hence, the page fault handler implements a FIFO scheme, that always selects the oldest virtual page in user memory for eviction (swap out).

Page faults are exceptions to the software raised by the hardware in cases described by Definitions 2.7 and 2.8. On the software layer, nevertheless, these

hardware page page faults receive different interpretations. This depends on the complexity of the handler and the features it supports. For instance, a page fault caused by a memory access to a page not loaded in main memory might be identified by dissimilar conditions in two distinct page fault handlers. As a result, we call *software page faults* the collection of situations which a page fault handler is able to treat. At the basis of any software page fault stands a hardware page fault. Our page-fault handler can handle two software page faults.

1. The *invalid access page fault*. It occurs when the accessed virtual page does not reside in physical memory signal by the valid bit unset.

2. The *zero protection page fault*. It occurs on the very first user write operation on a virtual page, after the allocation. Additional conditions for it are that the protection bit is set and the physical page index in the associated page table entry pointing to `ZFP`.

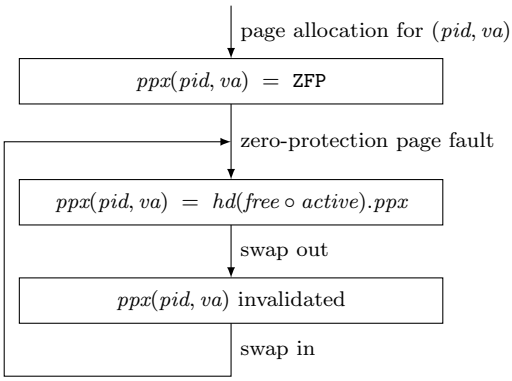
At allocation, virtual pages point to the zero-filled page and both the valid and protected bit in their page table entries are set. Pages that map to the zero-filled page do not have a page descriptor in the active list nor in the free list. The physical rights of the newly allocated pages allow for user read access. However, as soon as there will be a user write access on them, this will generate a zero protection fault. Hence, one page of the user memory must be filled with zeros and its associated page descriptor must be included in the active list. This implements actually the so-called *copy-on-write* principle. On swap in, pages are moved from the free list to the active list due to subsequent page faults to the zero protection fault. Observe that now the protection bit is not set in the respective page table entry since the only possible page fault from now on is the invalid access page fault. On swap out, the victim pages are moved from the active list back to the free list. The page table entry corresponding to the swapped-out virtual page is entirely cleared. Pages will be also moved from the active to the free list in case of memory release. But in this case, the page table entries will not get explicitly cleared. They simply become inaccessible due to the decrease of the page table length.

To offer a better understanding of the page-fault handler we depict in Figure 2.1 its mechanism drawing attention to all possible page descriptor movements between lists together with the corresponding conditions and effects on the page table entries.

The page-fault handler is implemented as a function `pfh_touch_addr` which receives four input arguments: the process identifier `pid`, the exception virtual address `addr`, the intention of a call to the handler `intent`, and the number `count` of calls to the handler during which the page specified by `pid` and `addr` must survive in the physical memory. The page-fault handler returns (in a non-error case) a translated physical address for the pair `(pid,vpx)` represented as an unsigned integer.

The function `pfh_touch_addr` serves as a single entry point for all cases where the kernel *has or might have to* swap in a page. That means the function `pfh_touch_addr` is invoked not only to handle user page faults but also to perform all (software) address translations in the kernel implementation. We distinguish four different intentions `intent` of a handler's call: (i) `READ`: the page in question should be readable afterwards, (ii) `WRITE`: the page in ques-

Figure 2.1: Page movements by the page-fault handler



tion can afterwards be read and written, (iii) `OVERWRITE`: the caller intends to overwrite the entire page after the call, and (iv) `SWAP_IN`: a page fault has occurred at the given address and the function will swap in the page for arbitrary accesses. We describe the page-fault handler implementation in detail in Section 5.6 where we give additional information on the role of its parameters `intent` and `count`. Below we describe the handler functionality in case a user page fault happens for the pair `pid` and `addr`.

The execution of the page-fault handler can be divided into three steps:

- First, we decide whether the exception operation was legal or illegal. By convention, an operation is illegal if we have a page table length exception. If this is the case, we abort the execution of the handler and exit with the code `INVALID_ADDR`. Otherwise, we continue.

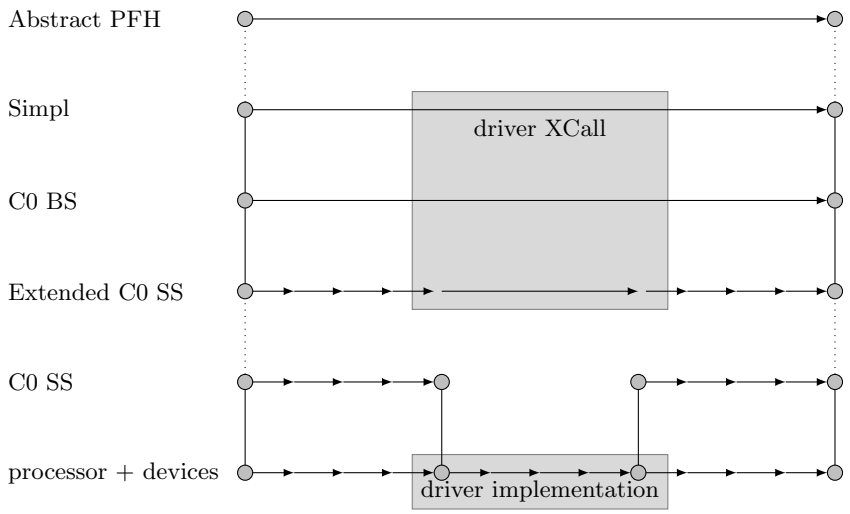
- Second, we determine the physical page index in the user memory that we intend to use for swap in. At this point, there are two possibilities for choosing this index. If there are still unused free pages in the user memory, we take one of their indices. Otherwise, we select and swap out one of the pages in user memory. By the first-in-first-out (FIFO) strategy we select for eviction the page associated with the head of the active list.

- Third, we check whether the exception page already lies in the physical memory, i.e., if the valid bit of the corresponding page table entry is set. If so, a zero protection page fault took place and therefore, we fill the target page determined in the second step with zeros. If the valid bit is not set, we simply swap in the exception page. Finally, we update the page table entry corresponding to the exception page accordingly.

2.5.3 Approach to Correctness Proof

So far we have presented the problem of virtual memory simulation, introduced a correctness statement for this problem — the CVM correctness theorem, and elaborated how demand paging is implemented in CVM in order to support virtual memory. The goal of the remainder of this thesis is to formulate and

Figure 2.2: Approach to prove correctness of demand paging



prove correctness theorems of demand paging such that they could be applied in the context of CVM's proof.

The target semantics of the CVM correctness theorem (Theorem 2.33) is VAMP ISA. The demand paging is implemented in C0 with calls to hard-disk drivers which contain assembly code. We might have verified this implementation at the level of C0 small-step semantics and using the simulation theorem between C0 and VAMP ISA project the obtained result to the hardware level. However, proving the correctness of programs of the size and complexity of the demand paging implementation is a big effort. In order to ease this problem we introduce a so-called *C0 semantics stack* in the next chapter (Chapter 3). The stack provides nice means to prove correctness of implementations not in the C0 small-step semantics but rather on a more abstract level of the language *Simpl* in the Hoare logics. The correctness results obtained on the level of Simpl could be transfered — via an intermediate level of C0 big-step semantics — down to C0 small-step semantics level. Moreover, the stack handles routines with inline assembly portions by abstracting them to extended calls (XCalls).

The overall approach to prove correctness of demand paging is shown at Figure 2.2. We first specify and prove all necessary and sufficient properties in terms of the PFH automaton as discussed further in chapter 4. By proving in Hoare logic that the abstraction mapping holds in the state after executing the page-fault handler provided it holds before, we obtain the desired properties at the level of Simpl (Chapter 5). We map these results — via the C0 big-step semantics — to the level of the (extended) C0 small-step semantics. We justify the XCalls to the hard disk driver by plugging in their correctness result [Alk09]. Next, with the help of the simulation theorem of C0 by ISA we are able to state the page-fault handler functional correctness in terms of the ISA semantics. This allows us to prove the page-fault handler top-level theorem. The last three steps are described in Chapter 7.

CHAPTER 3

Leveraging a Semantics Stack

In order to support pervasive verification of system software a stack of semantics for the C0 language has been carefully crafted in Verisoft. The C0 semantics stack comprises a Hoare logic, a big-step semantics, and a small-step semantics. By a higher level of abstraction in the Hoare logic compared to the small-step semantics, we gain efficiency for the verification of individual C0 programs. However, we have to integrate the results obtained in the Hoare logic into our systems stack. Certain simulation theorems allow to transfer program properties from the Hoare logic down to the small-step semantics. In this chapter we first introduce Simpl, a generic imperative language for which a Hoare logic was defined. In the previous chapter we have already defined the C0 small-step semantics, therefore, it remains only to introduce only the big-step semantics to cover all layers of the semantics stack. Finally, we present simulation theorems between the stack's layers. They will allow us to transfer correctness properties of programs between the layers.

The Hoare logic provides sufficient means to reason about pre- and postconditions of sequential, type-safe, and assembly-free C0 programs. The small-step semantics level allows integration with inline assembly code. The big-step semantics is a bridging layer, which is convenient to express results of the Hoare logic operationally. The core differences of the semantical layers of C0 can be summarized as follows:

- Hoare logic: split heap, compound values, implicit typing.
- Big step: single monolithic heap, compound values, explicit typing.
- Small step: single monolithic heap, flat values, explicit typing.

These design decisions reflect the purpose of the layers. The Hoare logic is tuned to support verification of individual programs, whereas the small-step semantics is nearer to the architecture level.

The levels below C0 allow integration of devices, or communication between memory parts which do not belong to the program range. Our approach is to abstract effects of those low-level computations into atomic *XCalls* (extended calls) in all semantics layers. The state space of C0 is augmented with an additional component that represents the state of the external component, e.g., the device or parts of physical memory. An XCall is a procedure call that makes a transition on this external state and communicates with C0 via parameter passing and return values. With this model it is straightforward to integrate XCalls into the semantics and into Hoare logic reasoning. Bodies of XCalls are typically implemented in assembly. An implementation proof of this piece of assembly justifies the abstraction to an atomic XCall. In this chapter we introduce individual layers of the stack as well as equivalence theorems between them.

In order to reason about programs on each of these levels a program's source code has to be correspondingly represented in Isabelle/HOL. Such representations are generated automatically by a translation tool developed in the frame of this work.

While discussing Simpl, big-step semantics, and their equivalence we follow the doctoral thesis of Schirmer [Sch06]. Our presentation of Hoare logic is based on the doctoral thesis of Petrova [Pet07]. The section about equivalence between big-step and small-step semantics is guided by a JAR article [AHL+09].

3.1 Simpl

Simpl is a rather general sequential imperative programming language that is not fixed to a specific real programming language like C, Pascal or Java. It is more like a model for imperative programs that allows to embed real programming languages for the purpose of program verification. Simpl makes no assumptions on the state space, it is polymorphic. We will denote the state space by type variable C_{H} which is at the end instantiated with a program-depended record. Basic actions like variable assignments or memory allocation are arbitrary state updates of type $C_{\mathrm{H}} \mapsto C_{\mathrm{H}}$.

Abstract syntax. Commands of Simpl are defined by the polymorphic data type $com(C_{\mathrm{H}})$ parametrized over the state space:

Skip	Do nothing.
Basic(f)	Basic command; $f :: C_{\mathrm{H}} \mapsto C_{\mathrm{H}}$ is a state update.
Seq(s_1, s_2)	Sequential composition; s_1 and s_2 are of type $com(C_{\mathrm{H}})$.
Cond(b, s_1, s_2)	Conditional statement; s_1, $s_2 :: com(C_{\mathrm{H}})$ and boolean condition b is modeled as a state set of type $pow(C_{\mathrm{H}})$.
While(b, s)	Loop; $s :: com(C_{\mathrm{H}})$ and $b :: pow(C_{\mathrm{H}})$.
Call(p)	Procedure call; p is a procedure name of type $\mathbb{S}$.
Guard(g, s)	Guarded command; $g :: pow(C_{\mathrm{H}})$ and $s :: com(C_{\mathrm{H}})$.

State. The core state space is polymorphic and denoted by type variable C_{H}. It includes the extended state of XCalls as an ordinary record component. In order to define semantics the state space is augmented with an error state $\bot$, i.e., the augmented state space is $C_{\mathrm{H}\bot}$. Executions start in some normal state $\lfloor c_{\mathrm{H}} \rfloor$. In case a guard *Guard*(g, s) is violated the runtime fault is signaled by the state $\bot$. Moreover, execution can get stuck because of a call to an undefined procedure.

Semantics. The operational semantics of Simpl

$$\Gamma \vdash_{\mathrm{H}} \langle s, c_{\mathrm{H}} \rangle \Rightarrow c'_{\mathrm{H}}$$

is defined inductively by the rules in Figure 2.1 of Schirmer's thesis[1]. It has a meaning that in procedure environment Γ execution of command s transforms the initial state c_{H} to the final state c'_{H}, where $\Gamma :: \mathbb{S} \mapsto com(C_{\mathrm{H}})$, $c_{\mathrm{H}}, c'_{\mathrm{H}} :: C_{\mathrm{H}\bot}$, and $s :: com(C_{\mathrm{H}})$.

XCalls are specified as basic commands *Basic*(f) where the function $f :: C_{\mathrm{H}} \mapsto C_{\mathrm{H}}$ updates the extended state component of C_{H} (and possibly some over variables).

Termination. To verify total correctness of a program one needs to show that the program terminates for all valid inputs. To guarantee termination of a Simpl program it is not sufficient to require the existence of a terminating computation: $\exists\, c'_{\mathrm{H}} : \Gamma \vdash_{\mathrm{H}} \langle s, c_{\mathrm{H}} \rangle \Rightarrow c'_{\mathrm{H}}$. Due to nondeterminism this does not guarantee that all computations from the same initial state s terminate. Guaranteed termination

$$\Gamma \vdash_{\mathrm{H}} s \downarrow c_{\mathrm{H}}$$

of program s in the initial state c_{H} is defined inductively by the rules in Figure 2.2 of Schirmer's thesis. If statement s terminates when started in state c_{H}, then there exists final state c'_{H} with respect to the semantics:

$$\Gamma \vdash_{\mathrm{H}} s \downarrow c_{\mathrm{H}} \quad \longrightarrow \quad \exists\, c'_{\mathrm{H}} : \Gamma \vdash_{\mathrm{H}} \langle s, c_{\mathrm{H}} \rangle \Rightarrow c'_{\mathrm{H}}.$$

The other direction is not valid since Simpl is nondeterministic — the execution branch that is taken depends on the particular input data.

Heap and pointers. The heap model we use excludes explicit address arithmetic but it is capable to represent typical heap structures like lists. We follow the split heap approach that goes back to Burstall [Bur72] and was recently

[1] Schirmer uses just a turnstile symbol $\vdash$. We extend it with subscript "H" to stress that it corresponds to Hoare logic as we also will have versions for big- and small-step semantics, signaled by subscripts "BS" and "SS", respectively.

taken up by Mehta and Nipkow [MN03]. The main benefit of this heap model is that it already excludes aliasing between between pointers of unequal type or to different structure fields. The typed view of memory is hard-wired into the model. That is why it is not possible to properly express low-level untyped operations like pointer arithmetic in it. To highlight that we do not calculate with pointers we introduce a type *ref-t* of references. It is isomorphic to the natural numbers. We declare the reference `NULL` as a constant without any definition, it is just one value upon the references. To model allocation and deallocation we need some bookkeeping of allocated references. We achieve this by introducing a list of allocated references *alloc* to the state space. To model the allocation of a new reference we use the function $new :: pow(\textit{ref-t}) \mapsto \textit{ref-t}$, which for a set of references A gives a fresh reference $a = new(A)$ with condition $a \notin \{\texttt{NULL}\} \cup A$. Since type *ref-t* is isomorphic to the natural numbers we have infinitely many references.

3.2 Hoare Logic

Hoare triples. The Hoare logic for Simpl is inductively defined by the rules in Figure 3.1 of Schirmer's thesis. The judgment for partial correctness has the following form:

$$\Gamma \vdash_{\mathrm{H}} P\, s\, Q.$$

The intended formal semantics of a Hoare triple is defined by the notion of validity:

$$\begin{aligned}\Gamma \vDash_{\mathrm{H}} P\, s\, Q \quad = \quad & \forall\, c_{\mathrm{H}}, c'_{\mathrm{H}} : \Gamma \vdash_{\mathrm{H}} \langle s, c_{\mathrm{H}}\rangle \Rightarrow c'_{\mathrm{H}} \\ & \wedge\, c_{\mathrm{H}} \in \{\lfloor x \rfloor : x \in P\} \longrightarrow c'_{\mathrm{H}} \in \{\lfloor x \rfloor : x \in Q\}.\end{aligned}$$

Given an execution of statement s from initial state c_{H} to final state c'_{H}, provided that c_{H} is some state satisfying the precondition P, then c'_{H} becomes a state satisfying Q. Validity and the Hoare logic are related by two important theorems:

- *Soundness*: $\Gamma \vdash_{\mathrm{H}} P\, s\, Q \;\longrightarrow\; \Gamma \vDash_{\mathrm{H}} P\, s\, Q$, and
- *Completeness*: $\Gamma \vDash_{\mathrm{H}} P\, s\, Q \;\longrightarrow\; \Gamma \vdash_{\mathrm{H}} P\, s\, Q$.

Total correctness means partial correctness plus termination. This is directly reflected in the validity notion for total correctness:

$$\Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} P\, s\, Q \quad = \quad \Gamma \vdash_{\mathrm{H}} P\, s\, Q \wedge \forall\, c'_{\mathrm{H}} \in \{\lfloor x \rfloor : x \in P\} : \Gamma \vdash_{\mathrm{H}} s \downarrow c'_{\mathrm{H}}.$$

The various judgments for total correctness are distinguished from partial correctness by the superscript "t". The total correctness Hoare logic for Simpl is inductively defined by the rules in Figure 3.3 in the thesis of Schirmer. The judgments has the following form:

$$\Gamma \vdash^{\mathrm{t}}_{\mathrm{H}} P\, s\, Q.$$

Here we have also soundness and completeness:

$$\Gamma \vdash^{\mathrm{t}}_{\mathrm{H}} P\, s\, Q \;\longrightarrow\; \Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} P\, s\, Q \qquad \Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} P\, s\, Q \;\longrightarrow\; \Gamma \vdash^{\mathrm{t}}_{\mathrm{H}} P\, s\, Q.$$

Keeping track of modified global and heap variables. Let us interpret sets as predicates, i.e., $P(x)$ means $x \in P$. We extend Hoare triples and judgments

for them with the following notation

$$\Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} P(c_{\mathrm{H}})\ s\ Q(c'_{\mathrm{H}}) \cap \Delta(c_{\mathrm{H}}, c'_{\mathrm{H}}) = \{a, b, c, \ldots\}.$$

It has a meaning that during the execution of a statement s, i.e., a transition from c_{H} to c'_{H}, *only* global or heap variables a, b, c, etc. were modified.

VCG: Verification condition generator. Since the Hoare logic is defined inductively by the rules for every language constructor, we can apply them backwards to a statement in order to decompose it to the atomic ones (*Skip* and *Basic*).

The idea of the *verification condition generator* (VCG) is the following: for judgment a $\Gamma \vdash_{\mathrm{H}} P\ s\ Q$ that we want to prove, we automatically apply the Hoare rules until program s is completely eliminated and a purely logical proof claim $P \subseteq WP$ remains. WP is a *weakest precondition* that is computed from the given postcondition Q by backward rules application. So, the weakest precondition is a set of states which has only the properties, which are necessary in order to guarantee, that execution of the program will end in a state, where the postconditions hold.

In order to be able to apply all the rules automatically we need a version of the rules that can be applied to any Hoare triple and can compute its weakest precondition. Therefore for every language statement we provide a rule of the following format:

$$\frac{P \subseteq WP \qquad T_1, \ldots, T_i}{\Gamma \vdash_{\mathrm{H}} P\ s\ Q},$$

where $T_1, \ldots, T_i$ are side conditions needed to compute the weakest precondition WP. Moreover, for each modified rule there exist a proof, that it can be deduced from the original ones.

- Let us consider transformation of the original rule for *Basic* as an example:

  $$\frac{}{\Gamma \vdash_{\mathrm{H}} \{c_{\mathrm{H}} | f(c_{\mathrm{H}}) \in Q\}\ \mathit{Basic}(f)\ Q}$$

  The postcondition can be actually applied to any state, the precondition is the weakest precondition for *Basic* statement, but we can not expect that the triple we want to prove exactly matches the precondition. Thus, we transform it to the the following one:

  $$\frac{P \subseteq \{c_{\mathrm{H}} | f(c_{\mathrm{H}}) \in Q\}}{\Gamma \vdash_{\mathrm{H}} P\ \mathit{Basic}(f)\ Q}$$

  This rule has an appropriate format and can be applied to any *Basic* statement.

- The rule for sequential composition combines pre- and postconditions for both sub-statements, where R is the weakest precondition for Q and s_2:

  $$\frac{\Gamma \vdash_{\mathrm{H}} P\ s_1\ R \qquad \Gamma \vdash_{\mathrm{H}} R\ s_2\ Q}{\Gamma \vdash_{\mathrm{H}} P\ \mathit{Seq}(s_1, s_2)\ Q}$$

- For the conditional statement the following rule will be used by the VCG:

  $$\frac{P \subseteq \{c_{\mathrm{H}} | (b(c_{\mathrm{H}}) \longrightarrow c_{\mathrm{H}} \in P_1) \wedge (\neg b(c_{\mathrm{H}}) \longrightarrow c_{\mathrm{H}} \in P_2)\} \qquad \Gamma \vdash_{\mathrm{H}} P_1\ s_1\ Q \qquad \Gamma \vdash_{\mathrm{H}} P_2\ s_2\ Q}{\Gamma \vdash_{\mathrm{H}} P\ \mathit{Cond}(b, s_1, s_2)\ Q}$$

If P_1 and P_2 are the weakest preconditions for both branches s_1 and s_2, then the weakest precondition for the conditional combines them with the value of the branching condition b:

$$\{c_{\mathrm{H}} | (b(c_{\mathrm{H}}) \longrightarrow c_{\mathrm{H}} \in P_1) \wedge (\neg b(c_{\mathrm{H}}) \longrightarrow c_{\mathrm{H}} \in P_2)\}.$$

So, if state s satisfies the condition b, then the precondition P_1 have to hold in it; and if c_{H} does not satisfy b, then it has to belong to P_2.

- For handling loops we need a rule which allows us to introduce an invariant. Since it cannot be computed by the rules from a while loop, it must be provided by the user. The statement $\mathit{WhileI}(I, b, s)$ introduces a while loop with the annotated invariant. Since the invariant does not have any influence on the deduction, it is semantically defined as a simple while loop $\mathit{WhileI}(I, b, s) = \mathit{While}(b, s)$. We need the invariant annotation only for the rule for the VCG:

$$\frac{P \subseteq I \qquad \Gamma \vdash_{\mathrm{H}} (I \cap \{c_{\mathrm{H}} | b(c_{\mathrm{H}})\})\ s\ I \qquad I \cap \{c_{\mathrm{H}} | \neg b(c_{\mathrm{H}})\} \subseteq Q}{\Gamma \vdash_{\mathrm{H}} P\ \mathit{WhileI}(I, b, s)\ Q}$$

  To prove a triple for the while loop by deduction we have to show three subgoals: (i) the precondition P must imply the invariant I, (ii) the invariant is maintained while the loop is being executed, and (iii) the invariant and the negated loop condition must imply the postcondition Q.

- The idea of the rule for a procedure call is to reduce the goal to proving correctness of the procedures body with pre- and postconditions P' and Q', respectively, such that P' is weaker than P whereas Q' is stronger than Q:

$$\frac{P \subseteq P' \qquad \Gamma \vdash_{\mathrm{H}} P' \text{ body of } p\ Q' \qquad Q' \subseteq Q}{\Gamma \vdash_{\mathrm{H}} P\ \mathit{Call}(p)\ Q}$$

The described verification condition generator is implemented in Isabelle/HOL by Schirmer [Sch06].

3.3 BS: C0 Big-Step Semantics

Values. The address model for C0 big-step semantics is rather abstract. No assumptions about data-alignment or consecutive addresses are made. One location can store any kind of value, even structured ones. This is quite similar to references in Simpl as introduced in Section 3.1. For a convenient translation of C0 addresses to Simpl we introduce type *loc-t* of locations with non-`NULL` references *ref-t*:

$$\textit{loc-t} \quad = \quad \{r :: \textit{ref-t} \mid r \neq \texttt{NULL}\}.$$

Primitive C0 values modeled by data type *prim-t* can be:

$\mathit{Bool}(b)$	A Boolean $b :: \mathbb{B}$.	$\mathit{Chr}(c)$	A character $c :: \mathbb{Z}$.
$\mathit{Intg}(i)$	A (signed) integer $i :: \mathbb{Z}$.	$\mathit{Addr}(a)$	A reference $a ::$ *loc-t*.
$\mathit{Unsgnd}(n)$	An unsigned integer $n :: \mathbb{N}$.	*Null*	The null reference.

C0 values modeled by data type *val-t* can be:

Prim(*p*)	Primitive values, where $p :: \textit{prim-t}$.
Arr(*vs*)	Arrays, where $vs :: \textit{val-t}^*$.
Struct(*fs*)	Structures, where $fs :: (\mathbb{S} \times \textit{val-t})^*$.

Since there is no extra layer of values in Simpl, `NULL` is an ordinary element of type *ref-t*, whereas in C0 *Null* is an extra constructor of values. The definition of type *loc-t* allows us to map value *Null* to reference `NULL` since it can not be occupied by a C0 address. The function $\textit{Ref} :: \textit{ref-t} \mapsto \textit{val-t}$ converts a reference to a value:

$$\textit{Ref}(r) = \begin{cases} \textit{Prim}(\textit{Null}) & \text{if } r = \texttt{NULL} \\ \textit{Prim}(\textit{Addr}(\textit{ref2loc}(r))) & \text{otherwise.} \end{cases}$$

Function *ref2loc*(r) converts a reference to a reference. Since types for both of them are just aliases for the natural numbers the conversion is identity. The same is true for *loc2ref*(l).

Programs. Although in Isabelle/HOL there is a slight technical difference in representation of C0 programs at the level of big-step and small-step semantics, in this thesis we will use the previously defined type *prog-t* to model programs at the BS level.

A big-step program might contain extended calls (XCalls) to some procedures which affect the extended state configuration $c_{\text{X}} :: C_{\text{X}}$. Semantics of extended procedures is defined in an *extended semantics environment* modeled by the type *xsem-t*:

$$\textit{xsem-t} = (\mathbb{S} \times (\mathbb{S} \times \textit{ty-t})^* \times \textit{ty-t}^* \times ((\mathbb{S} \mapsto \textit{val-t}_\perp) \times C_{\text{X}} \mapsto (\textit{val-t}^* \times C_{\text{X}})_\perp))^*.$$

It consists of (i) a procedure name, (ii) list of parameter names and types, (iii) list of return types, and (iv) a function that takes a parameter environment (partial mapping from parameter names to values) and an extended state, and returns (if possible) a pair of return values and updated extended state.

State. States c_{BS} of C0 big-step semantics are modeled by records of type C_{BS} which have the following fields.

$c_{\text{BS}}.\textit{gvars}$	Global variables, a mapping $\mathbb{S} \mapsto \textit{val-t}_\perp$ from names to values.
$c_{\text{BS}}.\textit{heap}$	The heap, a mapping $\textit{loc-t} \mapsto \textit{val-t}_\perp$ from locations to values.
$c_{\text{BS}}.\textit{lvars}$	Local variables modeled the same way as global.
$c_{\text{BS}}.\textit{free}$	The counter for the amount of free heap memory.
$c_{\text{BS}}.x$	The placeholder for the extended state of XCalls.

Expression evaluation. Local variables may hide global variables. Throughout execution we keep track of the local variables via a set L to decide whether a name refers to a local or global variable. This set is also used as a parameter for the expression evaluation. Expression evaluation $\textit{eval}_{\text{BS}}(L, c_{\text{BS}}, e)$ of expression $e :: \textit{expr-t}$ in state $c_{\text{BS}} :: C_{\text{BS}}$ and in context of local variables $L :: \textit{pow}(\mathbb{S})$ is defined in Figure 7.1 of Schirmer's thesis [Sch06]. The function produces a result of option type $\textit{val-t}_\perp$, which models possible run-time faults. If the context of local variables is irrelevant we will write $\textit{eval}_{\text{BS}}(c_{\text{BS}}, e)$ for $\textit{eval}_{\text{BS}}(\emptyset, c_{\text{BS}}, e)$.

Big-step semantics. The operational *big-step* semantics defined by judgment

$$\Pi, \mathit{xsem}, L \vdash_{\mathrm{BS}} \langle s, c_{\mathrm{BS}} \rangle \Rightarrow c'_{\mathrm{BS}}$$

means that with respect to a program Π, extended semantics environment *xsem*, and context of local variables L execution of statement s transforms the initial state c_{BS} to the final state c'_{BS}, where $\Pi :: \mathit{prog}_{\mathrm{BS}}$, $\mathit{xsem} :: \mathit{xsem\text{-}t}$, $L :: \mathit{pow}(\mathbb{S})$, $c_{\mathrm{BS}}, c'_{\mathrm{BS}} :: C_{\mathrm{BS}\,\bot}$, and $s :: \mathit{stmt\text{-}t}$. The semantics is defined in the thesis of Schirmer [Sch06] in Figulre 7.7.

The *termination-guaranteed* judgment for C0 programs

$$\Pi, \mathit{xsem}, L \vdash_{\mathrm{BS}} s \downarrow c_{\mathrm{BS}}$$

of statement s in the initial state c_{BS} within the context of program Π, extended semantics environment *xsem*, and local variables L is defined inductively by the rules in Figure 7.8 of Schirmer's thesis [Sch06]. A crucial property is the following equivalence between termination and the big-step semantics:

$$\Pi, \mathit{xsem}, L \vdash_{\mathrm{BS}} s \downarrow c_{\mathrm{BS}} \longleftrightarrow (\exists\, c'_{\mathrm{BS}} : \Pi, \mathit{xsem}, L \vdash_{\mathrm{BS}} \langle s, c_{\mathrm{BS}} \rangle \Rightarrow c'_{\mathrm{BS}}).$$

Hoare triples. Now we define the notion of a valid Hoare triple for C0 big-step semantics analogously to validity in Simpl. We start with partial correctness:

$$\begin{aligned} \Pi, \mathit{xsem}, L \vDash_{\mathrm{BS}} P\ s\ Q \quad = \quad & \forall\, c_{\mathrm{BS}}, c'_{\mathrm{BS}} : \Pi, \mathit{xsem}, L \vdash_{\mathrm{BS}} \langle s, c_{\mathrm{BS}} \rangle \Rightarrow c'_{\mathrm{BS}} \\ & \wedge\, c_{\mathrm{BS}} \in \{\lfloor x \rfloor : x \in P\} \longrightarrow c'_{\mathrm{BS}} \in \{\lfloor x \rfloor : x \in Q\}. \end{aligned}$$

Given an execution of statement s from initial state c_{BS} to final state c'_{BS}, provided that the initial state satisfies the precondition P then the execution of s does not cause a runtime fault and the final state satisfies the postcondition Q. Total correctness additionally requires termination:

$$\begin{aligned} \Pi, \mathit{xsem}, L \vDash^{\mathrm{t}}_{\mathrm{BS}} P\ s\ Q \quad = \quad & \Pi, \mathit{xsem}, L \vDash_{\mathrm{BS}} P\ s\ Q \\ & \wedge \forall\, c'_{\mathrm{BS}} \in \{\lfloor x \rfloor : x \in P\} : \Pi, \mathit{xsem}, L \vdash_{\mathrm{BS}} s \downarrow c'_{\mathrm{BS}}. \end{aligned}$$

Typing. C0 is a statically typed language. In Simpl the state is already implicitly typed by the translation to a state-record. The correspondence to C0 programs is only guaranteed for well-typed programs.

The judgment

$$HT \vdash_{\mathrm{v}} v :: T$$

expresses that value v is compatible with type $T :: \mathit{ty\text{-}t}$ with respect to an option heap typing $HT :: (\mathit{loc\text{-}t} \mapsto \mathbb{S})_{\bot}$. It is defined inductively by the rules in Figure 7.10 of Schirmer's thesis [Sch06].

The judgment

$$\Pi, VT, HT \vdash s\surd$$

ensures that statement $s :: \mathit{stmt\text{-}t}$ is well-typed with respect to program Π, variable typing $VT :: \mathbb{S} \mapsto \mathit{ty\text{-}t}_{\bot}$, and heap typing $HT :: \mathit{loc\text{-}t} \mapsto \mathbb{S}_{\bot}$. It is defined inductively by the rules in Figure 7.13 of Schirmer's thesis [Sch06].

Definite assignment. The local variables and the result variable of a procedure are not automatically initialized. However, uninitialized variables are a serious threat for a type-safe execution of a program. Here we supply a simple static analysis for the source program that ensures that we assign a value to a variable

before we read from it. The formalization of the definite assignment analysis basically consists of two parts. Function $\mathcal{A}$ calculates the set of variables that are certainly *assigned* to by a piece of code. The test $\mathcal{D}$ that ensures that reading the variable is safe, since it *definitely* was assigned to before. The definite assignment analysis for C0 is defined by the following functions:

for statements	$\mathcal{D}(s, L, A)$,	$\mathcal{A}(s, L)$,
for expressions	$\mathcal{D}_{\mathrm{e}}(e, L, A)$,	
for left-expressions	$\mathcal{D}_{\mathrm{l}}(e, L, A)$,	$\mathcal{A}_{\mathrm{l}}(e, L)$.

Parameter L is the set of local variables and A is the set of assigned variables. The definitions are given in Figure 7.14 of Schirmer's thesis [Sch06].

Well-formed programs. We consider program Π to be *well-formed*, denoted by *wf-prog?*(Π) [Sch06, Definition 7.42] , if it respects the following static conditions.

- Type names in the type environment and the global variables have to be unique and all types have to be well-formed.
- Moreover, all procedure names have to be unique and their definitions have to be well-formed.
- The names of parameters, local variables, and the result variable have to be unique and all their types have to be well-formed.
- The procedure body has to be well-typed with respect to the variable typing obtained from global variables, parameters, local variables, and the result variable.
- Moreover, the body has to pass the definite assignment test, where parameters, local variables, and the result variables are considered as local names and only the parameters are considered as assigned variables.
- Finally, the result variable has to be assigned in the procedure body.

Type safety. Type safety relates the static semantics like typing and definite assignment with the dynamic semantics, the execution of the program. It describes properties that are guaranteed during runtime if the static tests have been passed. Here we are interested in the fact that execution of statements preserves well-typedness or *conformance* of the program state. Conformance of a value to its type is already captured by the typing judgment $\lfloor HT \rfloor \vdash_{\mathrm{v}} v :: T$ where HT is the heap typing for the current state. A *store* like a local variable or the heap is a mapping from variable names $\mathbb{S}$ or locations *loc-t* to option values *val-t*$_\perp$. A store s conforms to static typing ST if every stored value conforms to its type:

$$HT \vdash s :: ST \quad = \quad \forall\, p, v, T : s(p) = \lfloor v \rfloor \wedge ST(p) = \lfloor T \rfloor \longrightarrow \lfloor HT \rfloor \vdash_{\mathrm{v}} v :: T.$$

A BS state is *conforming* if the heap, the local variables, and the global variables conform to the corresponding type environment. For a program state we define conformance predicate $TE \vdash c_{\mathrm{BS}} :: HT, LT, GT$, where $TE :: \mathbb{S} \mapsto$ *ty-t* is a type environment, $HT ::$ *loc-t* $\mapsto \mathbb{S}_\perp$ is a heap typing, and $LT, GT :: \mathbb{S} \mapsto$ *ty-t*$_\perp$ are typings for local and global variables, respectively.

DEFINITION 3.1 ▶
Conforming BS state

$$\begin{aligned} TE \vdash c_{\mathrm{BS}} :: HT, GT, LT \;=\; & HT \vdash c_{\mathrm{BS}}.heap :: (TE \circ_m HT) \\ \wedge\; & dom(HT) \subseteq dom(c_{\mathrm{BS}}.heap) \\ \wedge\; & \mathit{finite}(dom(c_{\mathrm{BS}}.heap)) \\ \wedge\; & HT \vdash c_{\mathrm{BS}}.gvars :: GT \\ \wedge\; & dom(GT) \subseteq dom(c_{\mathrm{BS}}.gvars) \\ \wedge\; & HT \vdash c_{\mathrm{BS}}.lvars :: LT. \end{aligned}$$

Above, by $f \circ_m g$ we denote a composition of partial mappings f and g , by $dom(f) = \{a | f(a) \neq \perp\}$ we denote a domain of partial mapping f , and by $\mathit{finite}(s)$ we denote that a set s is finite. The heap typing HT maps locations to type names, and the type environment TE maps type names to types. The heap has to conform to the composition of both. Moreover, the domain of the heap typing is a subset of the domain of the heap and both are finite. The global variables must conform to their types. The domain of the global variable typing is a subset of the domain of the global variables. Finally, the local variables have to conform to their types.

Sometimes, when we use a conforming BS state notation we might lack a global, local, or heap typing to supply. In such situations we will provide an *empty* typing, i.e., a typing with an empty set as a domain. We will denote such typings by [] sharing the notation with an empty list.

3.4 Property Transfer from Simpl to BS

At the end of a day we want to prove functional Properties and termination of C0 programs. We start with a C0 program, abstract it to Simpl and then verify this Simpl program by means of the Hoare logic. To transfer a Hoare triple from Simpl to C0 we need to know that the behavior of the C0 program is captured by the behavior of the Simpl program. Therefore, we have to prove that the C0 program can be simulated by the corresponding Simpl program. In the context of total correctness we also have to show that termination of the Simpl program implies termination of the corresponding C0 program.

A peculiarity of the translation from C0 to Simpl is that it cannot be defined generically for all C0 programs since variables in Simpl are represented as record fields. Therefore the shape of the record depends on every individual program.

State abstraction. To abstract the state we use the function

$$BS2H_{\mathrm{state}} :: C_{\mathrm{BS}} \mapsto pow(C_{\mathrm{H}}).$$

It takes a big-step state and yields the set of all related Simpl states of type C_{H}. Clearly, this function could only be defined for a particular program state space. We define the function for the state of the demand paging implementation in Section 6.2 (Defintion 6.19).

Program abstraction. An abstraction of C0 statements in the context of a particular procedure is defined by function

$$BS2H_{\mathrm{stmt}} :: \mathit{stmt\text{-}t} \mapsto com(C_{\mathrm{H}}).$$

It is introduced in Definition 8.46 and Figure 8.4 in Schirmer's thesis [Sch06].

Assertion abstraction and concretization. We want to transfer a Simpl Hoare triple

$$\Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} P_{\mathrm{H}}\ \mathit{BS2H}_{\mathrm{stmt}}(s)\ Q_{\mathrm{H}}$$

to its C0 big-step semantics variant

$$\Pi, L \vDash^{\mathrm{t}}_{\mathrm{BS}} P_{\mathrm{BS}}\ s\ Q_{\mathrm{BS}}.$$

Assertions P_{H} and P_{BS} as well as Q_{H} and Q_{BS} have to be related. There are two ways to describe this relation, either by concretizing a Hoare assertion to a BS assertion, or by abstraction of a BS assertion to a Hoare one.

Every state c_{BS} for which there exists an abstract state $c_{\mathrm{H}} \in \mathit{BS2H}_{\mathrm{state}}(c_{\mathrm{BS}})$ that satisfies P_{H} satisfies the concretization of assertion P_{H}:

$$\mathit{H2BS}_{\mathrm{asrt}} :: \mathit{pow}(C_{\mathrm{H}}) \mapsto \mathit{pow}(C_{\mathrm{BS}}),$$

$$\mathit{H2BS}_{\mathrm{asrt}}(P_{\mathrm{H}}) = \{c_{\mathrm{BS}} \mid \exists\, c_{\mathrm{H}} : c_{\mathrm{H}} \in \mathit{BS2H}_{\mathrm{state}}(c_{\mathrm{BS}}) \wedge c_{\mathrm{H}} \in P_{\mathrm{H}}\}.$$

The union of the abstracted states for all BS states satisfying P_{BS} is the abstraction of assertion P_{BS}:

$$\mathit{BS2H}_{\mathrm{asrt}} :: \mathit{pow}(C_{\mathrm{BS}}) \mapsto \mathit{pow}(C_{\mathrm{H}}),$$

$$\mathit{BS2H}_{\mathrm{asrt}}(P_{\mathrm{BS}}) = \{c_{\mathrm{H}} \mid \exists\, c_{\mathrm{BS}} : c_{\mathrm{H}} \in \mathit{BS2H}_{\mathrm{state}}(c_{\mathrm{BS}}) \wedge c_{\mathrm{BS}} \in P_{\mathrm{BS}}\}.$$

In practice, it turns out that a combination of both methods works best. For the precondition we use abstraction and for the postcondition concretization. For given assertions P_{BS}, P_{H}, Q_{BS}, and Q_{H} we need to show:

- $\mathit{BS2H}_{\mathrm{asrt}}(P_{\mathrm{BS}}) \subseteq P_{\mathrm{H}}$, and
- $\mathit{H2BS}_{\mathrm{asrt}}(Q_{\mathrm{H}}) \subseteq Q_{\mathrm{BS}}$.

Unfolding the definitions yields the following proof obligations:

- $c_{\mathrm{H}} \in \mathit{BS2H}_{\mathrm{state}}(c_{\mathrm{BS}}) \wedge c_{\mathrm{BS}} \in P_{\mathrm{BS}} \longrightarrow c_{\mathrm{H}} \in P_{\mathrm{H}}$, and
- $c'_{\mathrm{H}} \in \mathit{BS2H}_{\mathrm{state}}(c'_{\mathrm{BS}}) \wedge c'_{\mathrm{H}} \in Q_{\mathrm{H}} \longrightarrow c'_{\mathrm{BS}} \in Q_{\mathrm{BS}}$.

In both cases we obtain either $c_{\mathrm{H}} \in \mathit{BS2H}_{\mathrm{state}}(c_{\mathrm{BS}})$ or $c'_{\mathrm{H}} \in \mathit{BS2H}_{\mathrm{state}}(c'_{\mathrm{BS}})$, which we can exploit in order to transfer assertions.

The following theorem which is a slightly modified version of Schirmer's Corollary 8.43 is used to transfer Hoare triples from Simpl to C0 big-step semantics.

◀ THEOREM 3.2
Transfer from Simpl to BS

Let Π be a program, let GT be a typing of global variables, let LT be a typing of local variables, and let HT be a heap typing. Let us denote a combined typing of global and local variables by $GT \circ LT$. Let Π be well-formed (1), let s be a well-typed statement (2) which is also definitely assigned (3), and let the locations we assume to be assigned be a subset of the actually assigned values in the initial state (4). Moreover, assume that the initial state is conforming (5) and there exists at least one abstracted state for the initial state (6). Having the connections between preconditions P_{BS} and P_{H} (7) and postconditions Q_{BS} and Q_{H} (8), we can conclude the valid transfer:

$$\begin{array}{lll}
(1) & & \mathit{wf\text{-}prog?}(\Pi) \\
(2) & \wedge & \forall c_{\mathrm{BS}} \in P_{\mathrm{BS}} : \Pi, GT \circ LT, HT \vdash s\surd \\
(3) & \wedge & \mathcal{D}(s, \mathit{dom}(LT), A) \\
(4) & \wedge & \forall\, c_{\mathrm{BS}} \in P_{\mathrm{BS}} : A \subseteq \mathit{dom}(c_{\mathrm{BS}}.\mathit{lvars})
\end{array}$$

$$
\begin{array}{lll}
(5) & \wedge & \forall\, c_{\mathrm{BS}} \in P_{\mathrm{BS}} : TE \vdash c_{\mathrm{BS}} :: HT, LT, GT \\
(6) & \wedge & \forall\, c_{\mathrm{BS}} \in P_{\mathrm{BS}} : BS2H_{\mathrm{state}}(c_{\mathrm{BS}}) \neq \emptyset \\
(7) & \wedge & BS2H_{\mathrm{asrt}}(P_{\mathrm{BS}}) \subseteq P_{\mathrm{H}} \\
(8) & \wedge & (\forall\, HT' : dom(HT) \subseteq dom(HT') \longrightarrow \\
 & & \quad H2BS_{\mathrm{asrt}}(Q_{\mathrm{H}}) \subseteq \\
 & & \quad\quad \{c'_{\mathrm{BS}} \mid \quad TE \vdash c'_{\mathrm{BS}} :: HT', LT, GT \\
 & & \quad\quad\quad\quad \wedge\, A \cup (dom(LT)) \cap \mathcal{A}(s) \subseteq dom(c'_{\mathrm{BS}}.lvars) \\
 & & \quad\quad\quad\quad \longrightarrow c'_{\mathrm{BS}} \in Q_{\mathrm{BS}}\}) \\
 & \longrightarrow & \Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} P_{\mathrm{H}}\ BS2H_{\mathrm{stmt}}(s)\ Q_{\mathrm{H}} \longrightarrow \Pi, L \vDash^{\mathrm{t}}_{\mathrm{BS}} P_{\mathrm{BS}}\ s\ Q_{\mathrm{BS}}.
\end{array}
$$

3.5 Property Transfer from BS to SS

Extended C0 small-step semantics. In Section 2.3 we have defined the transition function δ_{SS} which describes the C0 source-level semantics of the compiler correctness theorem [Lei07]. However, it does not properly treat XCalls. For that we introduce an *extended* transition function δ_{SSX} which defines the *extended* C0 small-step semantics. Transition $\delta_{\mathrm{SSX}}(te, ft, c_{\mathrm{SS}}, x, xsem)$ extends δ_{SS} to handle XCalls on the extended state x, according to the extended semantics environment *xsem*. We execute n steps by δ^n_{SSX}.

The extended semantics environment for the small-step semantics is defined the same way as for the big-step semantics (Section 3.3). The only difference is value representation: in the definition of the type *xsem-t* the value type *val-t* is replaced by the C0 smal-step content type $\mathbb{N} \mapsto$ *mcell-t*. We do not introduce special type of the extended state for the small-step semantics, but rather overload the type *xsem-t*.

Valid SS configurations for property transfer. Additionally to the standard constraints on C0 small-step semantics configurations $C0\surd$ we impose further restrictions for the purpose of property transfer. These define the set of SS configurations $valid_{\mathrm{SS}}(te, ft)$.

- First, certain statements and expressions are not allowed since they are not supported by the big-step semantics or the Hoare logics, namely address-of operation, external calls, and inline assembly statements, denoted by *noAddrOf-Asm-ESCall?*(s) for a statement s. With the help of that we claim that forbidden statements and expressions do not occur in the program rest as well as all function calls reachable from the statements in it:

  $$
  \begin{array}{lll}
  & & noAddrOf\text{-}Asm\text{-}ESCall?(c_{\mathrm{SS}}.prog) \\
  & \wedge & \forall\, p \in SCalls(ft, scalls(c_{\mathrm{SS}}.prog)) : \\
  & & \quad noAddrOf\text{-}Asm\text{-}ESCall?((\!|map\text{-}of(ft, p)|\!|).body).
  \end{array}
  $$

  The function *scalls* collects the procedure calls of a statement and the set *SCalls* inductively collects further calls in the procedure environment for the given initial set.

- Global variables have to be initialized, denoted by *glob-init?*$(c_{\mathrm{SS}}.mem)$:

  $$\forall vn : vn \in map(fst, gst(c_{\mathrm{SS}}.mem)) \longrightarrow vn \in c_{\mathrm{SS}}.mem.gm.init.$$

- We allow only pointers to (root) heap locations in the memory, denoted by *only-heap-pointer?*($c_{\mathrm{SS}}.mem$). This predicate is a strengthened version of Leinenbach's *type correct memory* [Lei07, Definition 5.23] and *type correct heap* [Lei07, Definition 5.24].

- Moreover, every type in the heap has to have a proper type name in the type environment, denoted by *named-ty?*($te, hst(c_{\mathrm{SS}}.mem)$).

- Only root positions of local and global variables are valid return destinations: $\forall\, gv \in map(snd, c_{\mathrm{SS}}.mem.lm) : root?(gv)$.

- In case the return destination is a local variable it has to be defined in the frame stack below: *wf-retvars?*($snd(hd(c_{\mathrm{SS}}.mem.lm)), tl(c_{\mathrm{SS}}.mem.lm)$).

- Each procedure in the procedure environment has to pass the definite assignment check, where we assume parameters and local variables to be initialized:

$$\forall p \in \{map(snd, ft)\}, pns = \{map(fst, p.params)\}, lns = \{map(fst, p.lvars)\} : \mathcal{D}(p.body, pns \cup lns, pns)$$

- Similarly, the program rest has to pass the definite assignment check. However, as the program rest gets expanded during the small-step computation it may contain multiple returns and hence the program rest is split into several regions that correspond to procedure invocations on the frame stack. This generalization is formalized by $\mathcal{D}s$ and LAs:

$$\mathcal{D}s(c_{\mathrm{SS}}.prog, (map(fst, lst_{\mathrm{top}}(c_{\mathrm{SS}}.mem)), lm_{\mathrm{top}}(c_{\mathrm{SS}}.mem).init) \circ LAs(res_{\mathrm{top}}(c_{\mathrm{SS}}.mem), tl(c_{\mathrm{SS}}.mem.lm))).$$

Relational view of SS transitions. A final configuration is reached when the program rest is *Skip*. In this case the transition function is the identity. We define a relational view of the transition function that really stops in this configuration by a single rule:

$$\frac{c_{\mathrm{SS}}.prog \neq Skip}{te, ft, xsem \vdash_{\mathrm{SS}} \lfloor (c_{\mathrm{SS}}, x) \rfloor \rightarrow \delta_{\mathrm{SSX}}(te, ft, c_{\mathrm{SS}}, x, xsem)}.$$

We refer to the reflexive transitive closure by substituting the arrow $\rightarrow$ by $\rightarrow^*$.

SS Hoare triple. Our notion of a Hoare triple at the level of the small-step semantics is biased towards property transfer from the big-step level. We encode validity of configurations and transition invariants right into this notation as well as the set of local variables L which comes from the BS level. The reason for this peculiarity is that the set of local variables is directly encoded into each configuration in the small-step semantics. Hence we cannot relate those entities between the big- and the small-step semantics without referring to a small-step configuration which is only accessible within the pre- and postconditions of the Hoare triple.

$$
\begin{aligned}
&\Pi, xsem, L \vDash^{\mathrm{t}}_{\mathrm{SS}} P\ s\ Q\ = \\
&\quad \forall\, c_{\mathrm{SS}}, x : (c_{\mathrm{SS}}.mem, x) \in P \\
&\qquad \wedge\ c_{\mathrm{SS}} \in valid_{\mathrm{SS}}(\Pi.te, \Pi.ft) \\
&\qquad \wedge\ \#ret(c_{\mathrm{SS}}.prog) = 0 \\
&\qquad \wedge\ L = \{map(fst, lst_{\mathrm{top}}(c_{\mathrm{SS}}.mem))\} \\
&\qquad \wedge\ \neg(\Pi.te, \Pi.ft, xsem \vdash_{\mathrm{SS}} \lfloor (c_{\mathrm{SS}}, x)\rfloor \rightarrow \ldots(\infty)) \\
&\qquad \wedge\ \forall\, cx' :\ \Pi.te, \Pi.ft, xsem \vdash_{\mathrm{SS}} \lfloor (c_{\mathrm{SS}}, x)\rfloor \rightarrow^{*} cx'\ \wedge\ final?(cx') \\
&\qquad\quad \longrightarrow\ \exists\, c'_{\mathrm{SS}}, x', hst : \\
&\qquad\qquad cx' = \lfloor (c'_{\mathrm{SS}}, x')\rfloor \\
&\qquad\qquad \wedge\ (c'_{\mathrm{SS}}.mem, x') \in Q \\
&\qquad\qquad \wedge\ c'_{\mathrm{SS}} \in valid_{\mathrm{SS}}(\Pi.te, \Pi.ft) \\
&\qquad\qquad \wedge\ trans\text{-}inv?(c_{\mathrm{SS}}.prog, hst, c_{\mathrm{SS}}, c'_{\mathrm{SS}})
\end{aligned}
$$

We consider initial configuration (c_{SS}, x). The memory of the configuration and the extended state fulfill the precondition P. Moreover the initial configuration is valid, the program rest does not contain a return statement. Since we define total correctness, non-terminating computations are ruled out by the judgment $\Pi.te, \Pi.ft, xsem \vdash_{\mathrm{SS}} \lfloor (c_{\mathrm{SS}}, x)\rfloor \rightarrow \ldots(\infty)$ which is defined as

$$\exists f : f(0) = \lfloor (c_{\mathrm{SS}}, x)\rfloor \wedge \forall i : \Pi.te, \Pi.ft, xsem \vdash_{\mathrm{SS}} f(i) \rightarrow f(i+1).$$

For every computation the final configuration cx', denoted by $final?(cx')$, must not be the error configuration, the postcondition Q has to hold, and it has to be valid. A non-error final configuration has an empty program rest. Moreover, the transition invariant between initial and final configurations has to be satisfied.

DEFINITION 3.3 ▶ Transition invariant

$$
\begin{aligned}
&trans\text{-}inv?(s, hst, c_{\mathrm{SS}}, c'_{\mathrm{SS}})\ = \\
&\qquad tl(c'_{\mathrm{SS}}.mem.lm) = tl(c_{\mathrm{SS}}.mem.lm) \\
&\qquad \wedge\ lst_{\mathrm{top}}(c'_{\mathrm{SS}}.mem) = lst_{\mathrm{top}}(c_{\mathrm{SS}}.mem) \\
&\qquad \wedge\ gst(c'_{\mathrm{SS}}.mem) = gst(c_{\mathrm{SS}}.mem) \\
&\qquad \wedge\ hst(c'_{\mathrm{SS}}.mem) = hst(c_{\mathrm{SS}}.mem) \circ hst \\
&\qquad \wedge\ res_{\mathrm{top}}(c'_{\mathrm{SS}}.mem) = res_{\mathrm{top}}(c_{\mathrm{SS}}.mem) \\
&\qquad \wedge\ fst(hd(c_{\mathrm{SS}}.mem.lm)) \cup map(fst, lst_{\mathrm{top}}(c_{\mathrm{SS}}.mem)) \cap \mathcal{A}(s) \\
&\qquad\qquad \subseteq fst(hd(c'_{\mathrm{SS}}.mem.lm)).init
\end{aligned}
$$

The transition invariant captures essential invariants of the small-step computation that hold between the initial and final configurations of the procedure call that we transfer. First of all the computation only affects the topmost frame of local variables and the type information of this frame is not changed, as for the global variables. The type information for the heap memory may only grow. Return destinations of the top local memory frames are equal. Finally the increasing set of initialized local variables as approximated by the definite assignment analysis $\mathcal{A}$.

THEOREM 3.4 ▶ Transfer from BS to SS

Assume that SS configuration c_{SS} satisfies validity requirements for property transfer (1) and its program rest consists of statement s (2). Assume that local variables names context L consists of variables of the topmost local memory frame(3). Moreover, a precondition on the SS level is satisfied (4). If under these assumptions there exist pre- and postconditions on the BS level, such that BS Hoare triple holds (5), BS precondition is satisfied by the abstracted configuration (6), and for all final valid SS configurations (7) respecting the

transition invariant (8) and the BS postcondition (9) the SS postcondition holds (10), then SS Hoare triple holds:

$$
\begin{array}{lll}
(1) & & c_{\mathrm{SS}} \in \mathit{valid}_{\mathrm{SS}}(\Pi.\mathit{te}, \Pi.\mathit{ft}) \\
(2) & \wedge & c_{\mathrm{SS}}.\mathit{prog} = s \wedge \#\mathit{ret}(s) = 0 \\
(3) & \wedge & L = \mathit{map}(\mathit{fst}, \mathit{lst}_{\mathrm{top}}(c_{\mathrm{SS}}.\mathit{mem})) \\
(4) & \wedge & (c_{\mathrm{SS}}, x) \in P_{\mathrm{SS}} \\
(5) & \longrightarrow & (\exists\, P_{\mathrm{BS}}, Q_{\mathrm{BS}} : \Pi, \mathit{xsem}, L \vDash^{\mathrm{t}}_{\mathrm{BS}} P_{\mathrm{BS}}\ s\ Q_{\mathrm{BS}} \\
(6) & & \quad \wedge\ \mathit{SS2BS}_{\mathrm{state}}(c_{\mathrm{SS}}, x) \in P_{\mathrm{BS}} \\
(7) & & \quad \wedge\ (\forall\, c'_{\mathrm{SS}}, x', \mathit{hst}' : c'_{\mathrm{SS}} \in \mathit{valid}_{\mathrm{SS}}(\Pi.\mathit{te}, \Pi.\mathit{ft}) \\
(8) & & \qquad \wedge\ \mathit{trans\text{-}inv?}(s, \mathit{hst}', c_{\mathrm{SS}}, c'_{\mathrm{SS}}) \\
(9) & & \qquad \wedge\ \mathit{SS2BS}_{\mathrm{state}}(c'_{\mathrm{SS}}, x') \in Q_{\mathrm{BS}} \\
(10) & & \qquad \longrightarrow\ (c'_{\mathrm{SS}}, x') \in Q_{\mathrm{SS}})) \\
 & \longrightarrow & \Pi, \mathit{xsem}, L \vDash^{\mathrm{t}}_{\mathrm{SS}} P_{\mathrm{SS}}\ s\ Q_{\mathrm{SS}}
\end{array}
$$

The core premise of this transfer theorem strengthens the precondition and weakens the postcondition. Additionally, we take the different layers and the system invariants of the small-step layer into account. For an arbitrary initial configuration that fulfills the constraints of small-step validity and the precondition P_{SS} we have to supply a big-step Hoare triple $\Pi, \mathit{xsem}, L \vDash^{\mathrm{t}}_{\mathrm{BS}} P_{\mathrm{BS}}\ s\ Q_{\mathrm{BS}}$ such that the big-step abstraction of the configuration fulfills the precondition P_{BS}. For a final valid small-step configuration, that fulfills the transition invariant and for which the big-step abstraction fulfills the postcondition Q_{BS} we have to derive the small-step postcondition Q_{SS}. Note the existential quantification on the big-step pre- and postcondition. It is under the universal quantification of the initial configuration and hence can depend on this configuration. The function $\mathit{SS2BS}_{\mathrm{state}}$ is used to abstract a small-step configuration to a big-step state.

From Hoare triples to operational-semantics-style computations. At this point we are able to transfer a Hoare triple for a single procedure call to the small-step level. One basic motivation of reasoning at the low abstraction level of the small-step semantics instead of the convenient Hoare logic level is the ability to combine ordinary C0-computation with inline assembly code. Hence, it would be worthless if we were incapable of using the transferred result in a situation where inline assembly code is part of the program rest. The following theorem takes care about that.

◀ THEOREM 3.5
From Hoare triples to computations

$$
\begin{array}{lll}
(1) & & \Pi, \mathit{xsem}, L \vDash^{\mathrm{t}}_{\mathrm{SS}} P\ s\ Q \\
(2) & \wedge & s = c_{\mathrm{SS}}.\mathit{prog} \wedge \#\mathit{ret}(c_{\mathrm{SS}}.\mathit{prog}) = 0 \\
 & \wedge & (c_{\mathrm{SS}}.\mathit{mem}, x) \in P \\
 & \wedge & c_{\mathrm{SS}} \in \mathit{valid}'_{\mathrm{SS}}(\Pi.\mathit{te}, \Pi.\mathit{ft}) \\
 & \wedge & L = \mathit{map}(\mathit{fst}, \mathit{lst}_{\mathrm{top}}(c_{\mathrm{SS}}.\mathit{mem})) \\
(3) & \longrightarrow & \exists\, n, c'_{\mathrm{SS}}, x', \mathit{hst}' : \delta^{n}_{\mathrm{SSX}}(\Pi.\mathit{te}, \Pi.\mathit{ft}, c_{\mathrm{SS}}, x, \mathit{xsem}) = \lfloor (c'_{\mathrm{SS}}, x') \rfloor \\
(4) & & \qquad \wedge\ c'_{\mathrm{SS}}.\mathit{prog} = \mathit{Skip} \\
 & & \qquad \wedge\ c'_{\mathrm{SS}} \in \mathit{valid}'_{\mathrm{SS}}(\Pi.\mathit{te}, \Pi.\mathit{ft}) \\
 & & \qquad \wedge\ \mathit{trans\text{-}inv?}(s, \mathit{hst}', c_{\mathrm{SS}}, c'_{\mathrm{SS}}) \\
 & & \qquad \wedge\ (c'_{\mathrm{SS}}.\mathit{mem}, x') \in Q
\end{array}
$$

We start in a configuration where the program rest agrees with the state-

ment in the Hoare triple (1,2). As we have total correctness we know that the computation leads to a state where the statement is completely executed, i.e., has evaluated to *Skip* (3,4). In the remainder of the theorem the usual properties about the initial and final configuration show up.

CHAPTER 4

Specification of Demand Paging

The chapter introduces an abstract page-fault handler configuration as well as a configuration for extended state. These configurations are used to specify the intended behavior of the demand paging software. In the subsequent chapters we will map them to the demand paging implementation represented on the levels of Simpl, big-step, and small-step semantics. We introduce the initial abstract configuration which defines the state of the demand paging data structures after execution of the initialization code. An address translation algorithm, operations for transferring pages between the physical memory and the hard disk, and a page-fault handling algorithm are defined over these configurations. All correctness properties essential for the top-level correctness proof are collected into a validity predicate of the abstract page-fault handler. This predicate is the main invariant maintained by the demand paging implementation. It is established over the initial configuration and proven to be preserved under the page-fault handling algorithm.

4.1 Configurations

The general approach to verification of low-level implementations is abstraction, i.e., mapping the data structures and algorithms constituting the implementation to higher-level concepts. The implementation of demand paging operates on two kinds of variables: (i) C0 data structures and variables, and (ii) user and swap memories which could be accessed only by assembly code. For the first group we introduce an abstract page-fault handler configuration, or abstract PFH configuration, — a record which collects abstract HOL representations of the handler's C0 data structures. Memories from the second group are modeled by the so called extended state of the page-fault handler, or simply the extended state.

Before defining abstract PFH configurations let us formalize an auxiliary concept — a page descriptor.

Page descriptors. The record type *pd-t* holding information about one user page defines the type for page descriptors. One such page descriptor *pd* corresponds to exactly one physical page of user memory and has three components:

- $pd.pid :: \mathbb{N}$, the process identifier which denotes to which virtual machine an associated physical page belongs,
- $pd.vpx :: \mathbb{N}$, the virtual page index showing to which virtual page the corresponding physical page belongs, and
- $pd.ppx :: \mathbb{N}$, the physical page index which points to the user page in the physical memory.

Components of a page descriptor have to be appropriately bounded: the process identifier field *pd.pid* is bounded by the maximum number of processes, and the virtual *pd.vpx* and physical *pd.ppx* page index fields have to be page addresses, i.e., bounded by `TOT_PGS`.

DEFINITION 4.1 ► Sub-typing of page descriptors

$$\begin{aligned} subtyping_{\mathrm{pd}}?(pd) \quad &= \quad pd.pid < \texttt{MAX\_PID} \\ &\wedge \quad pd.vpx < \texttt{TOT\_PGS} \\ &\wedge \quad pd.ppx < \texttt{TOT\_PGS} \end{aligned}$$

4.1.1 Abstract Configuration

Abstract versions of data structures from the page-fault handler implementation together with those fields of process control blocks that are relevant for page-fault handling are collected in the record C_{PFH} . An abstract configuration c_{PFH} of the page-fault handler is an element of this type. It has the following fields:

- $c_{\mathrm{PFH}}.active :: pd\text{-}t^*$, the active list of page descriptors associated with user memory pages that store a virtual page,
- $c_{\mathrm{PFH}}.free :: pd\text{-}t^*$, the free list of descriptors corresponding to unused physical pages,
- $c_{\mathrm{PFH}}.bpfree :: \mathbb{N}^*$, the stack of free big-page indices,

- $c_{\text{PFH}}.pt :: \mathbb{N}^{**}$, the page table space,
- $c_{\text{PFH}}.bpt :: \mathbb{N}^{*}$, the big-page table space,
- $c_{\text{PFH}}.pto :: \mathbb{Z}^{*}$, the list of page table origins of processes,
- $c_{\text{PFH}}.ptl :: \mathbb{Z}^{*}$, the list of page table lengths of processes,
- $c_{\text{PFH}}.bpto :: \mathbb{Z}^{*}$, the list of big-page table origins of processes,
- $c_{\text{PFH}}.bptl :: \mathbb{Z}^{*}$, the list of big-page table lengths of processes.

The components of the abstract configuration are modeled by the unbounded data types. However, for verification we need to impose size constraints on them reflecting the size of corresponding data structures from the implementation.

Since active and free lists describe altogether all user memory pages their total length has to be equal to the total number of user memory pages. Elements of these lists are page descriptors. They have to be bounded by means of the sub-typing predicate for page descriptors (Definition 4.1).

$$\begin{aligned} \mathit{subtyping}_{\text{active-free}}?(c_{\text{PFH}}) &= |c_{\text{PFH}}.\mathit{active}| + |c_{\text{PFH}}.\mathit{free}| = \texttt{USER\_PGS} \\ &\wedge (\forall\, i < |c_{\text{PFH}}.\mathit{active}| : \mathit{subtyping}_{\text{pd}}?(c_{\text{PFH}}.\mathit{active}[i])) \\ &\wedge (\forall\, i < |c_{\text{PFH}}.\mathit{free}| : \mathit{subtyping}_{\text{pd}}?(c_{\text{PFH}}.\mathit{free}[i])) \end{aligned}$$

◀ DEFINITION 4.2
Sub-typing of active and free lists

If no processes store (parts of) their virtual memory on the hard disk all big pages are free. Therefore, the maximum length of the stack of free big pages is the total number of big pages. Each entry of the stack points to some big page. Hence, the values of the entries are also bounded by the total number of big pages.

$$\begin{aligned} \mathit{subtyping}_{\text{bpfree}}?(c_{\text{PFH}}) &= |c_{\text{PFH}}.\mathit{bpfree}| \leq \texttt{TOT\_BIG\_PGS} \\ &\wedge \forall\, i < |c_{\text{PFH}}.\mathit{bpfree}| : c_{\text{PFH}}.\mathit{bpfree}[i] < \texttt{TOT\_BIG\_PGS} \end{aligned}$$

◀ DEFINITION 4.3
Sub-typing of free big-pages stack

Our abstraction of the page table space is quite close to the corresponding array in the implementation: the size of dimensions of $c_{\text{PFH}}.pt$ is the same as in the implementation. The first dimension is bounded by TOT_PGS_PT whereas the second by PTES_PER_PG. Each entry of the page table space is four bytes long (TOT_PGS · PG_SZ).

$$\begin{aligned} \mathit{subtyping}_{\text{pt}}?(c_{\text{PFH}}) &= |c_{\text{PFH}}.pt| = \texttt{TOT\_PGS\_PT} \\ &\wedge (\forall\, i < \texttt{TOT\_PGS\_PT} : |c_{\text{PFH}}.pt[i]| = \texttt{PTES\_PER\_PG}) \\ &\wedge (\forall\, i < \texttt{TOT\_PGS\_PT} : \forall j < \texttt{PTES\_PER\_PG} : \\ &\qquad c_{\text{PFH}}.pt[i][j] < \texttt{TOT\_PGS} \cdot \texttt{PG\_SZ}) \end{aligned}$$

◀ DEFINITION 4.4
Sub-typing of page table space

Each big page has its own entry in the big-page table. This entry is an index of some big page. Therefore, it is bounded by the total number of big pages.

$$\begin{aligned} \mathit{subtyping}_{\text{bpt}}?(c_{\text{PFH}}) &= |c_{\text{PFH}}.\mathit{bpt}| = \texttt{TOT\_BIG\_PGS} \\ &\wedge \forall\, i < \texttt{TOT\_BIG\_PGS} : c_{\text{PFH}}.\mathit{bpt}[i] < \texttt{TOT\_BIG\_PGS} \end{aligned}$$

◀ DEFINITION 4.5
Sub-typing of big-page table space

As for the process control block components of the abstract configuration, in each component there are as many entries as processes in the system. The page table origin is bounded by the total number of virtual pages whereas the

big-page table origin by the total number of big pages. The same upper bounds are respected by page and big-page table lengths. However, the domain of their values is additionally extended with -1 to denote that a process is inactive and has no virtual memory.

DEFINITION 4.6 ► Sub-typing of PCBs

$$\begin{aligned} \mathit{subtyping}_{\mathrm{PCB}}?(c_{\mathrm{PFH}}) \quad = \quad & |c_{\mathrm{PFH}}.\mathit{pto}| = \texttt{MAX\_PID} \\ \wedge \quad & |c_{\mathrm{PFH}}.\mathit{ptl}| = \texttt{MAX\_PID} \\ \wedge \quad & |c_{\mathrm{PFH}}.\mathit{bpto}| = \texttt{MAX\_PID} \\ \wedge \quad & |c_{\mathrm{PFH}}.\mathit{bptl}| = \texttt{MAX\_PID} \\ \wedge \quad & (\forall\, i < \texttt{MAX\_PID} : 0 \leq c_{\mathrm{PFH}}.\mathit{pto}[i] < \texttt{TOT\_PGS} \\ & \qquad\qquad \wedge\, 0 \leq c_{\mathrm{PFH}}.\mathit{bpto}[i] < \texttt{TOT\_BIG\_PGS} \\ & \qquad\qquad \wedge\, -1 \leq c_{\mathrm{PFH}}.\mathit{ptl}[i] < \texttt{TOT\_PGS} \\ & \qquad\qquad \wedge\, -1 \leq c_{\mathrm{PFH}}.\mathit{bptl}[i] < \texttt{TOT\_BIG\_PGS}) \end{aligned}$$

DEFINITION 4.7 ► Sub-typing of abstract PFH configuration

Altogether, sub-typing of an abstract PFH configuration $\mathit{subtyping}_{\mathrm{PFH}}?(c_{\mathrm{PFH}})$ is the conjunction of Definitions 4.2–4.6.

4.1.2 Extended State

The extended state of the demand paging specification is an abstraction of the non-system part of the physical memory of the machine running the handler and the hard disk of this machine. The state is used to specify the effects of read and write operations to/from the hard disk invoked by the handler. The state is modeled by the record C_{X} whose instances c_{X} have two components:

- $c_{\mathrm{X}}.\mathit{mem} :: \mathbb{N}^{**}$, the user part of the physical memory not reachable by C0, and
- $c_{\mathrm{X}}.\mathit{swap} :: \mathbb{N}^{**}$, the content of the hard disk excluding the boot region.

Figure 4.1 depicts the parts of the physical memory and the hard disk content which are modeled by the extended state.

The component $c_{\mathrm{X}}.\mathit{mem}$ is an abstraction of the user memory region — `USER_PGS` pages of memory starting from address `KERNEL_PGS` — and the zero filled page. The component $c_{\mathrm{X}}.\mathit{swap}$ is an abstraction of the swap memory stored at the hard disk — the region of `TOT_BIG_PGS` big pages starting from address `BOOT_PGS`. Each of the extended state components is modeled as a list of pages, e.g., in the formula $c_{\mathrm{X}}.\mathit{mem}[i][j]$ the index i denotes the page address whereas j is an offset within a page.

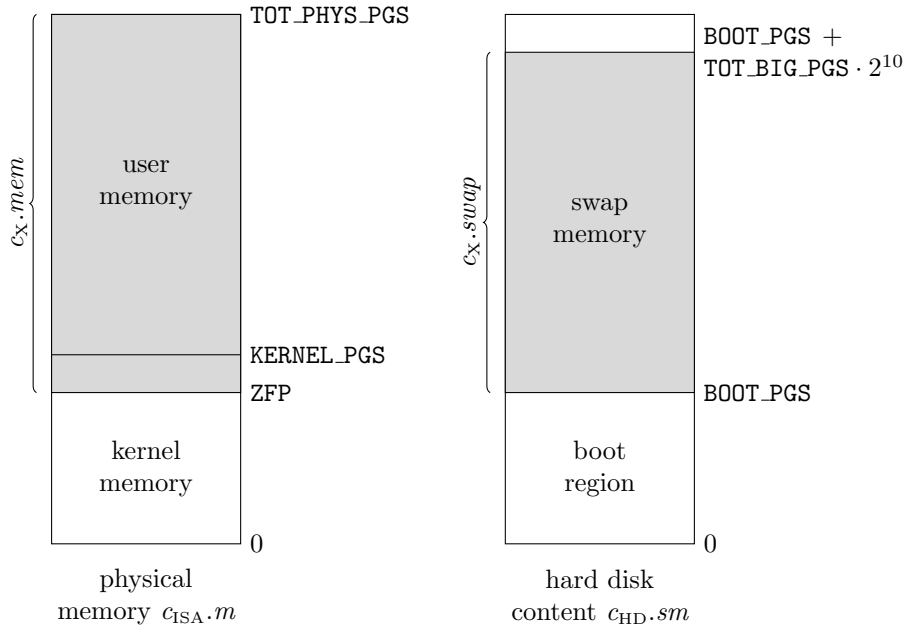
DEFINITION 4.8 ► Sub-typing of extended state

$$\begin{aligned} \mathit{subtyping}_{\mathrm{X}}?(c_{\mathrm{X}}) \quad = \quad & |c_{\mathrm{X}}.\mathit{mem}| = \texttt{USER\_PGS} + 1 \\ \wedge \quad & |c_{\mathrm{X}}.\mathit{swap}| = \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG} \\ \wedge \quad & (\forall\, i < |c_{\mathrm{X}}.\mathit{mem}| : |c_{\mathrm{X}}.\mathit{mem}[i]| = \texttt{PG\_SZ\_WD}) \\ \wedge \quad & (\forall\, i < |c_{\mathrm{X}}.\mathit{swap}| : |c_{\mathrm{X}}.\mathit{swap}[i]| = \texttt{PG\_SZ\_WD}) \end{aligned}$$

4.2 Initial Configuration

An abstract PFH configuration is set to its initial state by executing the initialization code of demand paging.

Figure 4.1: Modeling user and swap memories with extended state



Initial page descriptor. An initial configuration of a page descriptor is constructed by means of the function $\mathit{init\text{-}pd}(i) = \mathit{pd}_0$, such that $\mathit{pd}_0.\mathit{pid} = 0$, $\mathit{pd}_0.\mathit{vpx} = 0$, and $\mathit{pd}_0.\mathit{ppx} = i$.

Initial abstract PFH configuration. The initial configuration $\mathit{init\text{-}c}_{\mathrm{PFH}}$ of the abstract page-fault handler is defined as follows:

- $\mathit{init\text{-}c}_{\mathrm{PFH}}.\mathit{active} = [\,]$, the active list is empty,
- $\mathit{init\text{-}c}_{\mathrm{PFH}}.\mathit{free} = \mathit{map}(\mathit{init\text{-}pd}, [\mathtt{TOT\_PHYS\_PGS} - 1..\mathtt{KERNEL\_PGS}])$, the free list describes all user physical pages,
- $\mathit{init\text{-}c}_{\mathrm{PFH}}.\mathit{bpfree} = [\mathtt{TOT\_BIG\_PGS} - 1..0]$, the stack of free big-pages contains entries for all big pages,
- $\mathit{init\text{-}c}_{\mathrm{PFH}}.\mathit{pt} = \mathit{rep}(\mathit{rep}(0, \mathtt{PTES\_PER\_PG}), \mathtt{TOT\_PGS\_PT})$, all entries of the page table array are initialized with zeros,
- $\mathit{init\text{-}c}_{\mathrm{PFH}}.\mathit{bpt} = \mathit{rep}(0, \mathtt{TOT\_BIG\_PGS})$, so do the entries of the big-page table list,
- $\mathit{init\text{-}c}_{\mathrm{PFH}}.\mathit{pto} = 0 \circ \mathit{map}(\mathit{init\text{-}pto}, [1..\mathtt{MAX\_PID} - 1])$, the page table origins of users are distributed inside the page table array with equal gaps between them computed by the function

$$\mathit{init\text{-}pto}(i) = \frac{\mathtt{PT\_START} + \dfrac{(i-1) \cdot \mathtt{TOT\_PGS\_PT} \cdot \mathtt{PTES\_PER\_PG} \cdot 4}{\mathtt{MAX\_PID}}}{\mathtt{PG\_SZ}};$$

  note that we take into consideration the start offset PT_START of page tables in the physical memory,

- $init\text{-}c_{\mathrm{PFH}}.ptl = 0 \circ rep(-1, \texttt{MAX\_PID} - 1,)$, the page table lengths of processes are set to -1 which denotes that all processes have no allocated memory,
- $init\text{-}c_{\mathrm{PFH}}.bpto = 0 \circ rep(0, \texttt{MAX\_PID} - 1)$, the big-page table origins are initialized with zeros, and
- $init\text{-}c_{\mathrm{PFH}}.bptl = 0 \circ rep(-1, \texttt{MAX\_PID} - 1)$, the big-page table lengths are set to -1.

4.3 Address Translation

A memory access of a user process *pid* at virtual address *va* is subject to address translation. This operation either generates a page fault signal or delivers a physical memory address. In the following we define an address translation mechanism in terms of abstract PFH configurations.

4.3.1 Physical Memory Address

For a process identifier *pid* the component of an abstract PFH configuration $c_{\mathrm{PFH}}.pto[pid]$ stores the origin of the page table of the process *pid* in the physical memory of the underlying machine. The origin is measured in pages. On the side of the abstract configuration the process's page table origin is an index of the first dimension of the page table array $c_{\mathrm{PFH}}.pt$. We compute this index by means of the function

DEFINITION 4.9 ► Page table origin in page table space

$$pto_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid) \quad = \quad \frac{c_{\mathrm{PFH}}.pto[pid] \cdot \texttt{PG\_SZ} - \texttt{PT\_START}}{4 \cdot \texttt{PTES\_PER\_PG}}.$$

It first compensates for the start address PT_START of the page table memory region and then considers page-alignment of page tables: each page contains PTES_PER_PG page table entries, each four bytes long.

The abstract configuration's component $c_{\mathrm{PFH}}.ptl[pid]$ stores the page table length of the process *pid*. That means that the process *pid* has $c_{\mathrm{PFH}}.ptl[pid]$ entries in the page table. Since page tables are page aligned, we define a function which computes how many pages a process's page table occupies:

DEFINITION 4.10 ► Page table length in page table space

$$ptl_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid) \quad = \quad \frac{c_{\mathrm{PFH}}.ptl[pid] + 1}{\texttt{PTES\_PER\_PG}}.$$

Now we introduce page table entry addresses. Let *vpx* be a virtual page index. As the page table array has two dimensions we need two addresses: one for each dimension. The page table entry address for the first dimension is computed by means of the function

DEFINITION 4.11 ► Page table entry address: first dimension

$$ptea_1(c_{\mathrm{PFH}}, pid, vpx) \quad = \quad pto_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid) + vpx/\texttt{PTES\_PER\_PG}.$$

It counts *vpx* entries in the page table of the process *pid* starting at its origin $pto_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid)$ and aligns the result pagewise.

The page table entry address for the second dimension is defined as

DEFINITION 4.12 ► Page table entry address: second dimension

$$ptea_2(vpx) \quad = \quad vpx \bmod \texttt{PTES\_PER\_PG}.$$

In this case the address computation function simply ignores the part of *vpx* which is greater than the size of the second dimension of the page table array.

Having page table entry addresses we can define page table entries in terms of the abstract PFH configurations. They are computed by means of the function

$$pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, vpx) \quad = \quad c_{\mathrm{PFH}}.pt[ptea_1(c_{\mathrm{PFH}}, pid, vpx)][ptea_2(vpx)].$$

◀ DEFINITION 4.13
Page table entry

Finally, the physical memory address for a virtual address va is obtained by combining the physical page index with the byte index:

$$pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, va) \quad = \quad px(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, px(va))) \cdot \mathtt{PG\_SZ} + bx(va).$$

◀ DEFINITION 4.14
Physical memory address

4.3.2 Page Faults

Let pte be a four byte long page table entry. Table 4.1 defines how we compute valid, protected, and executable bits from the entry.

Table 4.1: Bits computed from a page table entry

Bit	Notation	Computation
valid	$valid?(pte)$	$pte/2^{\mathtt{VALID\_POS}} \bmod 2$
protected	$prot?(pte)$	$pte/2^{\mathtt{PROT\_POS}} \bmod 2$
executable	$exec?(pte)$	$pte/2^{\mathtt{EXEC\_POS}} \bmod 2$

If a memory access takes place at virtual address va which is greater or equal to the size of virtual memory of a process pid a page table length exception takes place:

$$ptlexcp_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, pid, va) \quad = \quad px(va) \geq c_{\mathrm{PFH}}.ptl[pid] + 1.$$

◀ DEFINITION 4.15
PTL exception

An invalid access page fault occurs if a process pid accesses memory at virtual address va such that the valid bit of the corresponding page table entry is not set:

$$iapf?(c_{\mathrm{PFH}}, pid, va) \quad = \quad \neg valid?(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, px(va))).$$

◀ DEFINITION 4.16
Invalid access page fault

A zero protection page fault occurs on writing at virtual address va, such that the corresponding physical page index points to the zero filled page. That the memory access is a write access is defined by the external parameter $intent$ which is passed to the page-fault handler. Any value of it which is different from `READ` denotes a write access of different kinds:

$$\begin{aligned} zppf?(c_{\mathrm{PFH}}, pid, va, intent) \quad = \quad & px(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, px(va))) = \mathtt{ZFP} \\ \wedge \quad & prot?(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, px(va))) \\ \wedge \quad & intent \neq \mathtt{READ}. \end{aligned}$$

◀ DEFINITION 4.17
Zero protection page fault

Altogether, page faults distinguishable on the software level:

$$\begin{aligned} & pf_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, pid, va, intent) = \\ & \qquad iapf?(c_{\mathrm{PFH}}, pid, va) \vee zppf?(c_{\mathrm{PFH}}, pid, va, intent). \end{aligned}$$

◀ DEFINITION 4.18
Page fault

4.4 I/O Operations on Swap Memory

Let *vpx* be a virtual page index, a natural number less than TOT_PGS. A big-page index of *vpx* is computed by means of the function

DEFINITION 4.19 ▶ Big-page index

$$\mathit{bpx\text{-}of\text{-}vpx}(\mathit{vpx}) \quad = \quad \mathit{vpx}/\texttt{PGS\_PER\_BIG\_PG}.$$

With constant values considered in this thesis a big page index has a meaning of ten leading bits of the virtual page index: $\mathit{bpx\text{-}of\text{-}vpx}(\mathit{vpx}) = \mathit{bin}(\mathit{vpx})[19:10]$. A big byte index of *vpx* is computed by meas of the function

DEFINITION 4.20 ▶ Big byte index

$$\mathit{bbx\text{-}of\text{-}vpx}(\mathit{vpx}) \quad = \quad \mathit{vpx} \bmod \texttt{PGS\_PER\_BIG\_PG}.$$

It has a meaning of ten trailing bits of the virtual page index: $\mathit{bbx\text{-}of\text{-}vpx}(\mathit{vpx}) = \mathit{bin}(\mathit{vpx})[9:0]$.

For a process identifier *pid* and a virtual page index *vpx* a *big-page table entry address* is computed by summing the big-page table origin of the process *pid* with the big-page index of *vpx*:

DEFINITION 4.21 ▶ Big-page table entry address

$$\mathit{bptea}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{vpx}, \mathit{pid}) \quad = \quad c_{\mathrm{PFH}}.\mathit{bpto}[\mathit{pid}] + \mathit{bpx\text{-}of\text{-}vpx}(\mathit{vpx}).$$

The corresponding *big-page table entry* is defined as an element of the big-page table list taken at the big-page table entry address:

DEFINITION 4.22 ▶ Big-page table entry

$$\mathit{bpte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{vpx}, \mathit{pid}) \quad = \quad c_{\mathrm{PFH}}.\mathit{bpt}[\mathit{bptea}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{vpx}, \mathit{pid})].$$

We obtain a *swap page index* by adding an offset of big byte index to the big-page table entry. Additionally, we have to consider an offset of BOOT_PGS pages which occupy the beginning of the hard disk with an operating system image:

DEFINITION 4.23 ▶ Swap page index

$$\mathit{spx}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{vpx}, \mathit{pid}) = \mathit{bpte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{vpx}, \mathit{pid}) \cdot \texttt{BIG\_PG\_SZ} + \mathit{bbx\text{-}of\text{-}vpx}(\mathit{vpx}) + \texttt{BOOT\_PGS}.$$

4.4.1 Address Adjustment

The granularity of hard disk accesses is a sector of 128 words (512 bytes). Hence, a single pages contains PG_SZ/512 = 8 sectors. A swap memory address computed by the function $\mathit{spx}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{vpx}, \mathit{pid})$ is measured in pages. In order to convert it into a sector address we introduce the function

$$\mathit{page2sec}(\mathit{ad}) \quad = \quad \mathit{ad} \cdot 8.$$

A conversion in opposite direction, i.e., from sector addresses to page addresses is performed by the function

$$\mathit{sec2page}(\mathit{ad}) \quad = \quad \mathit{ad}/8.$$

As depicted in Figure 4.1 the swap memory component $c_{\mathrm{X}}.\mathit{swap}$ of the extended state models the part of the hard disk swap memory above the boot region.

In order to use the swap memory page address to access the extended state we have to compensate the offset of BOOT_PGS. For this we define the following function over a page address *ad*:

DEFINITION 4.24 ▶ Swap memory address adjustment

$$\mathit{offs}_{\mathrm{swap}}(\mathit{ad}) \quad = \quad \mathit{ad} - \texttt{BOOT\_PGS}.$$

Similarly, the physical memory component $c_{\mathrm{X}}.mem$ of the extended state models the user part of the physical memory and the zero filled page. Addresses in the extended state component $c_{\mathrm{X}}.mem$ start from zero. The following function is used to convert page addresses in real physical memory to those in $c_{\mathrm{X}}.mem$ — we shift the former by the pages address of the zero filled page:

$$\mathit{offs}_{\mathrm{mem}}(ad) \quad = \quad ad - \mathtt{ZFP}.$$

◀ DEFINITION 4.25
Physical memory address adjustment

4.4.2 Swap In

In the demand paging implementation a page is transferred from the hard disk into the physical memory by means of the function `read_from_disk`, one of the hard disk driver functions. On the abstract side we incorporate the effects of this function into an XCall. The semantics of this XCall operating on the extended state is introduced in the following definition.

◀ DEFINITION 4.26
Semantics of reading from disk

Let c_{X} be an extended state, let ma be a physical memory page address, and let sa be a disk sector address. The semantics of reading from the hard disk is defined by the function $\mathit{read\text{-}from\text{-}disk} :: \mathbb{N} \times \mathbb{N} \times C_{\mathrm{X}} \mapsto C_{\mathrm{X}}$ which returns an updated extended state $c'_{\mathrm{X}} = \mathit{read\text{-}from\text{-}disk}(ma, sa, c_{\mathrm{X}})$ such that the page in the physical memory component of the extended state at the address ma is replaced by the swap memory page taken at the address sa converted to the page address:

$$c'_{\mathrm{X}}.mem[i] = \begin{cases} c_{\mathrm{X}}.\mathit{swap}[\mathit{offs}_{\mathrm{swap}}(\mathit{sec2page}(sa))] & \text{if } i = \mathit{offs}_{\mathrm{mem}}(ma) \\ c_{\mathrm{X}}.mem[i] & \text{otherwise.} \end{cases}$$

Note, that ma and sa address pages in the real physical memory of the underlying machine and the hard disk content. Therefore respective address adjustments with the function $\mathit{offs}_{\mathrm{mem}}$ and $\mathit{offs}_{\mathrm{swap}}$ for translating them to the extended state take place.

The page-fault handler calls the function `read_from_disk` trough a wrapper function `pfh_swap_in`. The latter computes the swap memory address and calls the hard disk driver.

◀ DEFINITION 4.27
Semantics of swap in

Let c_{PFH} be an abstract PFH configuration, let c_{X} be an extended state, let pid be a process identifier, let ppx be a physical page index, and let vpx be a virtual page index. The semantics of swap in is defined by the function $\mathit{swap\text{-}in} :: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times C_{\mathrm{PFH}} \times C_{\mathrm{X}} \mapsto C_{\mathrm{X}}$ which returns an updated extended state $c'_{\mathrm{X}} = \mathit{swap\text{-}in}(pid, ppx, vpx, c_{\mathrm{PFH}}, c_{\mathrm{X}})$ such that the page in the physical memory component of the extended state at the page address ppx is replaced by the swap memory page taken at the computed swap memory address:

$$c'_{\mathrm{X}}.mem[i] = \begin{cases} c_{\mathrm{X}}.\mathit{swap}[\mathit{offs}_{\mathrm{swap}}(\mathit{spx}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, vpx, pid))] & \text{if } i = \mathit{offs}_{\mathrm{mem}}(ppx) \\ c_{\mathrm{X}}.mem[i] & \text{otherwise.} \end{cases}$$

Address adjustments with the function $\mathit{offs}_{\mathrm{mem}}$ and $\mathit{offs}_{\mathrm{swap}}$ for translating them to the extended state take place.

4.4.3 Swap Out

In our demand paging implementation pages are transferred in the opposite direction, i.e., from the physical memory to the hard disk, by means of the function `write_to_disk`, the second of the hard disk driver functions. The following definition introduces the semantics of the XCall corresponding to the function. The XCall operates on the extended state.

DEFINITION 4.28 ► Semantics of writing to disk

Let c_{X} be an extended state, let *ma* be a physical memory page address, and let *sa* be a disk sector address. The semantics of writing to the hard disk is defined by the function *write-to-disk* $:: \mathbb{N} \times \mathbb{N} \times C_{\mathrm{X}} \mapsto C_{\mathrm{X}}$ which returns an updated extended state $c'_{\mathrm{X}} = \mathit{write\text{-}to\text{-}disk}(ma, sa, c_{\mathrm{X}})$ such that the page in the swap memory component at the address *sa* converted to the page address is replaced by the physical memory page taken at the address *ma*. Necessary address shifts with the functions $\mathit{offs}_{\mathrm{mem}}$ and $\mathit{offs}_{\mathrm{swap}}$ take place. Formally:

$$c'_{\mathrm{X}}.\mathit{swap}[i] = \begin{cases} c_{\mathrm{X}}.\mathit{mem}[\mathit{offs}_{\mathrm{mem}}(ma)] & \text{if } i = \mathit{offs}_{\mathrm{swap}}(\mathit{sec2page}(sa)) \\ c_{\mathrm{X}}.\mathit{swap}[i] & \text{otherwise.} \end{cases}$$

The page-fault handler calls the function `write_to_disk` trough a wrapper function `pfh_swap_out`. The latter computes the swap memory address and calls the hard disk driver.

DEFINITION 4.29 ► Semantics of swap out

Let c_{PFH} be an abstract PFH configuration, let c_{X} be an extended state, let *pid* be a process identifier, let *ppx* be a physical page index, and let *vpx* be a virtual page index. The semantics of swap out is defined by the function *swap-out* $:: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times C_{\mathrm{PFH}} \times C_{\mathrm{X}} \mapsto C_{\mathrm{X}}$ which returns an updated extended state $c'_{\mathrm{X}} = \mathit{swap\text{-}out}(pid, ppx, vpx, c_{\mathrm{PFH}}, c_{\mathrm{X}})$ such that the page in the swap memory components of the extended state at the computed swap memory address is replace by the physical memory page taken at the page address *ppx*. Necessary address shifts with the functions $\mathit{offs}_{\mathrm{mem}}$ and $\mathit{offs}_{\mathrm{swap}}$ take place. Formally:

$$c'_{\mathrm{X}}.\mathit{swap}[i] = \begin{cases} c_{\mathrm{X}}.\mathit{mem}(\mathit{offs}_{\mathrm{mem}}(ppx)) & \text{if } i = \mathit{offs}_{\mathrm{swap}}(\mathit{spx}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, vpx, pid)) \\ c_{\mathrm{X}}.\mathit{swap}[i] & \text{otherwise.} \end{cases}$$

4.4.4 Filling a Page with Zeros

DEFINITION 4.30 ► Semantics of zero fill page

Let c_{X} be an extended state, and let *ppx* be a physical page index. The semantics of page zero fill is defined by the function *zero-fill-page* $:: \mathbb{N} \times C_{\mathrm{X}} \mapsto C_{\mathrm{X}}$ which returns an updated extended state $c'_{\mathrm{X}} = \mathit{zero\text{-}fill\text{-}page}(ppx, c_{\mathrm{X}})$ such that the page in the physical memory component of the extended state at the page address *ppx* is filled with zeros; each of `PG_SZ_WD` zeros is represented as a four-byte natural number:

$$c'_{\mathrm{X}}.\mathit{mem}[i] = \begin{cases} \mathit{rep}(0, \texttt{PG\_SZ\_WD}) & \text{if } i = \mathit{offs}_{\mathrm{swap}}(ppx) \\ c_{\mathrm{X}}.\mathit{mem}[i] & \text{otherwise.} \end{cases}$$

4.5 Page-Fault Handling Algorithm

For natural numbers x and pos the function

$$\textit{set-bit}(x, pos) \quad = \quad x - x \bmod 2^{pos+1} + 2^{pos} + x \bmod 2^{pos}$$

sets the bit at the position pos of x. The bit is cleared by means of the function

$$\textit{clear-bit}(x, pos) \quad = \quad x - x \bmod 2^{pos+1} + x \bmod 2^{pos}.$$

Table 4.2 depicts how we use these two functions for setting and clearing valid, protected, and executable bits.

Suppose, a page fault takes place in the abstract PFH configuration c_{PFH} for a process identifier *pid*, virtual address *va*, and intention *intent*, i.e., $pf_{\text{PFH}}?(c_{\text{PFH}}, pid, va, intent)$. The goal of the page-fault handler is to transit to configuration c'_{PFH} such that the page fault predicate does not hold in c'_{PFH}. The transition is defined by a page-fault handling algorithm specified below.

The algorithm operates on two configurations: the abstract PFH configuration and the extended state. Modifications of the former specify a page replacement strategy together with validation and invalidation of necessary page table entries. Changes in the extended state define page transfer between physical and swap memory.

Table 4.2: Setting and clearing bits of a page table entry

Function	Definition	Meaning
set-valid(*pte*)	*set-bit*(*pte*, `VALID_POS`)	set valid bit
clear-valid(*pte*)	*clear-bit*(*pte*, `VALID_POS`)	clear valid bit
set-prot(*pte*)	*set-bit*(*pte*, `VALID_POS`)	set protected bit
set-exec(*pte*)	*set-bit*(*pte*, `EXEC_POS`)	set executable bit

4.5.1 Updating Data Structures

The abstract PFH configuration is brought to a non page faulting state c'_{PFH} by means of the function $\textit{handle-pf}_{\text{PFH}} :: C_{\text{PFH}} \times \mathbb{N} \times \mathbb{N} \mapsto C_{\text{PFH}}$ such that $c'_{\text{PFH}} = \textit{handle-pf}_{\text{PFH}}(c_{\text{PFH}}, pid, va)$. The function defines the following actions.

First of all a descriptor *vict* of a page for eviction from the physical memory is selected. 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 the free list is empty, the page at the head of the active lists evicted:

$$\textit{vict} = \begin{cases} c_{\text{PFH}}.\textit{active}[0] & \text{if } c_{\text{PFH}}.\textit{free} = [\,] \\ c_{\text{PFH}}.\textit{free}[0] & \text{otherwise.} \end{cases}$$

Update of the active and free lists specify the page replacement strategy. 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 the page-fault handler's liveness since a single instruction can cause up to two page fault on the physical machine — one during the fetch phase, the other during a load or store operation.

The active list is updated as follows. In case the free list is empty there is no page in the physical memory which could be immediately given to the page faulting process. Following the selected FIFO strategy we remove the element at the head of the active list. By this we obtain a vacant place in the active list. In case the free list is not empty there is at least one vacant place in the active list. The reason is that the sum of the active and free lists lengths is always equal to the total number of user pages. Now, we insert a descriptor *pd* of the page that is swapped in at the end of the (so far partially) updated active list. We assign the process identifier and virtual page index fields of *pd* those values that caused a page fault: $pd.pid = pid$ and $pd.vpx = px(va)$. The physical page index field of *pd* is assigned the page index of the evicted page: $pd.ppx = vict.ppx$. Formally:

DEFINITION 4.31 ▶
Updating active list during page-fault handling

$$c'_{\mathrm{PFH}}.active = \begin{cases} tl(c_{\mathrm{PFH}}.active) \circ pd & \text{if } c_{\mathrm{PFH}}.free = [\,] \\ c_{\mathrm{PFH}}.active \circ pd & \text{otherwise.} \end{cases}$$

The free list update completes the specification of the page replacement policy. In case the free list is empty nothing happens. Otherwise, its head element is removed:

DEFINITION 4.32 ▶
Updating free list during page-fault handling

$$c'_{\mathrm{PFH}}.free = \begin{cases} c_{\mathrm{PFH}}.free & \text{if } c_{\mathrm{PFH}}.free = [\,] \\ tl(c_{\mathrm{PFH}}.free) & \text{otherwise.} \end{cases}$$

The modifications of the active and free lists have to be reflected on the page tables: the entry of the page that is swapped out has to be invalidated while the entry which corresponds to the process identifier and virtual address at which the page fault is being handled must be validated.

The predicate $invalid\text{-}ptea?(i, j)$ holds if i and j are indices of the entry for invalidation in the page table array $c_{\mathrm{PFH}}.pt$. The indices correspond to the page table entry addresses for both dimensions computed for the process identifier and virtual addresses of the page subject to eviction:

$$\begin{array}{lcl} invalid\text{-}ptea?(i,j) & = & (i = ptea_1(c_{\mathrm{PFH}}, vict.pid, vict.vpx) \\ & \wedge & j = ptea_2(vict.vpx)). \end{array}$$

The predicate $valid\text{-}ptea?(i, j)$ holds if i and j are indices of the entry for validation in the page table array $c_{\mathrm{PFH}}.pt$. The indices correspond to the page table entry addresses for both dimensions computed for the page faulting process identifier and virtual address:

$$\begin{array}{lcl} valid\text{-}ptea?(i,j) & = & i = ptea_1(c_{\mathrm{PFH}}, pid, va) \\ & \wedge & j = ptea_2(va) \end{array}.$$

The page table array elements $c_{\mathrm{PFH}}.pt[i][j]$ are updated in a straightforward way. In case the free list is empty the entry with $invalid\text{-}ptea?(i, j)$ is invalidated by clearing its valid bit. The entry with $valid\text{-}ptea?(i, j)$ is validated by setting its valid bit. The executable bit is raised as well. All other entries remain unchanged. Formally:

DEFINITION 4.33 ▶
Updating page table space during page-fault handling

$$c'_{\mathrm{PFH}}.pt[i][j] = \begin{cases} clear\text{-}valid(c_{\mathrm{PFH}}.pt[i][j]) & \text{if } invalid\text{-}ptea?(i,j) \wedge c_{\mathrm{PFH}}.free = [\,] \\ set\text{-}exec(set\text{-}valid(vict.ppx \cdot \texttt{PG\_SZ})) & \text{if } valid\text{-}ptea?(i,j) \\ c_{\mathrm{PFH}}.pt[i][j] & \text{otherwise.} \end{cases}$$

4.5.2 Updating Extended State

On the side of the extended state a page fault is handled by means of the function $\mathit{handle\text{-}pf}_{\mathrm{X}} :: C_{\mathrm{PFH}} \times C_{\mathrm{X}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto C_{\mathrm{X}}$ which returns an updated extended state $c'_{\mathrm{X}} = \mathit{handle\text{-}pf}_{\mathrm{X}}(c_{\mathrm{PFH}}, c_{\mathrm{X}}, \mathit{pid}, \mathit{vpx}, \mathit{intent})$. The function specifies page transfer between the physical memory and the swap memory during treatments of page faults.

The extended state update must be consistent with the abstract PFH configuration update. This results in two facts: (i) the page pointed to by the page table entry invalidated during the abstract PFH configuration update must be swapped out from the physical memory, and (ii) the page pointed to by the page table entry validated during the abstract PFH configuration update must be either filled with zeros, filled with the swapped in data, or intentionally left unchanged.

First, we formally define the second fact. The data which is written to the validated page depends on the kind of a page fault: an invalid access or zero protection. For an easier distinction between them we introduce the predicate $\mathit{zfp?}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx})$. It holds if the page index of the page table entry which corresponds to the process identifier *pid* and the virtual page index *vpx* points to the zero filled page:

$$\mathit{zfp?}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx}) \quad = \quad \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx})) = \texttt{ZFP}.$$

Recall that the parameter *intent* defines the intention of a page-fault handler's call. In case it equals `OVERWRITE`, denoted by $\mathit{owr?}(\mathit{intent})$, the caller is going to overwrite the entire physical page which corresponds to *pid* and *vpx* by some data[1]. Therefore, we can optimize the algorithm by keeping the original data at that page. If intention is different from overwriting the physical memory component of the extended state is modified at the page address *vict.ppx* by filling the page with zeros, in case $\mathit{zfp?}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx})$, or with the swapped in data, otherwise.

Formally, the possible cases of a page filling are defined by the following function:

◀ DEFINITION 4.34
Page filling during page-fault handling

$$\mathit{fill\text{-}page}(c_{\mathrm{PFH}}, c_{\mathrm{X}}, \mathit{pid}, \mathit{vpx}, \mathit{intent}) = \begin{cases} \mathit{zero\text{-}fill\text{-}page}(\mathit{vict.ppx}, c_{\mathrm{X}}) & \text{if } \neg\mathit{owr?}(\mathit{intent}) \wedge \mathit{zfp?}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx}) \\ \mathit{swap\text{-}in}(\mathit{pid}, \mathit{vict.ppx}, \mathit{vpx}, c_{\mathrm{PFH}}, c_{\mathrm{X}}) & \text{if } \neg\mathit{owr?}(\mathit{intent}) \wedge \neg\mathit{zfp?}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx}) \\ c_{\mathrm{X}} & \text{otherwise.} \end{cases}$$

As for the first fact, we use the semantics of swap out with the process identifier, physical and virtual page indicies parameters taken from the descriptor *vict* of the page for eviction. By that we obtain an intermediate configuration $\hat{c_{\mathrm{X}}}$ of the extended state in which the invalidated page is swapped out:

$$\hat{c_{\mathrm{X}}} = \mathit{swap\text{-}out}(\mathit{vict.pid}, \mathit{vict.ppx}, \mathit{vict.vpx}, c_{\mathrm{PFH}}, c_{\mathrm{X}}).$$

Swap out takes place only if the free list is empty.

Altogether, we define the configuration c'_{X} of the extended state after page-fault handling. The result configuration c'_{X} is defined by the page filling function applied to the intermediate configuration $\hat{c_{\mathrm{X}}}$, in case the free list is empty, or to the original configuration c_{X}, otherwise:

[1] An example of such a call occurs in the CVM primitive `cvm_copy` (cf. Section 9.3 of Tsyban's thesis [Tsy09]).

DEFINITION 4.35 ▶
Updating extended state during page-fault handling

$$c'_{\mathrm{X}} = \begin{cases} \mathit{fill\text{-}page}(c_{\mathrm{PFH}}, \hat{c_{\mathrm{X}}}, \mathit{pid}, \mathit{vpx}, \mathit{intent}) & \text{if } c_{\mathrm{PFH}}.\mathit{free} = [\,] \\ \mathit{fill\text{-}page}(c_{\mathrm{PFH}}, c_{\mathrm{X}}, \mathit{pid}, \mathit{vpx}, \mathit{intent}) & \text{otherwise.} \end{cases}$$

4.5.3 Prevention of Most Recently Loaded Page Swap Out

Our demand paging system supports features for close interaction with CVM. The CVM implementation includes a primitive for copying data between two user processes. This primitive needs that both source and destination pages reside in the physical memory. However, if the page-fault handler just checks that the page is currently in the memory, a subsequent call to the handler might invalidate it. In order to overcome this problem, the parameter `count` with a default value of zero is introduced to the page-fault handler. When it is set to one the page-fault handler ensures that the specified page will survive the second call of the handler. In case the descriptor of the considered page is the head of the active list we need to push it one one position further inside the active list. An updated abstract PFH configuration c'_{PFH} in this case is obtained by means of the function *prevent-swap-out* :: $C_{\mathrm{PFH}} \mapsto C_{\mathrm{PFH}}$ which modifies only the active list components of its argument by swapping its first and second elements:

DEFINITION 4.36 ▶
Preventing page swap out

$$c'_{\mathrm{PFH}}.\mathit{active} = \mathit{hd}(\mathit{tl}(c_{\mathrm{PFH}}.\mathit{active})) \circ \mathit{hd}(c_{\mathrm{PFH}}.\mathit{active}) \circ \mathit{tl}(\mathit{tl}(c_{\mathrm{PFH}}.\mathit{active})).$$

4.6 Validity

We demand a variety of properties to hold for abstract PFH configurations. These properties reflect the functional correctness and are necessary for the top-level correctness proof of the page-fault handler. All of the properties are established for the first time after an execution of the page-fault handler initialization code and are preserved under calls to the handler. Therefore, we will refer to these properties as *validity invariants* of the page-fault handler.

Whenever we introduce such an invariant further in this section we introduce a lemma for it as well. Each lemma assumes that a page-fault takes place in a configuration c_{PFH} for a process identifier *pid* at virtual address *va* and claims that the invariant is preserved under the page-fault handling algorithm. Hence, each lemma states that the invariant holds in the configuration $\mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va})$.

Certainly, each validity invariant holds in the initial configuration $\mathit{init}\text{-}c_{\mathrm{PFH}}$ of the page-fault handler. However, due to the monotony of such lemmas we omit their statement. We rather present a single lemma in the end of this section claiming that all validity invariants hold in the initial configuration.

4.6.1 Invariants about Active and Free Lists

Active list describes only user processes. Recall that our system supports `MAX_PID` processes. The process with identifier $\mathit{pid} = 0$ is an operating-system kernel; all other identifiers denote user processes. We call *pid* a user process

identifier if it points to a user process and denote by the predicate

$$\textit{user-pid?}(\textit{pid}) \quad = \quad 0 < \textit{pid} < \texttt{MAX\_PID}.$$

As we support virtual memory only for the latter the above property must hold for all elements of the active list:

$$\textit{user-pid-active?}(c_{\text{PFH}}) \;=\; \forall\, i < |c_{\text{PFH}}.\textit{active}| : \textit{user-pid?}(c_{\text{PFH}}.\textit{active}[i].\textit{pid}).$$

◀ DEFINITION 4.37

Active list describes only user memory. We call *ppx* a user physical page index if its values lies with the user memory region, i.e., `USER_PGS` starting from the page address `KERNEL_PGS`, and denote by

$$\textit{user-ppx?}(\textit{ppx}) \quad = \quad \texttt{KERNEL\_PGS} \leq \textit{ppx} < \texttt{TOT\_PHYS\_PGS}.$$

This property holds for all elements of the active and free lists — they have their physical page index field inside the user memory range:

$$\textit{user-ppx-active-free?}(c_{\text{PFH}}) = \forall\, i < \texttt{USER\_PGS} : \textit{user-ppx?}((c_{\text{PFH}}.\textit{active} \circ c_{\text{PFH}}.\textit{free})[i].\textit{ppx}).$$

◀ DEFINITION 4.38

Active list describes only valid pages. A page descriptor is inserted in the active list in case the memory page pointed to by it is either already in the physical memory, or is supposed to be transferred to the main memory from the hard disk within the current call to the page-fault handler. The corresponding page table entry is valid afterwards. Therefore, all page table entries pointed to by the active list have their valid bits set:

$$\textit{valid-active?}(c_{\text{PFH}}) \quad = \quad \forall\, i < |c_{\text{PFH}}.\textit{active}| : \textit{valid?}(\textit{pte}_{\text{PFH}}(c_{\text{PFH}}, c_{\text{PFH}}.\textit{active}[i].\textit{pid}, c_{\text{PFH}}.\textit{active}[i].\textit{vpx})).$$

◀ DEFINITION 4.39

Active list describes only not protected pages. As it follows from Definition 4.39 the active list describes only valid pages. A page might be validated only during a call to the page-fault handler. According to the page-fault handling algorithm when a page table entry is validated only its valid and executable bits are raised. The protected bit is kept cleared. Hence, all elements of the active list describe page table entries which have their protected bits cleared:

$$\textit{not-prot-active?}(c_{\text{PFH}}) \quad = \quad \forall\, i < |c_{\text{PFH}}.\textit{active}| : \neg\textit{prot?}(\textit{pte}_{\text{PFH}}(c_{\text{PFH}}, c_{\text{PFH}}.\textit{active}[i].\textit{pid}, c_{\text{PFH}}.\textit{active}[i].\textit{vpx})).$$

◀ DEFINITION 4.40

Active list points to physical page indices. The physical page index field of active list's elements must point exactly to the physical page associated with a given process identifier and virtual page index. In other words, it must be equal to the page index of the page table entry computed for this process identifier and virtual page index.

$$\textit{ppx-active-eq-ppx-pt?}(c_{\text{PFH}}) \quad = \quad \forall\, i < |c_{\text{PFH}}.\textit{active}| : c_{\text{PFH}}.\textit{active}[i].\textit{ppx} = \textit{px}(\textit{pte}_{\text{PFH}}(c_{\text{PFH}}, c_{\text{PFH}}.\textit{active}[i].\textit{pid}, c_{\text{PFH}}.\textit{active}[i].\textit{vpx})).$$

◀ DEFINITION 4.41

Active and free lists do not point to zero-filled page. We allocate a page of virtual memory at page address *vpx* for a process *pid* by making the page table entry computed for *vpx* and *pid* point to the (shared) zero-filled page. A write access to this newly allocated virtual page generates a zero protection page fault. The page-fault handler assigns a real physical page to the pair (*pid*, *vpx*) and inserts this entry to the active list. By this, no descriptors pointing to the zero-filled page might occur in the active list. Initially, no elements of the free list point to the zero-filled page. Since only descriptors from the active list might be inserted in the free lists during our system run, there are no elements referring to the zero-filled page in the free list as well:

DEFINITION 4.42 ▶
$$\textit{ppx-active-free-not-zfp?}(c_{\mathrm{PFH}}) = \forall\, i < \mathtt{USER\_PGS} : (c_{\mathrm{PFH}}.\textit{active} \circ c_{\mathrm{PFH}}.\textit{free})[i].\textit{ppx} \neq \mathtt{ZFP}.$$

Active lists contains only distinct pairs (*pid*, *vpx*). None of the virtual pages of a particular user process might be stored by two or more different active pages. In other words, pairs of the form (*pid*, *vpx*) must be distinct within the active list:

DEFINITION 4.43 ▶
$$\textit{dstnct-pid-vpx-active?}(c_{\mathrm{PFH}}) = \forall\, i, j < |c_{\mathrm{PFH}}.\textit{active}|, i \neq j : c_{\mathrm{PFH}}.\textit{active}[i].(\textit{pid}, \textit{vpx}) \neq c_{\mathrm{PFH}}.\textit{active}[j].(\textit{pid}, \textit{vpx}).$$

Active and free lists have distinct physical page indices. A physical page cannot by shared by two or more virtual pages. The following invariant guarantees that there are no two page descriptors in the active list that point to the same physical page:

DEFINITION 4.44 ▶
$$\textit{dstnct-ppx-active-free?}(c_{\mathrm{PFH}}) = \forall\, i, j < \mathtt{USER\_PGS}, i \neq j : (c_{\mathrm{PFH}}.\textit{active} \circ c_{\mathrm{PFH}}.\textit{free})[i].\textit{ppx} \neq (c_{\mathrm{PFH}}.\textit{active} \circ c_{\mathrm{PFH}}.\textit{free})[j].\textit{ppx}.$$

Active list respects page table lengths. Recall that the page table lengths component of abstract PFH configurations encode amount of virtual memory allocated for processes. A process *pid* has $c_{\mathrm{PFH}}.\textit{ptl}[\textit{pid}]+1$ pages of virtual memory. Virtual page indices of active list's elements must be bounded from above by the respective virtual memory amount:

DEFINITION 4.45 ▶
$$\textit{vpx-less-ptl-active?}(c_{\mathrm{PFH}}) = \forall\, i < |c_{\mathrm{PFH}}.\textit{active}| : c_{\mathrm{PFH}}.\textit{active}[i].\textit{vpx} < c_{\mathrm{PFH}}.\textit{ptl}[c_{\mathrm{PFH}}.\textit{active}[i].\textit{pid}]+1.$$

Valid page table entries describe active list. The correspondence between page table entries which point to pages currently located in the physical memory and elements of the active list is given by the following invariant. Consider all user process identifiers *pid* and virtual page indices *vpx* such that the latter is bounded by the amount of virtual memory the corresponding process has. If

the valid bit of the page table entry computed for *pid* and *vpx* is raised and this entry does not point to the zero-filled page, then there is a page descriptor in the active lists such that its process identifier field equals *pid*, its virtual page index fields equals *vpx*, and its physical page index field equals the page index of the computed page table entry:

◀ DEFINITION 4.46

$$\begin{aligned}
&\mathit{valid\text{-}pte\text{-}descr\text{-}active?}(c_{\mathrm{PFH}}) = \\
&\qquad \forall\, 0 < \mathit{pid} < \mathtt{MAX_PID}, \mathit{vpx} < c_{\mathrm{PFH}}.\mathit{ptl}[\mathit{pid}] + 1 : \\
&\qquad \mathit{valid?}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx})) \\
&\qquad \wedge\, \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx})) \neq \mathtt{ZFP} \\
&\qquad \longrightarrow \exists\, i < |c_{\mathrm{PFH}}.\mathit{active}| : \\
&\qquad\qquad c_{\mathrm{PFH}}.\mathit{active}[i].\mathit{pid} = \mathit{pid} \\
&\qquad\qquad \wedge\; c_{\mathrm{PFH}}.\mathit{active}[i].\mathit{vpx} = \mathit{vpx} \\
&\qquad\qquad \wedge\; c_{\mathrm{PFH}}.\mathit{active}[i].\mathit{ppx} = \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{vpx})).
\end{aligned}$$

4.6.2 Invariants about Page Tables

Page tables refer only to user memory. User processes, certainly, must not modify kernel code and data. User processes operate with virtual addresses which are translated into physical memory addresses. The latter addresses must lie outside the kernel range. In order to guarantee that, page indices of page table entries must also have values outside the kernel range. However, we do not impose this restriction on all page table entries, but rather on those that are valid and do not point to the zero-filled page, and, therefore, can be used for address translation. We use the predicate *user-ppx?* in order to limit page indices in page tables:

◀ DEFINITION 4.47

$$\begin{aligned}
\mathit{user\text{-}ppx\text{-}pt?}(c_{\mathrm{PFH}}) \quad = \quad & \forall\, i < \mathtt{TOT_PGS_PT}, j < \mathtt{PTES_PER_PG} : \\
& \mathit{valid?}(c_{\mathrm{PFH}}.\mathit{pt}[i][j]) \wedge \mathit{px}(c_{\mathrm{PFH}}.\mathit{pt}[i][j]) \neq \mathtt{ZFP} \\
\longrightarrow \quad & \mathit{user\text{-}ppx?}(\mathit{px}(c_{\mathrm{PFH}}.\mathit{pt}[i][j])).
\end{aligned}$$

Page tables do not overlap. Our system features separate address spaces of user processes. Therefore page tables must not overlap. For any given pair of page tables of user processes, either one must end before the other starts or vice versa. The page table of a process i starts in the page table space at the origin $\mathit{pto}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, i)$. This page table has $\mathit{ptl}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, i)$ entries which constitute its length. The sum of the origin and the length has to be less than the origin of the next process. We express this more formally in the following definition:

◀ DEFINITION 4.48

$$\begin{aligned}
\mathit{not\text{-}overlap\text{-}pt?}(c_{\mathrm{PFH}}) \quad = \quad & \forall\, j < \mathtt{MAX_PID}, 0 < i < j : \\
& \mathit{pto}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, i) + \mathit{ptl}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, i) < \mathit{pto}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, j).
\end{aligned}$$

Page tables do not exceed page table space. A slightly different invariant ensures that all page tables lie within the boundaries of the page table space:

◀ DEFINITION 4.49

$$\begin{aligned}
&\mathit{pto\text{-}ptl\text{-}less\text{-}pt?}(c_{\mathrm{PFH}}) = \\
&\qquad \forall\, 0 < i < \mathtt{MAX_PID} : \mathit{pto}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, i) + \mathit{ptl}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, i) < \mathtt{TOT_PGS_PT}.
\end{aligned}$$

Protected pages point to zero-filled page. In our kernel implementation the protected bit is set only for page table entries of newly allocated pages. These entries point to the zero-filled page. A write access to a page described by such page table entry triggers a zero protection page fault. When handling it we transfer a real physical page into the main memory and make the respective page table entry point to it. At the same time we clear the protected bit. By this an invariant arises that the protected bit is equivalent to the fact that a page table entry points to the zero-filled page:

DEFINITION 4.50 ▶
$$\textit{zfp-eq-prot-pt?}(c_{\mathrm{PFH}}) = \forall\, i < \texttt{TOT\_PGS\_PT}, j < \texttt{PTES\_PER\_PG}: \textit{px}(c_{\mathrm{PFH}}.\textit{pt}[i][j]) = \texttt{ZFP} \longleftrightarrow \textit{prot?}(c_{\mathrm{PFH}}.\textit{pt}[i][j]).$$

Valid pages are executable. On the hardware level page faults occur in case (among others) the executable bit is not set on a fetched input. However, by design of our kernel we do not treat such case as a page fault on the software level. We solve the problem of inconsistency between page faults on the hardware and software levels by implementing the kernel in a way that the executable bit is set for all valid page table entries. The following invariant defines this formally:

DEFINITION 4.51 ▶
$$\textit{valid-is-exec-pt?}(c_{\mathrm{PFH}}) = \forall\, i < \texttt{TOT\_PGS\_PT}, j < \texttt{PTES\_PER\_PG}: \textit{valid?}(c_{\mathrm{PFH}}.\textit{pt}[i][j]) \longrightarrow \textit{exec?}(c_{\mathrm{PFH}}.\textit{pt}[i][j]).$$

Entries that point to zero-filled page are valid. When allocating a new page for a process we make its page table entry to point to the zero-filled page. At the same time we set the valid bit for this entry. Since in our implementation of demand paging while a page table entry points to the zero-filled page no modifications of its valid bit are done the following invariant holds:

DEFINITION 4.52 ▶
$$\textit{zfp-is-valid-pt?}(c_{\mathrm{PFH}}) = \forall\, i < \texttt{TOT\_PGS\_PT}, j < \texttt{PTES\_PER\_PG}: \textit{px}(c_{\mathrm{PFH}}.\textit{pt}[i][j]) = \texttt{ZFP} \longrightarrow \textit{valid?}(c_{\mathrm{PFH}}.\textit{pt}[i][j]).$$

4.6.3 Invariants about Big-Page Tables

Free and used big pages constitute all big pages. The length of the free big pages stack shows the total number of free big-pages. The total number of used big pages could be computed as follows. For a process *pid* the number of big pages consumed by it is $c_{\mathrm{PFH}}.\textit{bptl}[\textit{pid}] + 1$. If the process consumes no big-pages this sum is equal to zero. We sum the equation for all user process identifiers from 1 to $\texttt{MAX\_PID} - 1$ and obtain the desired total number of used big pages. Now, the following invariant states that the sum of all free big pages and all used big pages equal to the total number of big pages:

DEFINITION 4.53 ▶
$$\textit{total-bpages?}(c_{\mathrm{PFH}}) = |c_{\mathrm{PFH}}.\textit{bpfree}| + \sum_{i=1}^{\texttt{MAX\_PID}-1} (c_{\mathrm{PFH}}.\textit{bptl}[i] + 1) = \texttt{TOT\_BIG\_PGS}.$$

Big-page table entries are distinct. Free big page table entries are stored in the stack of free big pages. Next, we formally define a list of used big page table entries. A process *pid* occupies entries in the big page table at consecutive address from $c_{\text{PFH}}.bpto[pid]$ up to $c_{\text{PFH}}.bpto[pid] + c_{\text{PFH}}.bptl[pid]$. We obtain the list of big-page table entries of the process *pid* by accessing the big-page table space at these addresses:

$$\begin{aligned} \textit{used-bpte-pid}(c_{\text{PFH}}, pid) = [&c_{\text{PFH}}.bpt[c_{\text{PFH}}.bpto[pid]], \\ &c_{\text{PFH}}.bpt[c_{\text{PFH}}.bpto[pid] + 1], \\ &\ldots, \\ &c_{\text{PFH}}.bpt[c_{\text{PFH}}.bpto[pid] + c_{\text{PFH}}.bptl[pid]]]. \end{aligned}$$

Having this, we can obtain all used big-page table entries by iterating the user process identifier from 1 to $\texttt{MAX\_PID} - 1$ and concatenating the results:

$$\begin{aligned} \textit{used-bpte}(c_{\text{PFH}}) \quad = \quad &\textit{used-bpte-pid}(c_{\text{PFH}}, 1) \circ \textit{used-bpte-pid}(c_{\text{PFH}}, 2) \circ \ldots \\ &\circ \textit{used-bpte-pid}(c_{\text{PFH}}, \texttt{MAX\_PID} - 1). \end{aligned}$$

Now, the requirement for all big-page table entries to be distinct is expressed as all elements of the list made of free and used big-page table entries are distinct.

$$\begin{aligned} \textit{dstnct-bpte?}(c_{\text{PFH}}) \quad = \quad &\forall\, i, j < |c_{\text{PFH}}.bpfree \circ \textit{used-bpte}(c_{\text{PFH}})|, i \neq j : \\ &(c_{\text{PFH}}.bpfree \circ \textit{used-bpte}(c_{\text{PFH}}))[i] \neq (c_{\text{PFH}}.bpfree \circ \textit{used-bpte}(c_{\text{PFH}}))[j]. \end{aligned}$$

◀ DEFINITION 4.54

Big-page tables do not exceed big-page table space. This invariant is a version of Definition 4.49 for the big-page table space:

$$\begin{aligned} \textit{bpto-bptl-rel?}(c_{\text{PFH}}) \quad = \quad &\forall\, 0 < pid < \texttt{MAX\_PID} : \\ &c_{\text{PFH}}.bpto[pid] + c_{\text{PFH}}.bptl[pid] + 1 \leq \texttt{TOT\_BIG\_PGS} \end{aligned}$$

◀ DEFINITION 4.55

4.6.4 Invariants about Process Control Blocks

Page table origins are monotonic. In order to support disjointness of page tables of different process we require page table origins to be monotonic:

$$\textit{pto-mono?}(c_{\text{PFH}}) \quad = \quad \forall\, j < \texttt{MAX\_PID}, 0 < i < j : c_{\text{PFH}}.pto[i] < c_{\text{PFH}}.pto[j].$$

◀ DEFINITION 4.56

Active processes have memory. By design the active list describes pages only of processes which have some allocated memory. Hence, page table lengths of such processes must be non-negative:

$$\begin{aligned} \textit{ptl-pid-alloc-active?}(c_{\text{PFH}}) \quad = \quad &\forall\, i < |c_{\text{PFH}}.active| : \\ &c_{\text{PFH}}.ptl[c_{\text{PFH}}.active[i].pid] \geq 0. \end{aligned}$$

◀ DEFINITION 4.57

Relation between page table origins and lengths. The page table origin filed of a process control block stores the start page address of the page table for the process. This start arrdess is counted from the very beginning of the physical memory. Taking into consideration the fact that the page table space starts at the page address PT_START/PG_SZ in the physical memory we can represent the

origins as a sum of this start address and some offset x. This constitute the first conjunct of the invariant below. The second conjunct impose a restriction on the choice of x. Recall that, $c_{\text{PFH}}.ptl[pid] + 1$ is the amount of process's *pid* virtual memory measured in pages. From the other side, this number also defines the number of page table entries occupied by the process. Therefore, $(c_{\text{PFH}}.ptl[pid] + 1)/\texttt{PTES\_PER\_PG}$ is the number of pages in the page table space occupied by the process *pid*. The sum of x and this number must not exceed the number of pages in the page table space:

DEFINITION 4.58 ▶

$$\begin{aligned} \textit{pto-ptl-rel?}(c_{\text{PFH}}) \quad = \quad & \forall\, 0 < pid < \texttt{MAX\_PID} : \exists\, x : \\ & c_{\text{PFH}}.pto[pid] = \texttt{PT\_START}/\texttt{PG\_SZ} + x \\ \wedge \quad & x + (c_{\text{PFH}}.ptl[pid] + 1)/\texttt{PTES\_PER\_PG} < \texttt{TOT\_PGS\_PT}. \end{aligned}$$

Big-page table lengths could be computed from page table lengths. Due to the ration PGS_PER_BIG_PG between pages and big pages big page tables lengths always could be computed from page table lengths via division by PGS_PER_BIG_PG. There is an exception from this rule if a process has no virtual memory. In this case both the page table length and the big page table length must be equal to -1.

DEFINITION 4.59 ▶

$$\begin{aligned} \textit{ptl-bptl-rel?}(c_{\text{PFH}}) \quad = \quad & \forall\, 0 < pid < \texttt{MAX\_PID} : \\ & c_{\text{PFH}}.ptl[pid] = -1 \wedge c_{\text{PFH}}.bptl[pid] = -1 \\ \vee \quad & c_{\text{PFH}}.ptl[pid] = c_{\text{PFH}}.bptl[pid] \cdot \texttt{PGS\_PER\_BIG\_PG}. \end{aligned}$$

4.6.5 Altogether

We collect the validity invariants presented in this chapter into a single predicate.

DEFINITION 4.60 ▶
Validity of abstract PFH configurations

The overall validity invariant of the abstract PFH configuration $pfh\surd(c_{\text{PFH}})$ is the conjunction of (i) individual validity invariants, Definitions 4.37–4.59, and (ii) sub-typing requirements, Definition 4.7.

In the remainder of this chapter we prove three lemmas stating that the page-fault handler validity predicate holds for the initial abstract PFH configuration and that it is preserved under page-fault handling and page swap out prevention operations.

4.6.6 Validity Proofs

Table 4.3: Active list invariants trivial for *init*-c_{PFH}

Invariant	Definition	Invariant	Definition
user-pid-active?	4.37	*dstnct-pid-vpx-active?*	4.43
valid-active?	4.39	*vpx-less-ptl-active?*	4.45
not-prot-active?	4.40	*ptl-pid-alloc-active?*	4.57
ppx-active-eq-ppx-pt?	4.41		

LEMMA 4.61 ▶
Initial PFH configuration is valid

The validity invariant is established for the first time for the initial abstract PFH configuration:

$$pfh\surd(\textit{init}\text{-}c_{\text{PFH}}).$$

- All sub-typing requirements (Definition 4.7) are trivial to show from the invariants proven below. ◀ PROOF

- Since $\mathit{init}\text{-}c_{\text{PFH}}.\mathit{active} = [\,]$ all properties about elements of the active list hold. These are listed in Table 4.3.

- The free list contains $\texttt{TOT\_PHYS\_PGS} - \texttt{KERNEL\_PGS} = \texttt{USER\_PGS}$ elements, and for all of the the following holds:

$$\begin{aligned}\forall\, i < \texttt{USER\_PGS}: \quad & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[i].\mathit{pid} = 0 \\ \wedge\ & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[i].\mathit{vpx} = 0 \\ \wedge\ & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[i].\mathit{ppx} = \texttt{TOT\_PHYS\_PGS} - i - 1.\end{aligned}$$

  From that we conclude:

$$\begin{aligned} & \texttt{KERNEL\_PGS} \leq \texttt{TOT\_PHYS\_PGS} - i - 1 < \texttt{KERNEL\_PGS} + \texttt{USER\_PGS} \\ = \quad & \texttt{KERNEL\_PGS} \leq \mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[i].\mathit{ppx} < \texttt{TOT\_PHYS\_PGS} \\ = \quad & \mathit{user\text{-}ppx?}(\mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[i].\mathit{ppx}) \\ = \quad & \mathit{user\text{-}ppx\text{-}active\text{-}free?}(\mathit{init}\text{-}c_{\text{PFH}}) \quad \text{(Definition 4.38)},\end{aligned}$$

$$\begin{aligned} & \texttt{KERNEL\_PGS} \leq \texttt{TOT\_PHYS\_PGS} - i - 1 \\ = \quad & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[i].\mathit{ppx} \neq \texttt{ZFP} \\ = \quad & \mathit{ppx\text{-}active\text{-}free\text{-}not\text{-}zfp?}(\mathit{init}\text{-}c_{\text{PFH}}) \quad \text{(Definition 4.42), and}\end{aligned}$$

$$\begin{aligned} & i \neq j \\ = \quad & \texttt{TOT\_PHYS\_PGS} - i - 1 \neq \texttt{TOT\_PHYS\_PGS} - j - 1 \\ = \quad & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[i].\mathit{ppx} \neq \mathit{init}\text{-}c_{\text{PFH}}.\mathit{free}[j].\mathit{ppx} \\ = \quad & \mathit{dstnct\text{-}ppx\text{-}active\text{-}free?}(\mathit{init}\text{-}c_{\text{PFH}}) \quad \text{(Definition 4.44)}.\end{aligned}$$

- For page table origins and lengths as well as their big versions we have the following:

$$\begin{aligned}\forall\, 0 < \mathit{pid} < \texttt{MAX\_PID}: \quad & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{pto}[\mathit{pid}] = \mathit{init}\text{-}\mathit{pto}(\mathit{pid}) \\ \wedge\ & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{ptl}[\mathit{pid}] = -1 \\ \wedge\ & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{bpto}[\mathit{pid}] = 0 \\ \wedge\ & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{bptl}[\mathit{pid}] = -1.\end{aligned}$$

  Exploiting this we conclude:

$$\begin{aligned} & \mathit{init}\text{-}\mathit{pto}(\mathit{pid}_1) < \mathit{init}\text{-}\mathit{pto}(\mathit{pid}_2) \\ = \quad & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{pto}[\mathit{pid}_1] < \mathit{init}\text{-}c_{\text{PFH}}.\mathit{pto}[\mathit{pid}_2] \\ = \quad & \mathit{pto\text{-}mono?}(\mathit{init}\text{-}c_{\text{PFH}}) \quad \text{(Definition 4.56)},\end{aligned}$$

$$\begin{aligned} & \mathit{init}\text{-}\mathit{pto}(\mathit{pid}) = \texttt{PT\_START}/\texttt{PG\_SZ} + \frac{(\mathit{pid}-1)\cdot\texttt{TOT\_PGS\_PT}}{\texttt{MAX\_PID}} \wedge \\ & \qquad \frac{(\mathit{pid}-1)\cdot\texttt{TOT\_PGS\_PT}}{\texttt{MAX\_PID}} < \texttt{TOT\_PGS\_PT} \\ = \quad & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{pto}[\mathit{pid}] = \texttt{PT\_START}/\texttt{PG\_SZ} + x \wedge \\ & \qquad x + (\mathit{init}\text{-}c_{\text{PFH}}.\mathit{ptl}[\mathit{pid}] + 1)/\texttt{PTES\_PER\_PG} < \texttt{TOT\_PGS\_PT} \\ = \quad & \mathit{pto\text{-}ptl\text{-}rel?}(\mathit{init}\text{-}c_{\text{PFH}}) \quad \text{(Definition 4.58)},\end{aligned}$$

$$\begin{aligned} & 0 + (-1) + 1 \leq \texttt{TOT\_BIG\_PGS} \\ = \quad & \mathit{init}\text{-}c_{\text{PFH}}.\mathit{bpto}[\mathit{pid}] + \mathit{init}\text{-}c_{\text{PFH}}.\mathit{bptl}[\mathit{pid}] + 1 \leq \texttt{TOT\_BIG\_PGS} \\ = \quad & \mathit{bpto\text{-}bptl\text{-}rel?}(\mathit{init}\text{-}c_{\text{PFH}}) \quad \text{(Definition 4.55), and}\end{aligned}$$

$$\begin{array}{ll} & \mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{ptl}[\mathit{pid}] = -1 \wedge \mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{bptl}[\mathit{pid}] = -1 \\ = & \mathit{ptl}\text{-}\mathit{bptl}\text{-}\mathit{rel}?(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.59).} \end{array}$$

- For a process *pid* the values of page table origin and length within the page table space are

$$\begin{array}{rcl} \mathit{pto}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}) & = & \dfrac{\mathit{init}\text{-}\mathit{pto}(\mathit{pid}) \cdot \mathtt{PG\_SZ} - \mathtt{PT\_START}}{4 \cdot \mathtt{PTES\_PER\_PG}} \\ & = & \dfrac{(\mathit{pid}-1) \cdot \mathtt{TOT\_PGS\_PT}}{\mathtt{MAX\_PID}}, \text{ and} \\ \mathit{ptl}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}) & = & \dfrac{-1+1}{\mathtt{PTES\_PER\_PG}} \; = \; 0, \text{ respectively.} \end{array}$$

  Hence, the following invariants hold:

$$\begin{array}{ll} & \mathit{pid}_1 < \mathit{pid}_2 \\ = & \mathit{pto}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}_1) < \mathit{pto}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}_2) \\ = & \mathit{pto}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}_1) + \mathit{ptl}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}_1) < \mathit{pto}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}_2) \\ = & \mathit{not}\text{-}\mathit{overlap}\text{-}\mathit{pt}?(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.48), and} \end{array}$$

$$\begin{array}{ll} & \mathit{pto}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}) < \mathtt{TOT\_PGS\_PT} \\ = & \mathit{pto}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}) + \mathit{ptl}_{\mathrm{PFH}}(\mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{pid}) < \mathtt{TOT\_PGS\_PT} \\ = & \mathit{pto}\text{-}\mathit{ptl}\text{-}\mathit{less}\text{-}\mathit{pt}?(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.49).} \end{array}$$

- Invariant *valid-pte-descr-active*? (Definition 4.46)holds since there is no virtual page index *vpx* for any process *pid* such that

$$\mathit{vpx} < \mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{ptl}[\mathit{pid}] + 1.$$

- The stack of free big-pages is initialized with the list of elements from 0 till $\mathtt{TOT\_BIG\_PGS} - 1$ such that

$$\forall\, i < \mathtt{TOT\_BIG\_PGS} : \mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{bpfree}[i] = i.$$

  Using the fact that no process has any memory allocated and $\mathit{bptl}[\mathit{pid}] = -1$ we conclude:

$$\begin{array}{ll} & \mathtt{TOT\_BIG\_PGS} \\ = & \mathtt{TOT\_BIG\_PGS} + \sum_{i=1}^{\mathtt{MAX\_PID}-1} 0 \\ = & |\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{bpfree}| + \sum_{i=1}^{\mathtt{MAX\_PID}-1} (\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{bptl}[i] + 1) \\ = & \mathit{total}\text{-}\mathit{bpages}?(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.53).} \end{array}$$

  Also, there are no used big-page table entries $\mathit{used}\text{-}\mathit{bpte}(\mathit{init}\text{-}c_{\mathrm{PFH}}) = [\,]$, therefore

$$\begin{array}{ll} & i \neq j \\ = & (\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{bpfree})[i] \neq (\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{bpfree})[j] \\ = & \mathit{dstnct}\text{-}\mathit{bpte}?(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.54).} \end{array}$$

- The remaining group of validity invariants speaks about the page table space. All its entries are initialized with zeros:

  $$\forall\, i < \texttt{TOT\_PGS\_PT}, j < \texttt{PTES\_PER\_PG} : \mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j] = 0.$$

  Hence, we conclude:

  $$\begin{array}{ll} & \neg \mathit{valid?}(0) \\ = & \neg \mathit{valid?}(\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j]) \\ = & \mathit{user\text{-}ppx\text{-}pt?}(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.47)},\end{array}$$

  $$\begin{array}{ll} & \mathit{px}(0) \neq \texttt{ZFP} \wedge \neg \mathit{prot?}(0) \\ = & \mathit{px}(\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j]) = \texttt{ZFP} \longleftrightarrow \mathit{prot?}(\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j]) \\ = & \mathit{zfp\text{-}eq\text{-}prot\text{-}pt?}(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.50)},\end{array}$$

  $$\begin{array}{ll} & \neg \mathit{valid?}(0) \\ = & \mathit{valid?}(\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j]) \longrightarrow \mathit{exec?}(\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j]) \\ = & \mathit{valid\text{-}is\text{-}exec\text{-}pt?}(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.51), and}\end{array}$$

  $$\begin{array}{ll} & \mathit{px}(0) \neq \texttt{ZFP} \\ = & \mathit{px}(\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j]) = \texttt{ZFP} \longrightarrow \mathit{valid?}(\mathit{init}\text{-}c_{\mathrm{PFH}}.\mathit{pt}[i][j]) \\ = & \mathit{zfp\text{-}is\text{-}valid\text{-}pt?}(\mathit{init}\text{-}c_{\mathrm{PFH}}) \quad \text{(Definition 4.52)}.\end{array}$$

■

◀ LEMMA 4.62
Validity is preserved under page-fault handling

The validity invariant is preserved under handling a page fault which occurred during execution of a user process *pid* at a virtual address *va*:

$$\begin{array}{l} \mathit{pfh}\surd(c_{\mathrm{PFH}}) \\ \wedge\ \mathit{pf}_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}, \mathit{intent}) \\ \wedge\ \neg \mathit{ptlexcp}_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}) \\ \wedge\ \mathit{user\text{-}pid?}(\mathit{pid}) \\ \longrightarrow \mathit{pfh}\surd(\mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va})). \end{array}$$

◀ PROOF

During handling the page fault elements of the active and free lists are permuted and only the last element of the active list

$$l = |\mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}).\mathit{active}| - 1$$

has new values of its *pid* and *vpx* fields:

$$\begin{array}{lcl} \mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}).\mathit{active}[l].\mathit{pid} & = & \mathit{pid}, \\ \mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}).\mathit{active}[l].\mathit{vpx} & = & \mathit{px}(\mathit{va}). \end{array}$$

Besides that, only the page table-table space component $c_{\mathrm{PFH}}.\mathit{pt}$ of the abstract PFH configuration is changed in a way that the entry for the pair $(\mathit{pid}, \mathit{px}(\mathit{va}))$

$$\mathit{pte}_{\mathrm{PFH}}(\mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}), \mathit{pid}, \mathit{px}(\mathit{va}))$$

becomes valid and executable, and its physical page index is set to the *ppx* field of the active list's last element. Moreover, in case the free list was empty the page table entry corresponding to the first element of the active list

$$\mathit{pte}_{\mathrm{PFH}}(\mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}), c_{\mathrm{PFH}}.\mathit{active}[0].\mathit{pid}, c_{\mathrm{PFH}}.\mathit{active}[0].\mathit{vpx})$$

becomes invalid.

We will consider the most involved case of the proof: invariant *dstnct-pid-vpx-active*? (Definition 4.43) which states that the active list contains only distinct pairs $(\mathit{pid}, \mathit{vpx})$. In order to prove this distinction we consider pairs (i, j) of page descriptor positions in the active list. A non-triviality

arises when one of the pair's element, say j, is the newly inserted one: $i < l$ and $j = l$. In case the free list was not empty the elements at position i remains the same, otherwise

$$\mathit{handle\text{-}pf}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}).\mathit{active}[i] = c_{\mathrm{PFH}}.\mathit{active}[i+1].$$

Let us denote this element as the i'-th one. So, we need to show

$$(c_{\mathrm{PFH}}.\mathit{active}[i'].\mathit{pid}, c_{\mathrm{PFH}}.\mathit{active}[i'].\mathit{vpx}) \neq (\mathit{pid}, \mathit{px}(\mathit{va})).$$

We prove the goal by contradiction. Assume, that $c_{\mathrm{PFH}}.\mathit{active}[i'].\mathit{pid} = \mathit{pid}$ and $c_{\mathrm{PFH}}.\mathit{active}[i'].\mathit{vpx} = \mathit{px}(\mathit{va})$. Using validity invariant *valid-active*? (Definition 4.39) we get

$$\mathit{valid?}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va})),$$

which is equivalent by Definition 4.16 to

$$\neg \mathit{iapf?}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}).$$

Using validity invariants *valid-pte-descr-active*? and *ppx-active-free-not-zfp*? (Definitions 4.46 and 4.42) we conclude

$$\mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va})) \neq \mathtt{ZFP},$$

which is equivalent by Definition 4.17 to

$$\neg \mathit{zppf?}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}, \mathit{intent}).$$

Combining these negated page fault signals we obtain

$$\neg \mathit{pf}_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{va}, \mathit{intent}),$$

■ which contradicts with the lemma's assumptions.

LEMMA 4.63 ▶ Validity is preserved under preventing page swap out

The validity invariant is preserved under preventing page swap out:

$$\begin{aligned} &\mathit{pfh}\surd(c_{\mathrm{PFH}}) \wedge c_{\mathrm{PFH}}.\mathit{free} = [\,] \\ \longrightarrow\ &\mathit{pfh}\surd(\mathit{prevent\text{-}swap\text{-}out}(c_{\mathrm{PFH}})). \end{aligned}$$

PROOF ▶ The algorithm for preventing page swap out only permutes elements of the active list. Since all properties concerning the active list talk only about ■ its elements and not their order, they could be easily shown.

CHAPTER 5

Implementation Correctness

In the previous chapter we have introduced abstract page-fault handler configurations and formally specified demand paging operations. The goal of this chapter is to formally verify that our demand paging implementation satisfies the desired specification. We fulfill our goal by conducting the corresponding proofs in the Hoare logics. For this we translate the C0 implementation of demand paging into Simpl and define a concrete shape of the Simpl state. Next we introduce an abstraction relation from the concrete Simpl state towards the abstract page-fault handler configuration: global and heap variables from the Simpl implementation are mapped to their meanings on the abstract side. This relation will play the essential role in specifying pre- and postcondition for the functions of our demand paging implementation in the Hoare logics. We elaborate on the implementation details of the initialization code, swap in and swap out routines, as well as the page-fault handler, and introduce pre- and postcondition for these functions. Having them, we prove that they are respected by our implementation.

5.1 Simpl Hoare Logic State Space

We will call the state space for code verification of the demand paging implementation the Simpl state space or the Hoare logic state space and denote it by C_{H}. The subscript "$_{\text{H}}$" stands for "Hoare". A single state or configuration is denoted by c_{H}. The Simpl state is a record whose individual elements are global and local variables from the demand paging implementation. All variables of structural types are flattened: for each individual field there is a a separate component in the state space. The state space contains a special variable $alloc :: ref\text{-}t^*$ which is a list of already allocated references. As pointed out before, the Simpl state space also aggregates the extended state of demand paging: $c_{\text{H}}.x :: C_{\text{X}}$. For each pointer variable or pointer structure field a heap function is inserted into the state space as stated in Table 5.1. State space components for heap functions are annotated with a subscript "$_{\text{heap}}$". Global variables from the implementation are listed in Table 5.2. Note that this table contains also variables like $c_{\text{H}}.sr_{\text{glob}}$ which are not used in the demand paging code but rather in some other CVM parts. We have them in the state space in order to be able to show by means of the Hoare logic verification environment that they are not modified by the demand paging code. Global variables have a subscript "$_{\text{glob}}$". Finally, Table 5.3 lists used local variables and function parameters. Note that in order to reduce the state space size we share local variables between functions. Local variables have suffixes "$_{\text{loc}}$".

Table 5.1: Simpl state components for heap functions

Name	Type	Meaning
pt_{heap}	$ref\text{-}t \mapsto \mathbb{N}^{**}$	page table array
pid_{heap}	$ref\text{-}t \mapsto \mathbb{N}$	process id field of page descriptors
vpx_{heap}	$ref\text{-}t \mapsto \mathbb{N}$	virtual page index field of page descriptors
ppx_{heap}	$ref\text{-}t \mapsto \mathbb{N}$	physical page index field of page descriptors
$next_{\text{heap}}$	$ref\text{-}t \mapsto ref\text{-}t$	'next' field in active and free lists
$prev_{\text{heap}}$	$ref\text{-}t \mapsto ref\text{-}t$	'previous' field in active and free lists

5.2 Abstraction Relation from Simpl

In this section we introduce an abstraction relation from the demand paging Simpl implementation states c_{H} towards abstract PFH configurations c_{PFH}. The relation ties together global and heap variables from the implementation and individual components of the abstract configuration. The relation is structured according to components in the abstract state space: below we introduce separate conjuncts for each component.

5.2.1 Doubly-Linked Lists

To specify and verify programs which use (doubly) linked lists we use the definitions of Schirmer who himself follows the approach of Mehta and Nipkow [MN03]. Generally, pointer structures in the heap are abstracted to a suitable HOL type. A list in the heap is abstracted to a HOL list of references. After this abstraction specification and verification takes place in the domain

Table 5.2: Simpl state components for global variables

Name	Type	Meaning
$pcb\text{-}ef_{\text{glob}}$	$\mathbb{Z}^{**}$	PCB exception frames
$pcb\text{-}ihd_{\text{glob}}$	$\mathbb{N}^{*}$	PCB user-defined signals
$pcb\text{-}bpto_{\text{glob}}$	$\mathbb{Z}^{*}$	PCB big-page table origins
$pcb\text{-}bptl_{\text{glob}}$	$\mathbb{Z}^{*}$	PCB big-page table lengths
$pcb\text{-}empty_{\text{glob}}$	$\mathbb{Z}^{*}$	PCB dummy space
$pages\text{-}used_{\text{glob}}$	$\mathbb{N}$	number of currently used virtual pages
$active_{\text{glob}}$	*ref-t*	(pointer to the head of the) active list
$free_{\text{glob}}$	*ref-t*	(pointer to the head of the) free list
$ppx2pd_{\text{glob}}$	*ref-t**	reverse lookup array for physical page indices
$pages\text{-}free_{\text{glob}}$	$\mathbb{N}$	number of currently free physical pages
pt_{glob}	*ref-t*	pointer to the page table array
$bpfree_{\text{glob}}$	$\mathbb{N}^{*}$	list for the stack of free big pages
$bpages\text{-}free_{\text{glob}}$	$\mathbb{N}$	number of free big pages
bpt_{glob}	$\mathbb{N}^{*}$	big-page table array
sr_{glob}	$\mathbb{N}$	devices interrupt mask
cup_{glob}	$\mathbb{N}$	current user process
$kheap_{\text{glob}}$	$\mathbb{Z}$	kernel heap size

of HOL lists. Predicate $list_{\text{H}}? :: \textit{ref-t} \times (\textit{ref-t} \mapsto \textit{ref-t}) \times \textit{ref-t}^{*} \mapsto \mathbb{B}$ defines this abstraction.

$$list_{\text{H}}?(p, next, l) = \begin{cases} p = \texttt{NULL} & \text{if } l = [\,] \\ p = a \wedge p \neq \texttt{NULL} \wedge list_{\text{H}}?(next(p), next, l') & \text{if } l = a \circ l' \end{cases}$$

◀ DEFINITION 5.1
List abstracted from Simpl

The list of references is obtained by means of the heap function *next* by starting with the reference p following the references in the list l up to `NULL`. A more involved kind of linked lists is a doubly linked list. The list of references l constituting it is traversed in two directions by means of heap functions *next* and *prev*. The first and last elements of a doubly linked lists are p and q. Predicate $dlist_{\text{H}}? :: \textit{ref-t} \times (\textit{ref-t} \mapsto \textit{ref-t}) \times (\textit{ref-t} \mapsto \textit{ref-t}) \times \textit{ref-t} \times \textit{ref-t}^{*} \mapsto \mathbb{B}$ defines an abstraction for doubly linked lists.

$$dlist_{\text{H}}?(p, next, prev, q, l) = list_{\text{H}}?(p, next, l) \wedge list_{\text{H}}?(q, prev, rev(l))$$

◀ DEFINITION 5.2
Dobly-linked list abstracted from Simpl

Thus, a doubly linked list is nothing but two singly linked listed traversed in opposite directions.

Based on this abstraction a library of functions on doubly-linked lists was specified and verified in Hoare logics against its C0 implementation (on the Simpl level) [Ngu05, Sta06]. We import this library for our proofs, since our demand paging implementation uses the functions for creating a list, as well as inserting and deleting elements from it.

5.2.2 Page and Big-Page Management

Active and free lists. The page management lists are implemented as doubly-linked lists of page descriptors. Such list uses the following variables: (i) a pointer $p :: \textit{ref-t}$ to the head of the list, (ii) heap functions $next_{\text{heap}}$ and $prev_{\text{heap}}$

Table 5.3: Simpl state components for local variables

Name	Type	Meaning
pid_{loc}	$\mathbb{N}$	process id parameter
$dummy_{\text{loc}}$	$\mathbb{Z}$	integer result of calls to subroutines
$dummy\text{-}bool_{\text{loc}}$	$\mathbb{B}$	boolean result of calls to subroutines
$mem\text{-}addr_{\text{loc}}$	$\mathbb{N}$	memory address parameter
$disk\text{-}addr_{\text{loc}}$	$\mathbb{N}$	disk address parameter
s_{loc}	*ref-t*	pointer to a page descriptor
n_{loc}	$\mathbb{N}$	loop counter
$addr_{\text{loc}}$	$\mathbb{N}$	virtual address parameter
$intent_{\text{loc}}$	$\mathbb{N}$	intention of a call to the handler
$count_{\text{loc}}$	$\mathbb{N}$	number of calls to the handler during which the given page must not be swapped out
pte_{loc}	$\mathbb{N}$	page table entry
vpx_{loc}	$\mathbb{N}$	virtual page index parameter
$result_{\text{loc}}$	$\mathbb{N}$	intermediate result
$vict_{\text{loc}}$	*ref-t*	(pointer to the) page descriptor for swap out
$active\text{-}tail_{\text{loc}}$	*ref-t*	(pointer to the) tail of the active list
ppx_{loc}	$\mathbb{N}$	physical page index parameter
bpx_{loc}	$\mathbb{N}$	virtual big-page index
bbx_{loc}	$\mathbb{N}$	big-page byte index
$pbpx_{\text{loc}}$	$\mathbb{N}$	physical big-page index
res_{loc}	$\mathbb{Z}$	result variable

— both of the type *ref-t* $\mapsto$ *ref-t* — used to obtain next elements in the list, (iii) heap functions pid_{heap}, vpx_{heap}, and ppx_{heap} — all of the type *ref-t* $\mapsto \mathbb{N}$ — to retrieve fields for a process identifier, virtual and physical page indicies, respectively, and (iv) a pointer q :: *ref-t* to the last element of the list.

On the abstract side a page management list is specified by a list *pds* :: *pd-t** of (abstract) page descriptors. It is easy to notice that the abstraction specifies only the content of a page management list and lacks information about the structure of the list, i.e., how list elements are placed in the memory. We fix this problem by introducing an additional parameter to our abstraction mapping. The structure of the list's elements in the memory is given by a list of references l. All these three concepts, namely, variables from the implementation, a list of (abstract) page descriptors, and a list of references in the memory, are tied together in the predicate below. It defines an abstraction for doubly linked lists of page descriptors. Its meaning is that (i) the variables from the implementation form a doubly linked list with the structures of references given by l, (ii) there are as many references in l as in *pds*, and (iii) each page descriptor field obtained on the implementation side corresponds to the one on the abstract side.

DEFINITION 5.3 ▶ List of page descriptors abstracted from Simpl

$$\begin{aligned} pds\text{-}abs_{\text{H}}?(p, next_{\text{heap}}, prev_{\text{heap}}, pid_{\text{heap}}, vpx_{\text{heap}}, ppx_{\text{heap}}, l, pds) = \\ \exists\, q : dlist_{\text{H}}?(p, next_{\text{heap}}, prev_{\text{heap}}, q, l) \wedge |l| = |pds| \\ \wedge \forall\, i < |l| : pid_{\text{heap}}(l[i]) = pds[i].pid \\ \wedge vpx_{\text{heap}}(l[i]) = pds[i].vpx \\ \wedge ppx_{\text{heap}}(l[i]) = pds[i].ppx \end{aligned}$$

Having this, it is easy to define parts of the abstraction relation for the active and free lists. For the active list the parameter $refs_{\text{active}}$ plays the role of the references list l whereas $c_{\text{PFH}}.active$ is the list of the corresponding abstract page descriptors.

$$\begin{aligned} active\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}, refs_{\text{active}}) = \\ pds\text{-}abs_{\text{H}}?(&c_{\text{H}}.active_{\text{glob}}, c_{\text{H}}.next_{\text{heap}}, c_{\text{H}}.prev_{\text{heap}}, \\ &c_{\text{H}}.pid_{\text{heap}}, c_{\text{H}}.vpx_{\text{heap}}, c_{\text{H}}.ppx_{\text{heap}}, \\ &refs_{\text{active}}, c_{\text{PFH}}.active) \end{aligned}$$

◀ DEFINITION 5.4
Active list abstracted from Simpl

For the free list l is substituted by $refs_{\text{free}}$ and $c_{\text{PFH}}.free$ is the list of the respective abstract page descriptors.

$$\begin{aligned} free\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}, refs_{\text{free}}) = \\ pds\text{-}abs_{\text{H}}?(&c_{\text{H}}.free_{\text{glob}}, c_{\text{H}}.next_{\text{heap}}, c_{\text{H}}.prev_{\text{heap}}, \\ &c_{\text{H}}.pid_{\text{heap}}, c_{\text{H}}.vpx_{\text{heap}}, c_{\text{H}}.ppx_{\text{heap}}, \\ &c_{\text{PFH}}.refs_{\text{free}}, c_{\text{PFH}}.free) \end{aligned}$$

◀ DEFINITION 5.5
Free list abstracted from Simpl

Stack of free big pages. On the implementation side the stack of free big pages is a fixed-sized array of `TOT_BIG_PGS` elements. The index of the stack's topmost element is stored in the variable $c_{\text{H}}.bpages\text{-}free_{\text{glob}}$. All the elements between $c_{\text{H}}.bpages\text{-}free_{\text{glob}}$ and `TOT_BIG_PGS` have unknown values. On the abstract side this data structure is modeled as a stack abstract data type: push and pop operations actually change the stack's length. Considering these facts we need to state in the abstraction relation for the stack of free big pages that the abstract stack is a prefix of the implementation stack and that the length of the latter is bounded by `TOT_BIG_PGS`.

$$\begin{aligned} bpfree\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) = \\ &|c_{\text{H}}.bpfree_{\text{glob}}| = \texttt{TOT\_BIG\_PGS} \\ \wedge\, \forall\, i < |c_{\text{PFH}}.bpfree| &: c_{\text{H}}.bpfree_{\text{glob}}[i] = c_{\text{PFH}}.bpfree[i] \end{aligned}$$

◀ DEFINITION 5.6
Stack of free big pages abstracted from Simpl

5.2.3 Page and Big-Page Tables

Page table space. We access page tables of user processes in the implementation through a data structure called the page table space. This data structure is a two-dimensional array located in the heap memory. The pointer to the beginning of the array is stored in the global variable $c_{\text{H}}.pt_{\text{glob}}$. Since the array of page tables is the first variable allocated on the heap the value of $c_{\text{H}}.pt_{\text{glob}}$ has always to be 1. The content of this array could be accessed in the Simpl implementation by means of the heap function $c_{\text{H}}.pt_{\text{heap}}$. The obtained content $c_{\text{H}}.pt_{\text{heap}}(c_{\text{H}}.pt_{\text{glob}})$ has to match the abstract page table space $c_{\text{PFH}}.pt$. This defines the abstraction relation for the page table space.

$$pt\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \quad = \quad (c_{\text{H}}.pt_{\text{glob}} = 1) \wedge (c_{\text{H}}.pt_{\text{heap}}(c_{\text{H}}.pt_{\text{glob}}) = c_{\text{PFH}}.pt)$$

◀ DEFINITION 5.7
Page table space abstracted from Simpl

Big-page table space. As for the big-page table space its abstraction relation is straightforward. The space is implemented as a global memory array $c_{\text{H}}.bpt_{\text{glob}}$. Its content has to be equal to the one from the abstraction, i.e., $c_{\text{PFH}}.bpt$.

$$bpt\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \quad = \quad c_{\text{H}}.bpt_{\text{glob}} = c_{\text{PFH}}.bpt$$

◀ DEFINITION 5.8
Big-page table space abstracted from Simpl

5.2.4 Process Control Blocks

All abstraction relation parts for PCB fields relevant for demand paging follow the same patters. A single process control block is implemented as a structure comprising, among others, the following fields: (i) the exception frame which is an array of integers; its elements correspond to the visible registers of the physical machine, (ii) the big-page table origin field, and (iii) the big-page table length field. These structures for individual PCBs are assembled into an array of `MAX_PID` blocks. We are interested only in user processes and hence consider only items with indices *pid* respecting $0 < pid < \texttt{MAX\_PID}$. Moreover, only page table origins and lengths as well as their big-page versions are of our interest. Page table origin and length are stored in the exception frame array at positions `PTO` and `PTL`, respectively, whereas the big-page version of these concepts are stored in the individual PCB structure fields. Altogether, four equations below define the abstraction relation for relevant PCB fields.

DEFINITION 5.9 ▶ Page table origins abstracted from Simpl

$$pto\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) = \forall\, 0 < pid < \texttt{MAX\_PID} : c_{\mathrm{H}}.pcb\text{-}ef_{\mathrm{glob}}[pid][\texttt{PTO}] = c_{\mathrm{PFH}}.pto[pid]$$

DEFINITION 5.10 ▶ Page table lengths abstracted from Simpl

$$ptl\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) = \forall\, 0 < pid < \texttt{MAX\_PID} : c_{\mathrm{H}}.pcb\text{-}ef_{\mathrm{glob}}[pid][\texttt{PTL}] = c_{\mathrm{PFH}}.ptl[pid]$$

DEFINITION 5.11 ▶ Big-page table origins abstracted from Simpl

$$bpto\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) = \forall\, 0 < pid < \texttt{MAX\_PID} : c_{\mathrm{H}}.pcb\text{-}bpto_{\mathrm{glob}} = c_{\mathrm{PFH}}.bpto[pid]$$

DEFINITION 5.12 ▶ Big-page table lengths abstracted from Simpl

$$bptl\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) = \forall\, 0 < pid < \texttt{MAX\_PID} : c_{\mathrm{H}}.pcb\text{-}bptl_{\mathrm{glob}} = c_{\mathrm{PFH}}.bptl[pid]$$

5.2.5 Miscellaneous

The implementation of demand paging also contains a number of global variables for which there is no analogous components in the abstract state However, these variables have precise meanings and their values could be expressed as formulas over the state space of the abstraction.

The number of free pages $c_{\mathrm{H}}.pages\text{-}free_{\mathrm{glob}}$ is mapped to the length of the free list.

DEFINITION 5.13 ▶ Free physical pages abstracted from Simpl

$$pages\text{-}free\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) \quad = \quad c_{\mathrm{H}}.pages\text{-}free_{\mathrm{glob}} = |c_{\mathrm{PFH}}.free|$$

The number of free big-pages $c_{\mathrm{H}}.bpages\text{-}free_{\mathrm{glob}}$ equals to the length of the stack of free big pages.

DEFINITION 5.14 ▶ Free big pages abstracted from Simpl

$$bpages\text{-}free\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) \quad = \quad c_{\mathrm{H}}.bpages\text{-}free_{\mathrm{glob}} = |c_{\mathrm{PFH}}.bpfree|$$

The total number of currently used virtual pages is stored in the implementation variable $c_{\mathrm{H}}.pages\text{-}used_{\mathrm{glob}}$. This value is specified by the sum of virtual

pages associated with each user.

$$\textit{pages-used-abs}_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) \quad = \quad c_{\mathrm{H}}.\textit{pages-used}_{\mathrm{glob}} = \sum_{i=1}^{\texttt{MAX\_PID}} (c_{\mathrm{PFH}}.\textit{ptl}[i]+1)$$

◀ DEFINITION 5.15
Used virtual pages abstracted from Simpl

The reverse lookup array used for for associating physical page indices to page descriptors is mapped to the abstract PFH state as follows. The length of the implementation array $c_{\mathrm{H}}.\textit{ppx2pd}_{\mathrm{glob}}$ which stores pointers to page descriptors equals to the number of physical pages. For a user physical page index *ppx* the element $c_{\mathrm{H}}.\textit{ppx2pd}_{\mathrm{glob}}[\textit{ppx}]$ is contained in the list of active $\textit{refs}_{\mathrm{active}}$ or free $\textit{refs}_{\mathrm{free}}$ references. Applying the heap function for physical page indices to this element we obtain the value *ppx*.

$$\begin{aligned}\textit{ppx2pd-abs}_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, \textit{refs}_{\mathrm{active}}, \textit{refs}_{\mathrm{free}}) = \\ |c_{\mathrm{H}}.\textit{ppx2pd}_{\mathrm{glob}}| = \texttt{TOT\_PHYS\_PGS} \\ \wedge\, \forall \textit{ppx} : \textit{user-ppx?}(\textit{ppx}) \longrightarrow c_{\mathrm{H}}.\textit{ppx2pd}_{\mathrm{glob}}[\textit{ppx}] \in \textit{refs}_{\mathrm{active}} \circ \textit{refs}_{\mathrm{free}} \\ \wedge\, c_{\mathrm{H}}.\textit{ppx}_{\mathrm{heap}}(c_{\mathrm{H}}.\textit{ppx2pd}_{\mathrm{glob}}[\textit{ppx}]) = \textit{ppx}\end{aligned}$$

◀ DEFINITION 5.16
Reverse lookup array abstracted from Simpl

5.2.6 Altogether

Now we can combine all abstraction relations for individual variables into an overall abstraction relation from Simpl implementation towards the abstract PFH configuration.

The relation $\textit{abs}_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, \textit{refs}_{\mathrm{active}}, \textit{refs}_{\mathrm{free}})$ is defined as a conjunction of definitions 5.4–5.16.

◀ DEFINITION 5.17
Abstraction relation from Simpl

For convenience we combine this abstraction relation with the validity invariant of abstract PFH configurations in a single predicate below. Also this predicate contains terms defining validity properties for the lists of active $\textit{refs}_{\mathrm{active}}$ and free $\textit{refs}_{\mathrm{free}}$ references. Since active and free lists are disjoint doubly-linked list we claim that $\textit{refs}_{\mathrm{active}}$ and $\textit{refs}_{\mathrm{free}}$ are also do not overlap. Further, we know that these lists are allocated in the heap memory directly after the page table space. As we have a single pointer to the page table space it consumes the very first address in the heap memory. Hence elements of the active and free lists are accessed via pointers whose values start at two. As there is `USER_PGS` elements altogether in both lists we devote `USER_PGS` addresses in the heap memory starting from two to these lists. Formally, this is stated in the following definition.

$$\begin{aligned}\textit{abs}_{\mathrm{H}}\surd(c_{\mathrm{H}}, c_{\mathrm{PFH}}, \textit{refs}_{\mathrm{active}}, \textit{refs}_{\mathrm{free}}) = \\ \textit{abs}_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, \textit{refs}_{\mathrm{active}}, \textit{refs}_{\mathrm{free}}) \\ \wedge\, \textit{pfh}\surd(c_{\mathrm{PFH}}) \\ \wedge\, \textit{refs}_{\mathrm{active}} \cap \textit{refs}_{\mathrm{free}} = \emptyset \\ \wedge\, \textit{refs}_{\mathrm{active}} \cup \textit{refs}_{\mathrm{free}} = \{x \mid 2 \leq x \leq \texttt{USER\_PGS} + 1\}\end{aligned}$$

◀ DEFINITION 5.18
Abstraction relation from Simpl combined with validity

5.3 Initialization Code

When the kernel starts for the first time after the reset signal it has to execute the initialization code of demand paging `pfh_init()` in order to set up

necessary data structures such that they conform to the initial abstract PFH configuration (cf. Section 4.2).

5.3.1 Implementation

The initialization code of demand paging is presented in Listing 5.1. The implementation starts with an invocation of function `zero_fill_page` which fills the page at address `ZFP` with zeros. Next, at line 7 we allocate the page table space in the heap memory. With the loop at lines 9–16 we initialize those fields of process control blocks which are relevant for demand paging: page table origins and lengths as well as their big versions. We distribute page table origins of processes across the page table space with equal segments between the origins of each two consecutive processes. The page table lengths are set to -1, which denotes that all user processes have no virtual memory yet. At lines 17–18 we create empty active and free lists. The loop at lines 21–27 is intended to fill the empty list with `USER_PGS` = `TOT_PHYS_PGS` − `KERNEL_PGS` elements and initialize with them the reverse lookup array `ppx2pd`. The last loop of the function (lines 30–33) initializes the stack of free big pages.

Listing 5.1: Initialization code of demand paging

```
 1 int pfh_init()
 2 {
 3   struct pd*          s;
 4   unsigned int        n;
 5   int                 dummy;
 6   dummy             = zero_fill_page(ZFP);
 7   pt                = new(pt_t);
 8   n                 = 1u;
 9   while (n < MAX_PID) {
10     pcb[n].ef[PTO] = int(PX(PT_START +
11                             (n-1u) * TOT_PGS_PT * PTES_PER_PG * 4u / MAX_PID));
12     pcb[n].ef[PTL] = -1;
13     pcb[n].bpto    = 0;
14     pcb[n].bptl    = -1;
15     n              = n + 1u;
16   }
17   active            = dList_New();
18   free              = dList_New();
19   pages_used        = 0u;
20   n                 = KERNEL_PGS;
21   while (n < TOT_PHYS_PGS) {
22     s               = new(struct pd);
23     ppx2pd[n]       = s;
24     s->ppx           = n;
25     free            = dList_InsertHead(free, s);
26     n               = n + 1u;
27   }
28   pages_free        = TOT_PHYS_PGS - KERNEL_PGS;
29   n                 = 0u;
30   while (n < TOT_BIG_PGS) {
31     bpfree[n]       = TOT_BIG_PGS - 1u-n;
32     n               = n + 1u;
33   }
34   bpages_free       = TOT_BIG_PGS;
35   return 0;
36 }
```

5.3.2 Specification

Precondition. The precondition $PRE^{\mathrm{H}}_{\mathrm{init}}?(c_{\mathrm{H}})$ to the initialization code of demand paging is a conjunction of the following facts:

- $c_{\mathrm{H}}.pcb\text{-}ef_{\mathrm{glob}} = rep(rep(0, \texttt{EF\_DIM}), \texttt{MAX\_PID})$, exception frames of all processes are initialized with zeros,
- $c_{\mathrm{H}}.bpt_{\mathrm{glob}} = rep(0, \texttt{TOT\_BIG\_PGS})$, the big-page table variable is initialized with zeros,
- $|c_{\mathrm{H}}.pcb\text{-}bpto_{\mathrm{glob}}| = \texttt{MAX\_PID}$, the big-page table origin variable is a list of length equal to the number of processes,
- $|c_{\mathrm{H}}.pcb\text{-}bptl_{\mathrm{glob}}| = \texttt{MAX\_PID}$, the big page table length variable is a list of length equal to the number of processes,
- $|c_{\mathrm{H}}.ppx2pd_{\mathrm{glob}}| = \texttt{TOT\_PHYS\_PGS}$, the reverse lookup array variable is a list of length equal to the total number of virtual pages,
- $|c_{\mathrm{H}}.bpfree_{\mathrm{glob}}| = \texttt{TOT\_BIG\_PGS}$, the stack of free big pages variable is a list of length equal to the total number of big pages, and
- $c_{\mathrm{H}}.alloc = [\,]$, the list of allocated references is empty.

Postcondition. The main idea of the postcondition for the demand paging initialization code is to express that data structures from the implementation respect the values defined by the initial abstract PFH configuration $init\text{-}c_{\mathrm{PFH}}$. However, some variables in the implementation have no equivalents on the abstract side. Therefore, we have to claim their initial values directly. One category of such variables are the exception frames of process control blocks. The initial abstract PFH configuration defines only the values of registers PTO and PTL. In order to describe the remaining fields — all initialized with zeros — we introduce the following predicate.

◀ DEFINITION 5.19
Initialized exception frame

$$init\text{-}pcb\text{-}ef?(c_{\mathrm{H}}) = \forall\, 0 < i < \texttt{MAX\_PID} : \\ \wedge \forall\, j < \texttt{EF\_DIM}, j \neq \texttt{PTO}, j \neq \texttt{PTL} : c_{\mathrm{H}}.pcb\text{-}ef_{\mathrm{glob}}[i][j] = 0$$

The postcondition $POST^{\mathrm{H}}_{\mathrm{init}}?(c_{\mathrm{H}})$ of the initialization code is a conjunction of the following terms.

- $abs_{\mathrm{H}}\surd(c'_{\mathrm{H}}, init\text{-}c_{\mathrm{PFH}}, [\,], [\texttt{USER\_PGS}+1, \ldots, 2])$, the implementation state c'_{H} after the execution of the initialization code satisfies the abstraction relation towards the initial abstract PFH configuration $init\text{-}c_{\mathrm{PFH}}$; the latter configuration is valid. Initially there are no references to elements of the the active list. All user pages — there are USER_PGS of them — are free and the free list is the second variable allocated on the heap (after the page table array). This is indicated by the argument $[\texttt{USER\_PGS}+1, \ldots, 2]$ of the valid abstraction relation. Note that we start counting references from 1.
- $init\text{-}pcb\text{-}ef?(c'_{\mathrm{H}})$, exception frames are appropriately initialized.
- $c'_{\mathrm{H}}.alloc = [\texttt{USER\_PGS} + 1, \ldots, 1]$, the Simpl allocation list $c'_{\mathrm{H}}.alloc$ keeps track of addresses consumed by the heap variables allocated in the initialization code: the first address belongs to the pointer to the page table space whereas the further USER_PGS addresses refer to the elements of active and free lists.

- $c'_{\mathrm{H}}.x = \mathit{zero\text{-}fill\text{-}page}(\mathtt{ZFP}, c_{\mathrm{H}}.x)$, in the extended state the page at the page address ZFP is filled with zeros.

- $c'_{\mathrm{H}}.\mathit{res}_{\mathrm{loc}} = 0$, the result value of the function execution is zero.

5.3.3 Correctness

The theorem below states the correctness of the initialization code at the level of Simpl: (i) the postcondition of the function is satisfied in the state after the function call provided its precondition has been satisfied in the state before the call, (ii) the function terminates, and (iii) only those global and heap variables are changed in the Simpl state that have been modified by the function. We express this statement using a Hoare triple for Simple (cf. Section 3.2).

THEOREM 5.20 ▶ Implementation correctness of the initialization code

$$\begin{aligned}\Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} & \mathit{PRE}^{\mathrm{H}}_{\mathrm{init}}?(c_{\mathrm{H}}) \\ & c_{\mathrm{H}}.\mathit{res}_{\mathrm{loc}} = \mathit{Call}\,\mathtt{pfh\_init}() \\ & \mathit{POST}^{\mathrm{H}}_{\mathrm{init}}?(c'_{\mathrm{H}}) \cap \Delta(c_{\mathrm{H}}, c'_{\mathrm{H}}) = \\ & \{\mathit{kheap}_{\mathrm{glob}}, \mathit{pt}_{\mathrm{glob}}, \mathit{pcb\text{-}ef}_{\mathrm{glob}}, \mathit{pcb\text{-}bpto}_{\mathrm{glob}}, \mathit{pcb\text{-}bptl}_{\mathrm{glob}}, \mathit{active}_{\mathrm{glob}}, \mathit{free}_{\mathrm{glob}}, \\ & \mathit{pages\text{-}used}_{\mathrm{glob}}, \mathit{pages\text{-}free}_{\mathrm{glob}}, \mathit{bpfree}_{\mathrm{glob}}, \mathit{bpages\text{-}free}_{\mathrm{glob}}, \mathit{ppx2pd}_{\mathrm{glob}}, \\ & \mathit{pt}_{\mathrm{heap}}, \mathit{pid}_{\mathrm{heap}}, \mathit{vpx}_{\mathrm{heap}}, \mathit{ppx}_{\mathrm{heap}}, \mathit{next}_{\mathrm{heap}}, \mathit{prev}_{\mathrm{heap}}, \mathit{alloc}, x\}\end{aligned}$$

PROOF ▶ We annotate loops of the function with invariants $\mathit{INV}^i_{\mathrm{init}}?$ and ranking functions $\mathit{RANK}^i_{\mathrm{init}}$ for $i \in \{1, 2, 3\}$ defined in Appendix B, run VCG, and obtain the HOL subgoals stated below to be proven. Functions f_x denote modifications done over implementation configuration c_{H} by code lines x in Listing 5.1.

Subgoal 1. Implication from the precondition to the first invariant:

$$\mathit{PRE}^{\mathrm{H}}_{\mathrm{init}}?(c_{\mathrm{H}}) \longrightarrow \mathit{INV}^1_{\mathrm{init}}?(f_{6\text{–}8}(c_{\mathrm{H}})).$$

Subgoal 2. Preservation of the first invariant:

$$\mathit{INV}^1_{\mathrm{init}}?(c_{\mathrm{H}}, c_{\mathrm{X}}) \longrightarrow \mathit{INV}^1_{\mathrm{init}}?(f_{10\text{–}15}(c_{\mathrm{H}})).$$

Subgoal 3. Implication from the first invariant to the second invariant:

$$\mathit{INV}^1_{\mathrm{init}}?(c_{\mathrm{H}}, c_{\mathrm{X}}) \longrightarrow \mathit{INV}^2_{\mathrm{init}}?(f_{17\text{–}20}(c_{\mathrm{H}})).$$

Subgoal 4. Preservation of the second invariant:

$$\mathit{INV}^2_{\mathrm{init}}?(c_{\mathrm{H}}, c_{\mathrm{X}}) \longrightarrow \mathit{INV}^2_{\mathrm{init}}?(f_{22\text{–}26}(c_{\mathrm{H}})).$$

Subgoal 5. Implication from the second invariant to the third invariant:

$$\mathit{INV}^2_{\mathrm{init}}?(c_{\mathrm{H}}, c_{\mathrm{X}}) \longrightarrow \mathit{INV}^3_{\mathrm{init}}?(f_{28\text{–}29}(c_{\mathrm{H}})).$$

Subgoal 6. Preservation of the third invariant:

$$\mathit{INV}^3_{\mathrm{init}}?(c_{\mathrm{H}}, c_{\mathrm{X}}) \longrightarrow \mathit{INV}^3_{\mathrm{init}}?(f_{31\text{–}32}(c_{\mathrm{H}})).$$

Subgoal 7. Implication from the third invariant to the postcondition:

$$\mathit{INV}^3_{\mathrm{init}}?(c_{\mathrm{H}}, c_{\mathrm{X}}) \longrightarrow \mathit{POST}^{\mathrm{H}}_{\mathrm{init}}?(f_{34\text{–}35}(c_{\mathrm{H}})).$$

■

5.4 Swap In

The swap in routine `pfh_swap_in` is an interface between the hard disk driver part for reading `read_from_disk` and the page-fault handler. While data is transferred between the physical memory and the hard disk by function `read_from_disk` the main purpose of the function `pfh_swap_in` is to compute memory and swap addresses and pass them to `read_from_disk`. The swap in routine implements the corresponding I/O operation defined in Section 4.4 (cf. Definition 4.27).

5.4.1 Implementation

The source code of function `pfh_swap_in` is presented in Listing 5.2. The function loads a page from the hard disk at an address specified by process identifier `pid` and virtual page index `vpx` to the physical memory at page address `ppx`. The implementation starts with a computation of (virtual) big-page index `bpx` and big byte index `bbx` from parameter `vpx`. By accessing the corresponding big-page table entry at line 11 we obtain physical big-page index `pbpx`. At line 12 we compute the disk sector address from `pbpx` and `bbx` performing the required conversions from pages to sectors as well as noting the boot region offset. At line 14 we pass this address and the memory address computed at line 13 to the hard disk driver routine `read_from_disk`.

Listing 5.2: Source code of the swap in

```
 1 int pfh_swap_in(unsigned int pid, unsigned int ppx, unsigned int vpx)
 2 {
 3   bool             dummy_bool;
 4   unsigned int     bpx;
 5   unsigned int     bbx;
 6   unsigned int     pbpx;
 7   unsigned int     disk_addr;
 8   unsigned int     mem_addr;
 9   bpx             = BPX(vpx);
10   bbx             = BBX(vpx);
11   pbpx            = BPTE(pid, bpx);
12   disk_addr       = (pbpx << 13u) + (bbx << 3u) + (BOOT_PGS << 3u);
13   mem_addr        = ppx << 12u;
14   dummy_bool      = read_from_disk(mem_addr, disk_addr);
15   return 0;
16 }
```

5.4.2 Specification

Precondition. Below we present the facts constituting the precondition $PRE^{\mathrm{H}}_{\mathrm{swap\_in}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}})$ to the swap in routine. First of all we impose a number of constraints on the function's parameters:

- $0 < c_{\mathrm{H}}.pid_{\mathrm{loc}} <$ `MAX_PID`, the process identifier parameter corresponds to some user process,
- `ZFP` $\leq c_{\mathrm{H}}.ppx_{\mathrm{loc}} <$ `TOT_PHYS_PGS` the physical page index addresses a page within the user region of the physical memory, and
- $c_{\mathrm{H}}.vpx_{\mathrm{loc}} \leq c_{\mathrm{PFH}}.ptl[c_{\mathrm{H}}.pid_{\mathrm{loc}}]$, the virtual page index parameter is bounded by the ammount of the process's virtual memory.

The last constraint implies that the user process identified by $c_{\mathrm{H}}.pid_{\mathrm{loc}}$ has some virtual memory: $c_{\mathrm{PFH}}.ptl[c_{\mathrm{H}}.pid_{\mathrm{loc}}] \geq 0$.

Further, it would be meaningful to have an abstraction relation from the Simpl implementation towards the abstract PFH configuration in the precondition. However, it turns out that we do not need a complete relation for verification of the function but rather its particular parts, namely:

- $bpt\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}})$, the abstraction relation for the big-page table space, and
- $bpto\text{-}abs_{\mathrm{H}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}})$, the abstraction relation for the big-page table origins.

Finally, the precondition requires an appropriate sub-typing of the extended state: $subtyping_{\mathrm{X}}?(c_{\mathrm{H}}.x)$.

Postcondition. The postcondition $POST^{\mathrm{H}}_{\mathrm{swap\_in}}?(c'_{\mathrm{H}}, c_{\mathrm{PFH}})$ of the swap in function comprises the following terms:

- $c_{\mathrm{H}}.x' = swap\text{-}in(c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.ppx_{\mathrm{loc}}, c_{\mathrm{H}}.vpx_{\mathrm{loc}}, c_{\mathrm{PFH}}, c_{\mathrm{H}}.x)$, the extended state is updated by the semantics of the swap in function (cf. Definition 4.27),
- $subtyping_{\mathrm{X}}?(c'_{\mathrm{H}}.x)$, the sub-typing of the extended state is preserved, and
- $c'_{\mathrm{H}}.res_{\mathrm{loc}} = 0$, the result value of the function execution is zero.

5.4.3 Correctness

The theorem below states the correctness of the swap in routine at the level of Simpl.

THEOREM 5.21 ▶ Implementation correctness of the swap in

$$\begin{aligned}\Gamma \vdash^{\mathrm{t}}_{\mathrm{H}} & PRE^{\mathrm{H}}_{\mathrm{swap\_in}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) \\ & c_{\mathrm{H}}.res_{\mathrm{loc}} = Call\ \texttt{swap\_in}(c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.ppx_{\mathrm{loc}}, c_{\mathrm{H}}.vpx_{\mathrm{loc}}) \\ & POST^{\mathrm{H}}_{\mathrm{swap\_in}}?(c'_{\mathrm{H}}, c_{\mathrm{PFH}}) \wedge \Delta(c_{\mathrm{H}}, c'_{\mathrm{H}}) = \{x\}\end{aligned}$$

5.5 Swap Out

The swap out routine `pfh_swap_out` is an interface between the hard disk driver part for writing `write_to_disk` and the page-fault handler. The swap out routine implements the corresponding I/O operation from Section 4.4 (cf. Definition 4.29).

5.5.1 Implementation

The source code of function `pfh_swap_out` is presented in Listing 5.3. It differs from `pfh_swap_in` only at the call at line 14: we invoke `write_to_disk`.

5.5.2 Specification

Precondition. The precondition $PRE^{\mathrm{H}}_{\mathrm{swap\_out}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}})$ to the swap out function is the same as in the swap in function.

Listing 5.3: Source code of the swap out

```
1 int pfh_swap_out(unsigned int pid, unsigned int ppx, unsigned int vpx )
2 {
3   bool             dummy_bool;
4   unsigned int     bpx;
5   unsigned int     bbx;
6   unsigned int     pbpx;
7   unsigned int     disk_addr;
8   unsigned int     mem_addr;
9   bpx             = BPX(vpx);
10  bbx             = BBX(vpx);
11  pbpx            = BPTE(pid, bpx);
12  disk_addr       = (pbpx << 13u) + (bbx << 3u) + (BOOT_PGS << 3u);
13  mem_addr        = ppx << 12u;
14  dummy_bool      = write_to_disk(mem_addr, disk_addr);
15  return 0;
16 }
```

Postcondition. The postcondition $POST^{\mathrm{H}}_{\mathrm{swap\_out}}?(c'_{\mathrm{H}}, c_{\mathrm{PFH}})$ to the swap out function differs from the one for the swap in function only in the term concerning the extended state update. In the swap out function we use the semantics of the swap out operation for this update (cf. Definition 4.29):

$$c'_{\mathrm{H}}.x = \mathit{swap\text{-}out}(c_{\mathrm{H}}.\mathit{pid}_{\mathrm{loc}}, c_{\mathrm{H}}.\mathit{ppx}_{\mathrm{loc}}, c_{\mathrm{H}}.\mathit{vpx}_{\mathrm{loc}}, c_{\mathrm{PFH}}, c_{\mathrm{H}}.x).$$

5.5.3 Correctness

The theorem below states the correctness of the swap out routine at the level of Simpl.

◀ THEOREM 5.22
Implementation correctness of the swap out

$$\begin{aligned}\Gamma \;\vDash^{\mathrm{t}}_{\mathrm{H}}\; & \mathit{PRE}^{\mathrm{H}}_{\mathrm{swap\_out}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}) \\ & c_{\mathrm{H}}.\mathit{res}_{\mathrm{loc}} = \mathit{Call}\ \texttt{swap\_out}(c_{\mathrm{H}}.\mathit{pid}_{\mathrm{loc}}, c_{\mathrm{H}}.\mathit{ppx}_{\mathrm{loc}}, c_{\mathrm{H}}.\mathit{vpx}_{\mathrm{loc}}) \\ & \mathit{POST}^{\mathrm{H}}_{\mathrm{swap\_out}}?(c'_{\mathrm{H}}, c_{\mathrm{PFH}}) \wedge \Delta(c_{\mathrm{H}}, c'_{\mathrm{H}}) = \{x\}\end{aligned}$$

5.6 Page-Fault Handler

Page faults are treated by a page-fault handler, a piece of software which translates addresses and loads missing pages from the hard disk into the physical memory. The page-fault handler function is called in two situations: when page-fault exceptions occur and when the kernel executes primitives that access user memory. In the second case the handler simulates address translation for the kernel, which runs untranslated, and makes sure that the corresponding memory page is swapped in.

5.6.1 Implementation

Recall that the function `pfh_touch_addr` serves as a single entry point for all cases where the kernel has or might have to swap in a page. The function considers virtual address `addr` of user process `pid`. We distinguish four different intentions `intent` of a handler's call:

- `READ`: the page in question should be readable afterwards,
- `WRITE`: the page in question can afterwards be read and written,

- `OVERWRITE`: the caller intends to overwrite the entire page after the call, and
- `SWAP_IN`: a page fault has occurred at the given address and the function will swap in the page for arbitrary accesses.

Listing 5.4: Source code of the page-fault handler

```
1 unsigned int pfh_touch_addr(unsigned int pid, unsigned int addr,
2                              unsigned int intent, unsigned int count)
3 {
4   int                   dummy;
5   unsigned int          pte;
6   unsigned int          vpx;
7   unsigned int          result;
8   struct pd*            vict;
9   struct pd*            active_tail;
10  result              = 0u;
11  vpx                 = PX(addr);
12  if (intent == SWAP_IN && vpx > unsigned(pcb[pid].ef[PTL])) {
13    result            = INVALID_ADDR;
14  } else {
15    pte               = PTE(pid, vpx);
16    if (intent != SWAP_IN && VALID(pte) && (intent == READ || !PROT(pte))) {
17      if (count > pages_free) {
18        count         = count - pages_free;
19        vict          = active;
20        while ((vict->vpx != vpx || vict->pid != pid) && count > 0u) {
21          count       = count - 1u;
22          vict        = vict->next;
23        }
24        active_tail   = vict;
25        while (count > 0u) {
26          count       = count - 1u;
27          active_tail = active_tail->next;
28        }
29        if (active_tail != vict) {
30          active      = dList_Delete(active, vict);
31          dummy       = dList_InsertAfter(active_tail, vict);
32        }
33      }
34      result          = (pte & PPX_MASK) + BX(addr);
35    } else {
36      if (free == NULL) {
37        vict          = active;
38        active        = dList_Delete(active, vict);
39        PTE(vict->pid, vict->vpx)
40                      = PTE(vict->pid, vict->vpx) & INVALID_MASK;
41        dummy         = pfh_swap_out(vict->pid, vict->ppx, vict->vpx);
42      } else {
43        vict          = free;
44        free          = dList_Delete(free, vict);
45        pages_free    = pages_free - 1u;
46      }
47      active          = dList_Append(active, vict);
48      vict->pid       = pid;
49      vict->vpx       = vpx;
50      if (intent != OVERWRITE) {
51        if (PX(PTE(pid, vpx)) == ZFP) {
52          dummy       = zero_fill_page(vict->ppx);
53        } else {
54          dummy       = pfh_swap_in(pid, vict->ppx, vpx);
55        }
56      }
57      pte             = (vict->ppx << 12u) | VALID_MASK | EXEC_MASK;
58      PTE(pid, vpx)   = pte;
59      result          = (pte & PPX_MASK) + BX(addr);
60    }
61  }
62  return result;
63 }
```

Sometimes, it is necessary to have multiple pages available in memory. However, if `pfh_touch_addr` just checks that the page is currently in the memory, a subsequent call to `pfh_touch_addr` might invalidate it. In order to overcome this problem, the parameter `count` is used. It should be set to the number of subsequent calls to `pfh_touch_addr` that the currently touched page should at least survive. Note that the function assumes that the argument `count` is smaller than the number of user-available physical pages. The function will either return the translated physical address of the given virtual address or the error `INVALID_ADDR` in case of a page table length exception.

The implementation starts by computing virtual page index `vpx` of the given address. At lines 12–13 we check for a page table length exception and exit the handler with return code `INVALID_ADDR` in case it takes place. It is up to the kernel to decide what should happen in this situation. Otherwise we compute page table entry `pte` for the given parameters. At lines 16–34 we consider the case when the requested page is already accessible and must be guaranteed to survive `count` further calls to `pfh_touch_addr`. With the loop at lines 20–23 we search for the descriptor of the requested page in the active list. If the page was found to early in the active list, we push it (as little as possible) backwards at lines 24–32. Line 34 assign the translated physical memory address to the result variable. Further, at lines 35–60 we treat the situation of a legal user page fault or kernel-requested touch of a non-accessible page. The free list is examined at line 36 in order to find out whether any unused page resides in the physical memory and could be given (lines 43–46) to the page faulting process. If not, a page from an active list is evicted (lines 37–41). The 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 and intention (lines 50–56). The page table entry of the evicted page is invalidated (line 39–40) while the valid and executable bits of the loaded page are set (line 57). Finally, at line 59 we compute the translated physical memory address to be returned.

5.6.2 Specification

Precondition. Besides such typical things as parameter sub-typing as well as an abstraction relation the precondition to the page-fault handler includes the following condition. The parameter *intent* of the page-fault handler denotes the intention of a handlers's invocation. That a micro kernel calls the page-fault handler with an intention `SWAP_IN` means that a real hardware page-fault takes place. Recall that the page fault predicate $pf_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, pid, va, intent)$ does not hold for the case of a page table length exception. The latter is signaled by a separate predicate $ptlexcp_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, pid, va)$. Hence, in case of an `SWAP_IN` intention either a page fault, or a PTL exception predicate must hold. Conversely, if the intention is different from `SWAP_IN` there is no hardware page faults on the underlying machine. As a consequence, no page table length exception occurs either. Formally, these facts are gathered in the following definition.

◀ DEFINITION 5.23
No PTL Exception unless swap in

$$pf\text{-}swap\text{-}in?(c_{\mathrm{PFH}}, pid, va, intent) = \begin{cases} ptlexcp_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, pid, va) \vee pf_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, pid, va, intent) & \text{if } intent = \texttt{SWAP\_IN} \\ \neg ptlexcp_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, pid, va) & \text{otherwise} \end{cases}.$$

Now we define the precondition $PRE^{\mathrm{H}}_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, \mathit{refs}_{\mathrm{active}}, \mathit{refs}_{\mathrm{free}})$ to the page-fault handler formally. The first group of terms constituting the precondition impose restrictions on parameter ranges:

- $\mathit{user\text{-}pid}?(c_{\mathrm{H}}.\mathit{pid}_{\mathrm{loc}})$, the process identifier parameter corresponds to some user process,
- $c_{\mathrm{H}}.\mathit{addr}_{\mathrm{loc}} < \texttt{TOT\_PGS} \cdot \texttt{PG\_SZ}$, the virtual address parameter does not exceed the size of a machine word, and
- $c_{\mathrm{H}}.\mathit{count}_{\mathrm{loc}} \in \{0, 1\}$, we are going to make at most one subsequent call to the page-fault handler.

Also we require that the user process $c_{\mathrm{H}}.\mathit{pid}_{\mathrm{loc}}$ has some virtual memory: $c_{\mathrm{PFH}}.\mathit{ptl}[c_{\mathrm{H}}.\mathit{pid}_{\mathrm{loc}}] \geq 0$. Further, the precondition contains the predicate $\mathit{pf\text{-}swap\text{-}in}?(c_{\mathrm{PFH}}, c_{\mathrm{H}}.\mathit{pid}_{\mathrm{loc}}, c_{\mathrm{H}}.\mathit{addr}_{\mathrm{loc}}, c_{\mathrm{H}}.\mathit{intent}_{\mathrm{loc}})$. As described above, this predicate makes a case distinction on the intention parameter and gives us information about page fault and PTL exception signals in each case.

Finally, we need to state the abstraction relation and the validity requirements:

- $\mathit{abs}_{\mathrm{H}}\surd(c_{\mathrm{H}}, c_{\mathrm{PFH}}, \mathit{refs}_{\mathrm{active}}, \mathit{refs}_{\mathrm{free}})$, the abstraction relation from Simpl towards the abstract PFH configuration holds, and the latter configuration is valid, and
- $\mathit{subtyping}_{\mathrm{X}}?(c_{\mathrm{H}}.x)$, the extended state respects the sub-typing.

Postcondition. Depending on the functionality of our page-fault handler we distinguish four cases of its invocation:

Case 1. User page fault with a PTL exception: a hardware page fault happens due to a page table length exception.

Case 2. Legal user page fault: a hardware page fault happens and it is not a PTL exception.

Case 3. Kernel requested touch of an accessible page: the kernel wants to ensure that a certain page will reside in the physical memory for a specified number of subsequent calls to the page-fault handler and this page is already in the physical memory.

Case 4. Kernel requested touch of an inaccessible page: the kernel wants to ensure that a certain page will reside in the physical memory for a specified number of subsequent calls to the page-fault handler and this page is not present in the physical memory.

A distinction between these four cases is performed formally inside the predicate $POST^{\mathrm{H}}_{\mathrm{ta}}?(c'_{\mathrm{H}}, c_{\mathrm{PFH}}, \mathit{refs}_{\mathrm{active}}, \mathit{refs}_{\mathrm{free}})$. Below we discuss how this predicate is defined in each case.

Case 1: User page fault with a PTL exception. The case takes place if the intention parameter is SWAP_IN and there is a PTL exception:

$$c_{\mathrm{H}}.intent_{\mathrm{loc}} = \texttt{SWAP\_IN} \wedge ptlexcp_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}}).$$

A page table length exceptions signals the kernel that some kind of invalid memory operation has been undertaken. The page-fault handler does nothing in this case and reports an error to the kernel. More precisely, the case postcondition contains the following terms:

- $abs_{\mathrm{H}}\surd(c'_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}})$, the abstract relation as well is validity of the abstract PFH configuration is preserved,
- $c'_{\mathrm{H}}.x = c_{\mathrm{H}}.x$, the extended state is unchanged, and
- $c'_{\mathrm{H}}.res_{\mathrm{loc}} = \texttt{INVALID\_ADDR}$, the result variable is assigned an invalid memory operation code.

Case 2: Legal user page fault. This situation happens if the intention parameter differs from SWAP_IN and the page fault predicate holds:

$$c_{\mathrm{H}}.intent_{\mathrm{loc}} \neq \texttt{SWAP\_IN} \wedge pf_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}}, c_{\mathrm{H}}.intent_{\mathrm{loc}}).$$

The page-fault handler executes the page-fault handler code in this case. The code brings the implementation data structures to the state consistent to the abstract PFH configuration after applying the page-fault handling algorithm. The extended state reflects necessary page transfers between the physical and the swap memories. Formally, let us introduce the following abbreviations:

- $c'_{\mathrm{PFH}} = handle\text{-}pf_{\mathrm{PFH}}(c_{\mathrm{PFH}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}})$ is the abstract PFH configuration after applying our page-fault handling algorithm,
- $refs'_{\mathrm{active}} = tl(refs_{\mathrm{active}}) \circ hd(refs_{\mathrm{active}})$ and $refs'_{\mathrm{free}} = refs_{\mathrm{free}}$ are lists of active and free references after executing the algorithm in case the list of free references was empty, i.e., $refs_{\mathrm{free}} = [\,]$, and
- $refs'_{\mathrm{active}} = refs_{\mathrm{active}} \circ hd(refs_{\mathrm{free}})$ and $refs'_{\mathrm{free}} = tl(refs_{\mathrm{free}})$ are the same lists in case the list of free references was not empty, i.e., $refs_{\mathrm{free}} \neq [\,]$.

Formally, the postcondition of the legal user page fault case is a conjunction of the following facts:

- $abs_{\mathrm{H}}\surd(c'_{\mathrm{H}}, c'_{\mathrm{PFH}}, refs'_{\mathrm{active}}, refs'_{\mathrm{free}})$, the abstraction relations holds between the Simpl implementation configuration and the updated abstract PFH configuration,
- $c'_{\mathrm{H}}.x = handle\text{-}pf_{\mathrm{X}}(c_{\mathrm{PFH}}, c_{\mathrm{H}}.x, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.vpx_{\mathrm{loc}}, c_{\mathrm{H}}.intent_{\mathrm{loc}})$, the configuration of the extended state after transferring pages between the physical and swap memory is specified by the page-fault handling algorithm,
- $subtyping_{\mathrm{X}}?(c'_{\mathrm{H}}.x)$, the updated extended state is sub typed, and
- $c'_{\mathrm{H}}.res_{\mathrm{loc}} = pma_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}})$, the result variable of the page-fault handler stores the translated physical memory address for parameters $c_{\mathrm{H}}.pid_{\mathrm{loc}}$ and $c_{\mathrm{H}}.addr_{\mathrm{loc}}$.

Case 3: Kernel requested touch of an inaccessible page. The case takes place if the intention parameter is SWAP_IN and there is no PTL exception:

$$c_{\mathrm{H}}.intent_{\mathrm{loc}} = \texttt{SWAP\_IN} \wedge \neg ptlexcp_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}}).$$

The postcondition in this case is the same as in case 2.

Case 4: Kernel requested touch of an accessible page. This situation happens if the intention parameter differs from SWAP_IN and there is no page fault:

$$c_{\mathrm{H}}.intent_{\mathrm{loc}} \neq \texttt{SWAP\_IN} \wedge \neg pf_{\mathrm{PFH}}?(c_{\mathrm{PFH}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}}, c_{\mathrm{H}}.intent_{\mathrm{loc}}).$$

The case corresponds to a situation when the kernel executes CVM primitives that access user memory. Recall that sometimes it is necessary to have two certain pages available in the physical memory. Examples include a situation when the kernel copies data between two user processes. However, if the page-fault handler just checks that the page is currently in the memory, a subsequent call to the handler might invalidate it. In order to overcome this problem, the parameter $c_{\mathrm{H}}.count_{\mathrm{loc}}$ is set to one denoting that the page will survive the second call the page-fault handler. In case the descriptor of the considered page is the head of the active list we need to push it one one position further inside the active list. Whether such situation takes place is determined by the following predicate:

DEFINITION 5.24 ► Push page further in active list

$$\begin{aligned} \textit{to-push}?(c_{\mathrm{PFH}}, count, pid, ad) = {} & count = 1 \\ & \wedge\ c_{\mathrm{PFH}}.free = [\,] \\ & \wedge\ hd(c_{\mathrm{PFH}}.active).vpx = px(ad)^{\cdot} \\ & \wedge\ hd(c_{\mathrm{PFH}}.active).pid = pid \end{aligned}$$

In case $\textit{to-push}?(c_{\mathrm{PFH}}, c_{\mathrm{H}}.count_{\mathrm{loc}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}})$ holds we update the active list — it first element is moved to the second position:

$$c'_{\mathrm{PFH}}.active = hd(tl(c_{\mathrm{PFH}}.active)) \circ hd(c_{\mathrm{PFH}}.active) \circ tl(tl(c_{\mathrm{PFH}}.active)).$$

The updated version of the list $refs'_{\mathrm{active}}$ of active references in this case is obtained in a similar manner.

Now, the postcondition in the case consists of the following terms:

- $abs_{\mathrm{H}}\surd(c'_{\mathrm{H}}, c'_{\mathrm{PFH}}, refs'_{\mathrm{active}}, refs_{\mathrm{free}})$, the abstraction relations holds between the Simpl implementation configuration and the updated abstract PFH configuration,
- $c'_{\mathrm{H}}.x = c_{\mathrm{H}}.x$, the extended state is unchanged, and
- $c'_{\mathrm{H}}.res_{\mathrm{loc}} = pma_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}})$, the result variable of the page-fault handler stores the translated physical memory address for parameters $c_{\mathrm{H}}.pid_{\mathrm{loc}}$ and $c_{\mathrm{H}}.addr_{\mathrm{loc}}$.

5.6.3 Correctness

The theorem below states the correctness of the page-fault handler at the level of Simpl.

◀ THEOREM 5.25

Implementation correctness of the page-fault handler

$$\Gamma \vDash^{\mathrm{t}}_{\mathrm{H}} PRE^{\mathrm{H}}_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}})$$
$$c_{\mathrm{H}}.res_{\mathrm{loc}} = Call\ \texttt{pfh\_touch\_addr}(c_{\mathrm{H}}.pid_{\mathrm{loc}}, c_{\mathrm{H}}.addr_{\mathrm{loc}}, c_{\mathrm{H}}.intent_{\mathrm{loc}}, c_{\mathrm{H}}.count_{\mathrm{loc}})$$
$$POST^{\mathrm{H}}_{\mathrm{ta}}?(c'_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}) \cap \Delta(c_{\mathrm{H}}, c'_{\mathrm{H}}) =$$
$$\{active_{\mathrm{glob}}, free_{\mathrm{glob}}, pages\text{-}free_{\mathrm{glob}}, pt_{\mathrm{heap}}, pid_{\mathrm{heap}}, vpx_{\mathrm{heap}}, next_{\mathrm{heap}}, prev_{\mathrm{heap}}, x\}$$

◀ PROOF

We annotate loops of the function with invariants $INV^i_{\mathrm{ta}}?$ and ranking functions $RANK^i_{\mathrm{ta}}$ for $i \in \{1, 2\}$ defined in Appendix B, run VCG, and obtain HOL subgoals stated below to be proven. Functions f_x denote modifications done over implementation configuration c_{H} by code lines x in Listing 5.4.

Subgoal 1. Implication from the precondition to the postcondition:

$$PRE^{\mathrm{H}}_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}) \longrightarrow POST^{\mathrm{H}}_{\mathrm{ta}}?(f_{10\text{–}13,36\text{–}62}(c_{\mathrm{H}}), c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}).$$

Subgoal 2. Implication from the precondition to the first invariant:

$$PRE^{\mathrm{H}}_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}) \longrightarrow INV^1_{\mathrm{ta}}?(f_{10\text{–}11,15\text{–}16}(c_{\mathrm{H}}), c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}).$$

Subgoal 3. Preservation of the first invariant:

$$INV^1_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}) \longrightarrow INV^1_{\mathrm{ta}}?(f_{21\text{–}22}(c_{\mathrm{H}}), c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}).$$

Subgoal 4. Implication from the first invariant to the second invariant:

$$INV^1_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}) \longrightarrow INV^2_{\mathrm{ta}}?(f_{24}(c_{\mathrm{H}}), c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}).$$

Subgoal 5. Preservation of the second invariant:

$$INV^2_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}) \longrightarrow INV^2_{\mathrm{ta}}?(f_{26\text{–}27}(c_{\mathrm{H}}), c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}).$$

Subgoal 6. Implication from the second invariant to the postcondition:

$$INV^2_{\mathrm{ta}}?(c_{\mathrm{H}}, c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}) \longrightarrow POST^{\mathrm{H}}_{\mathrm{ta}}?(f_{34,62}(c_{\mathrm{H}}), c_{\mathrm{PFH}}, refs_{\mathrm{active}}, refs_{\mathrm{free}}).$$

■

CHAPTER 6

Property Transfer

So far we have proven in the Hoare Logics that our Simpl implementation of demand paging respects its specification. Our ultimate goal is to infer correctness properties of the demand paging in the semantics of VAMP ISA. A big milestone on this way is to prove correctness of the implementation represented in the C0 small-step semantics. In this chapter we exploit the C0 semantics stack in order to transfer implementation correctness results from the level of Simpl down to the level of SS. We achieve this goal through an intermediate transfer of correctness results to the big-step semantics level. We define abstraction relations between BS and SS states on the one side, and abstract PFH configurations on the other. With the help of these relations we formulate specifications of the demand paging functions on BS and SS levels. Finally, using the C0 semantics stack we prove that corresponding representations of the implementation meet their specifications. Throughout the transfer, calls to hard-disk drivers whose bodies are implemented in assembly are modeled as XCalls. Thus, the target level of this chapter is the *extended* C0 small-step semantics, i.e., it contains trusted XCalls to the drivers. We will get rid of these XCalls by plugging in the correctness statements of the drivers in Chapter 7.

6.1 Abstraction Relation from BS

This section defines abstraction relation

$$abs_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, HT, locs_{\mathrm{active}}, locs_{\mathrm{free}})$$

which claims that big-step semantics configuration c_{BS} implements abstract PFH configuration c_{PFH}. As auxiliary parameters the relation takes heap typing $HT :: (\mathit{loc\text{-}t} \mapsto \mathbb{S}_{\perp})$ and lists of locations in memory $locs_{\mathrm{active}}, locs_{\mathrm{free}} :: \mathit{loc\text{-}t}^*$ which define the structure of the active and free lists. Technically, the relation follows the idea of the abstraction relation from Simpl (Section 5.2) and comprises predicates for individual components of the abstract FPH configuration. We define them below, but do not give much textual description as the predicates differ from those defined in Section 5.2 mainly in data representation. For each predicate we provide references to the corresponding abstraction relation's from Simpl components such that the reader can examine differences between definition for BS and for Simpl. Below we remind the reader of the key differences in the memory models of C0 big-step semantics and Simpl.

Differences in the Simpl and BS memory models

As described in Chapter 3 the Simpl level features a split heap: every component of a structure is stored in a separate heap. Moreover implicit typing of variables is used. The program state is a record where every program variable and every split heap are fields with their own HOL types. In Section 5.1 we define such a record for our implementation of demand paging. With this shape of a program state it is convenient to define the abstraction relation for the variables and data structures from our demand paging implementation in Simpl towards the PFH abstraction: both levels are operating on ordinary HOL variables (cf. Section 5.2).

In contrast to that, C0 big-step semantics (cf. Section 3.3) features a single monolithic heap with compound values and explicit typing. The program state is record with three separate components for global, local, and heap variables. These components are partial mappings from variable names (locations, in case of heap variables) to compound values. The values are modeled by inductive data type *val-t* with separate constructors for primitive, array, and structure values. The primitive values are modeled by inductive data type *prim-t* whose constructors correspond to primitive data types supported by C0: booleans, integers, unsigned numbers, characters, and references.

C0 big-step semantics shares the model of expressions with C0 small-step semantics: expressions are instances of inductive data type *expr-t* (cf. Section 2.3). In order to evaluate an expression $e :: \mathit{expr\text{-}t}$ with respect to a big-step semantics state c_{BS} we use the function $eval_{\mathrm{BS}}(c_{\mathrm{BS}}, e)$ [Sch06, Figure 7.1]. It returns the value $v :: \mathit{val\text{-}t}_{\perp}$ of the expression e. This expression evaluation function will be used in definitions of abstractions relations from BS for different variables from the demand paging implementation. The common idea behind these abstraction relations is to evaluate the respective variable from the big-step semantics state by means of the function $eval_{\mathrm{BS}}$ and to express the obtained value with the help of inductive constructors of data type *val-t* and the corresponding HOL values taken from the abstract PFH configuration.

6.1.1 Doubly-Linked Lists

The idea for abstracting list implementation in BS is the same as for Simpl but another syntax for variable access (pointers, structure fields, etc.) is used. Also it is crucial that on the big-step level a single heap is used which is not split for different structure fields. Predicate

$$\mathit{list}_{\mathrm{BS}}? :: C_{\mathrm{BS}} \times (\mathit{loc\text{-}t} \mapsto \mathbb{S}_{\perp}) \times \mathit{val\text{-}t} \times \mathbb{S} \times \mathbb{S} \times \mathit{loc\text{-}t}^* \mapsto \mathbb{B}$$

has the meaning of $\mathit{list}_{\mathrm{H}}?$ (Definition 5.1) in big-step semantics. The differences are in parameters of the predicate. It takes (i) complete BS state c_{BS} to evaluate expressions, (ii) heap typing HT, (iii) pointer p to the first element of the list, (iv) type name tn of list elements, (v) field name $next$ which corresponds to the list's "next"-field which yields successor elements, and (vi) list of locations l specifying the list structure.

The list predicate in the big-step semantics means that in the non-degenerate case p points to the first address in list l which is appropriately typed and we can obtain p's successor p' for which the BS list predicate holds with l's tail l':

◀ DEFINITION 6.1
List abstracted from BS

$$\mathit{list}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, p, tn, next, l) = \begin{cases} p = \mathit{Prim}(\mathit{Null}) & \text{if } l = [\,] \\ p = \mathit{Prim}(\mathit{Addr}(a)) \wedge HT(a) = \lfloor tn \rfloor & \\ \quad \wedge \exists\, p' : \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{StructAcc}(\mathit{Deref}(\mathit{Lit}(p)), next)) = \lfloor p' \rfloor & \\ \qquad \wedge\ \mathit{list}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, p', tn, next, l') & \text{if } l = a \circ l'. \end{cases}$$

A BS doubly-linked list is formalized by application of the BS list predicate in both directions. Pointer q to the list's last element and the "predecessor"-field name $prev$ are additionally involved:

◀ DEFINITION 6.2
Doubly-linked list abstracted from BS

$$\mathit{dlist}_{\mathrm{BS}}? :: C_{\mathrm{BS}} \times (\mathit{loc\text{-}t} \mapsto \mathbb{S}_{\perp}) \times \mathit{val\text{-}t} \times \mathit{val\text{-}t} \times \mathbb{S} \times \mathbb{S} \times \mathbb{S} \times \mathit{loc\text{-}t}^* \mapsto \mathbb{B},$$

$$\mathit{dlist}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, p, q, tn, next, prev, l) = \mathit{list}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, p, tn, next, l) \wedge \mathit{list}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, q, tn, prev, \mathit{rev}(l)).$$

Since in this work we deal only with page-management lists whose elements are page descriptors we formalize below a doubly-linked list of page descriptors by instantiating the type name as well as parameters $next$ and $prev$:

$$\mathit{dlist}^{\mathrm{pd}}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, p, q, l) = \mathit{dlist}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, p, q, \mathtt{pd}, \mathtt{next}, \mathtt{prev}, l).$$

6.1.2 Page and Big-Page Management

First of all we define an abstraction relation for lists of page descriptors. It states that list of abstract page descriptors $\mathit{pds} :: \mathit{pd\text{-}t}^*$ and list of locations $l :: \mathit{loc\text{-}t}^*$ specify a doubly linked list of page descriptors starting at the variable of name vn in big-step semantics configuration c_{BS} with respect to heap typing HT. Each implementation page descriptor fields `pid`, `vpx`, and `ppx` are specified by the respective components of the respective abstract page descriptor (cf. Definition 5.3).

DEFINITION 6.3 ▶ List of page descriptors abstracted from BS

$$\begin{aligned}&\mathit{pds\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, \mathit{HT}, \mathit{vn}, l, \mathit{pds}) = \\ &\quad |l| = |\mathit{pds}| \wedge (\exists\, v : c_{\mathrm{BS}}.\mathit{gvars}(\mathit{vn}) = \lfloor v \rfloor \wedge \exists\, q : \mathit{dlist}^{\mathrm{pd}}_{\mathrm{BS}}?(c_{\mathrm{BS}}, \mathit{HT}, v, q, l)) \\ &\quad \wedge \forall i < |l|, \mathit{eval}_{\mathrm{pd}} = \lambda\, \mathit{fn} : \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{StructAcc}(\mathit{Deref}(\\ &\qquad\qquad \mathit{Lit}(\mathit{Prim}(\mathit{Addr}(l[i])))))), \mathit{fn}) : \\ &\qquad \mathit{eval}_{\mathrm{pd}}(\texttt{pid}) = \lfloor \mathit{Prim}(\mathit{Unsgnd}(\mathit{pds}[i].\mathit{pid})) \rfloor \\ &\quad \wedge\, \mathit{eval}_{\mathrm{pd}}(\texttt{vpx}) = \lfloor \mathit{Prim}(\mathit{Unsgnd}(\mathit{pds}[i].\mathit{vpx})) \rfloor \\ &\quad \wedge\, \mathit{eval}_{\mathrm{pd}}(\texttt{ppx}) = \lfloor \mathit{Prim}(\mathit{Unsgnd}(\mathit{pds}[i].\mathit{ppx})) \rfloor.\end{aligned}$$

By instantiating the head variable name with `active` and lists of abstract page descriptors and locations with $c_{\mathrm{PFH}}.\mathit{active}$ and $\mathit{locs}_{\mathrm{active}}$, respectively, we obtain abstraction relation for the active list (cf. Definition 5.4).

DEFINITION 6.4 ▶ Active list abstracted from BS

$$\begin{aligned}&\mathit{active\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, \mathit{HT}, \mathit{locs}_{\mathrm{active}}) = \\ &\qquad\qquad \mathit{pds\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, \mathit{HT}, \texttt{active}, \mathit{locs}_{\mathrm{active}}, c_{\mathrm{PFH}}.\mathit{active}).\end{aligned}$$

Similarly, we define abstraction relation for the free list (cf. Definition 5.5).

DEFINITION 6.5 ▶ Free list abstracted from BS

$$\begin{aligned}&\mathit{free\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, \mathit{HT}, \mathit{locs}_{\mathrm{free}}) = \\ &\qquad\qquad \mathit{pds\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, \mathit{HT}, \texttt{free}, \mathit{locs}_{\mathrm{free}}, c_{\mathrm{PFH}}.\mathit{free}).\end{aligned}$$

As for the stack of free big pages, on the implementation side it is a fixed-size array of `TOT_BIG_PGS` elements. Its abstract counterpart $c_{\mathrm{PFH}}.\mathit{bpfree}$, however, has variable length and stores only indices of free big pages. We bridge this gap by claiming existence of an array's postfix vs in the BS implementation such that the prefixes of implementation and abstract arrays match (cf. Definition 5.6).

DEFINITION 6.6 ▶ Stack of free big pages abstracted from BS

$$\begin{aligned}&\mathit{bpfree\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}) = \\ &\qquad \exists\, \mathit{vs} : |c_{\mathrm{PFH}}.\mathit{bpfree}| + |\mathit{vs}| = \texttt{TOT\_BIG\_PGS} \\ &\qquad\quad \wedge\, \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{bpfree})) = \\ &\qquad\qquad \lfloor \mathit{Arr}(\mathit{map}(\lambda\, x : \mathit{Prim}(\mathit{Unsgnd}(x)), c_{\mathrm{PFH}}.\mathit{bpfree}) \circ \mathit{vs}) \rfloor\end{aligned}$$

6.1.3 Page and Big-Page Tables

The abstraction relation for the page table space states that BS implementation variable `pt` is a reference which points to the first heap address and by dereferencing it we obtain a two dimensional array on the heap with elements matching those of the abstract page table space $c_{\mathrm{PFH}}.\mathit{pt}$ (cf. Definition 5.7).

DEFINITION 6.7 ▶ Page table space abstracted from BS

$$\begin{aligned}&\mathit{pt\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}) = \\ &\qquad \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{pt})) = \lfloor \mathit{Prim}(\mathit{Addr}(1)) \rfloor \\ &\quad \wedge\, \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{Deref}(\mathit{Lit}(\mathit{Prim}(\mathit{Addr}(1))))) = \\ &\qquad \lfloor \mathit{Arr}(\mathit{map}(\lambda\, x : \mathit{Arr}(\mathit{map}(\lambda\, y : \mathit{Prim}(\mathit{Unsgnd}(y)), x)), c_{\mathrm{PFH}}.\mathit{pt})) \rfloor.\end{aligned}$$

The abstraction relation for the big-page table space claims that BS implementation variable `bpt` is an array which matches the abstract big-page table space $c_{\mathrm{PFH}}.\mathit{bpt}$ (cf. Definition 5.8).

DEFINITION 6.8 ▶ Big-page table space abstracted from BS

$$\begin{aligned}&\mathit{bpt\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}) = \\ &\mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{bpt})) = \lfloor \mathit{Arr}(\mathit{map}(\lambda\, x : \mathit{Prim}(\mathit{Unsgnd}(x)), c_{\mathrm{PFH}}.\mathit{bpt})) \rfloor.\end{aligned}$$

6.1.4 Process Control Blocks

When it comes to the abstraction relation for process control blocks the major differences between the Simpl and big-step semantics come to light. In the Simpl with its flattened array representation it was sufficient only to claim correspondence between the values of Simpl arrays representing the PCB fields and theirs abstract counterparts (cf. Definitions 5.9–5.11). The big-step semantics has a more accurate memory model. Therefore, in order to define the abstraction relations for the process control blocks we have to consider how the array of PCB data structures is declared (Listing 2.1) and model the latter in the BS memory. The idea behind the abstraction relation is depicted in Figure 6.1.

Recall that abstract PCBs fields like $c_{\text{PFH}}.pto$ are arrays of `MAX_PID` elements including the (unused) element at position zero reserved for the kernel. The abstraction relation for the page table origins states that by evaluating BS variable for the PCB array `pcb` we obtain an array of structures with the first element equal to some value pcb_{kernel}. Each remaining element $i \in [0..\texttt{MAX\_PID}-2]$ has the following fields: (i) exception frame array fields `ef` composed out of some prefix $ef_{\text{hd}}[i]$, the page table origin constructed from the abstract one $c_{\text{PFH}}.pto[i{+}1]$, and some postfix $ef_{\text{tl}}[i]$, (ii) user-defined interrupt handlers $ihd[i]$, (iii) big-page table origins and lengths $bpto[i]$ and $bptl[i]$, and (iv) empty space *empty*. Moreover the abstraction relation claims that lengths of ef_{hd} and ef_{tl} are $\texttt{MAX\_PID}-1$ as well as that lengths of their elements $ef_{\text{hd}}[i]$ $ef_{\text{tl}}[i]$ are `PTO` and $\texttt{EF\_DIM}-\texttt{PTL}$. The latter reflects position of the page table origin element in the exception frame array.

◀ DEFINITION 6.9
Page table origins abstracted from BS

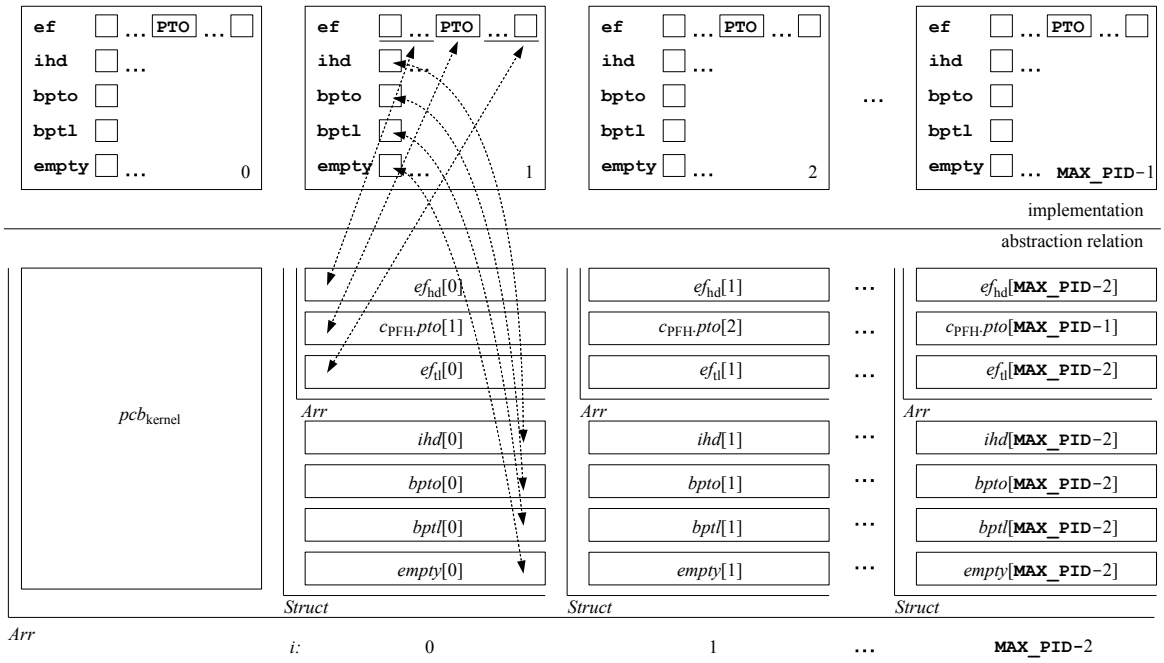
$$
\begin{aligned}
&\mathit{pto\text{-}abs}_{\text{BS}}?(c_{\text{BS}}, c_{\text{PFH}}) = \exists\, \mathit{pcb}_{\text{kernel}}, \mathit{ef}_{\text{hd}}, \mathit{ef}_{\text{tl}}, \mathit{ihd}, \mathit{bpto}, \mathit{bptl}, \mathit{empty} :\\
&\quad \mathit{eval}_{\text{BS}}(c_{\text{BS}}, \mathit{VarAcc}(\texttt{pcb})) =\\
&\quad \lfloor \mathit{Arr}(\mathit{pcb}_{\text{kernel}} \circ\\
&\qquad \mathit{map}(\lambda\, i : \mathit{Struct}([(\texttt{ef}, \mathit{Arr}(\mathit{ef}_{\text{hd}}[i] \circ\\
&\qquad\qquad [\mathit{Prim}(\mathit{Intg}(c_{\text{PFH}}.pto[i{+}1]))] \circ \mathit{ef}_{\text{tl}}[i])),\\
&\qquad\quad \mathit{ihd}[i], \mathit{bpto}[i], \mathit{bptl}[i], \mathit{empty}[i]]),\\
&\qquad [0..\texttt{MAX\_PID}-2]))\rfloor\\
&\quad \wedge\, |\mathit{ef}_{\text{hd}}| = |\mathit{ef}_{\text{tl}}| = \texttt{MAX\_PID}-1\\
&\quad \wedge\, (\forall\, i < \texttt{MAX\_PID}-1 : |\mathit{ef}_{\text{hd}}[i]| = \texttt{PTO})\\
&\quad \wedge\, (\forall\, i < \texttt{MAX\_PID}-1 : |\mathit{ef}_{\text{tl}}[i]| = \texttt{EF\_DIM}-\texttt{PTL})
\end{aligned}
$$

The abstraction relation for the page table lengths is defined in the same fashion. However, since `PTL` is the last element which stores some value in the exception frame array there is no postfix of the latter.

◀ DEFINITION 6.10
Page table lengths abstracted from BS

$$
\begin{aligned}
&\mathit{ptl\text{-}abs}_{\text{BS}}?(c_{\text{BS}}, c_{\text{PFH}}) = \exists\, \mathit{pcb}_{\text{kernel}}, \mathit{ef}_{\text{hd}}, \mathit{ihd}, \mathit{bpto}, \mathit{bptl}, \mathit{empty} :\\
&\qquad \mathit{eval}_{\text{BS}}(c_{\text{BS}}, \mathit{VarAcc}(\texttt{pcb})) =\\
&\qquad \lfloor \mathit{Arr}(\mathit{pcb}_{\text{kernel}} \circ\\
&\qquad\quad \mathit{map}(\lambda\, i : \mathit{Struct}([(\texttt{ef}, \mathit{Arr}(\mathit{ef}_{\text{hd}}[i] \circ\\
&\qquad\qquad [\mathit{Prim}(\mathit{Intg}(c_{\text{PFH}}.ptl[i{+}1]))])),\\
&\qquad\quad \mathit{ihd}[i], \mathit{bpto}[i], \mathit{bptl}[i], \mathit{empty}[i]]),\\
&\qquad\quad [0..\texttt{MAX\_PID}-2]))\rfloor\\
&\quad \wedge\, |\mathit{ef}_{\text{hd}}| = \texttt{MAX\_PID}-1 \wedge \forall\, i < \texttt{MAX\_PID}-1 : |\mathit{ef}_{\text{hd}}| = \texttt{PTL}
\end{aligned}
$$

Figure 6.1: Abstraction relation for page table origins



The abstraction relation for the big-page table origins is even simpler as it does not require us to consider the insides of the exception frame.

DEFINITION 6.11 ▶ Big-page table origins abstracted from BS

$$\begin{aligned}&\mathit{bpto\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}) = \exists\, \mathit{pcb}_{\mathrm{kernel}}, \mathit{ef}, \mathit{ihd}, \mathit{bptl}, \mathit{empty} :\\ &\quad \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{pcb})) =\\ &\quad \lfloor \mathit{Arr}(\mathit{pcb}_{\mathrm{kernel}} \circ \mathit{map}(\lambda\, i : \mathit{Struct}([\mathit{ef}[i], \mathit{ihd}[i],\\ &\qquad\qquad (\texttt{bpto}, \mathit{Prim}(\mathit{Intg}(c_{\mathrm{PFH}}.\mathit{bpto}[i{+}1]))),\\ &\qquad\qquad \mathit{bptl}[i], \mathit{empty}[i]]),\\ &\qquad [0..\texttt{MAX\_PID}-2]))\rfloor\end{aligned}$$

DEFINITION 6.12 ▶ Big-page table lengths abstracted from BS

The abstraction relation for the big-page table lengths is stated by predicate $\mathit{bptl\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}})$ which is defined similarly to the predicate above.

6.1.5 Miscellaneous

The abstraction relations for the number of free physical pages, free big pages, and used virtual pages are straightforward and follow Definitions 5.13–5.15.

DEFINITION 6.13 ▶ Free physical pages abstracted from BS

$$\begin{aligned}&\mathit{pages\text{-}free\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}) =\\ &\quad \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{pages\_free})) = \lfloor \mathit{Prim}(\mathit{Unsgnd}(|c_{\mathrm{PFH}}.\mathit{free}|))\rfloor.\end{aligned}$$

DEFINITION 6.14 ▶ Free big pages abstracted from BS

$$\begin{aligned}&\mathit{bpages\text{-}free\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}) =\\ &\quad \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{bpages\_free})) = \lfloor \mathit{Prim}(\mathit{Unsgnd}(|c_{\mathrm{PFH}}.\mathit{bpfree}|))\rfloor.\end{aligned}$$

$$\begin{aligned}&\mathit{pages\text{-}used\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}) = \\ &\qquad \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{pages\_used})) = \\ &\qquad\qquad \lfloor \mathit{Prim}(\mathit{Unsgnd}(\sum_{i=1}^{\texttt{MAX\_PID}-1} (c_{\mathrm{PFH}}.\mathit{ptl}[i] + 1)))\rfloor.\end{aligned}$$

◀ DEFINITION 6.15
Used virtual pages abstracted from BS

The reverse lookup array is supposed to map user physical page indices to corresponding page descriptors. Therefore, we state existence of an array of pointers to page descriptors $\mathit{pds}_{\mathrm{kernel}}$ for the kernel pages. Further, this array is succeeded by an array obtained from the values of user page descriptors $\mathit{pds}_{\mathrm{user}}$ to form the complete array stored at variable ppx2pd. The lengths of $\mathit{pds}_{\mathrm{kernel}}$ and $\mathit{pds}_{\mathrm{user}}$ are equal to the number of kernel and user pages, respectively. Next, the abstraction relation states that pointers to user page descriptors are drawn from the list of active or free locations and correctly typed with respect to heap typing HT. Finally, dereferencing of these pointers gives us the values of corresponding user physical page indices (cf. Definition 5.16).

$$\begin{aligned}&\mathit{ppx2pd\text{-}abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, HT, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) = \exists\, \mathit{pds}_{\mathrm{kernel}}, \mathit{pds}_{\mathrm{user}} : \\ &\quad \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{VarAcc}(\texttt{ppx2pd})) = \\ &\quad \lfloor \mathit{Arr}(\mathit{pds}_{\mathrm{kernel}} \circ \mathit{map}(\lambda\, x : \mathit{Prim}(\mathit{Addr}(x)), \mathit{pds}_{\mathrm{user}}))\rfloor \\ &\wedge\, |\mathit{pds}_{\mathrm{kernel}}| = \texttt{KERNEL\_PGS} \wedge |\mathit{pds}_{\mathrm{user}}| = \texttt{USER\_PGS} \\ &\wedge\, \forall\, i, j = i - \texttt{KERNEL\_PGS} : \mathit{user\text{-}ppx}?(i) \longrightarrow \\ &\qquad \mathit{pds}_{\mathrm{user}}[j] \in \mathit{locs}_{\mathrm{active}} \circ \mathit{locs}_{\mathrm{free}} \\ &\qquad \wedge\, \mathit{HT}(\mathit{pds}_{\mathrm{user}}[j]) = \lfloor \texttt{pd} \rfloor \\ &\qquad \wedge\, \mathit{eval}_{\mathrm{BS}}(c_{\mathrm{BS}}, \mathit{StructAcc}(\mathit{Deref}(\mathit{Lit}(\mathit{Prim}(\mathit{Addr}(\mathit{pds}_{\mathrm{user}}[j])))), \texttt{ppx})) = \\ &\qquad\quad \lfloor \mathit{Prim}(\mathit{Unsgnd}(i))\rfloor\end{aligned}$$

◀ DEFINITION 6.16
Reverse lookup array abstracted from BS

6.1.6 Altogether

Similarly to the abstraction relation from Simpl we combined the individual components defined above into the abstraction relation from BS.

The overall abstraction relation from the BS implementation towards abstract PFH configurations $\mathit{abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, HT, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}})$ is a conjunction of Definitions 6.4–6.16.

◀ DEFINITION 6.17
Abstraction relation from BS

Finally, we combine the overall abstraction relation together with validity properties of the abstract page-fault handler.

$$\begin{aligned}&\mathit{abs}_{\mathrm{BS}}\surd(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, HT, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) = \\ &\qquad \mathit{abs}_{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, HT, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) \\ &\qquad \wedge\, \mathit{pfh}\surd(c_{\mathrm{PFH}}) \\ &\qquad \wedge\, \mathit{locs}_{\mathrm{active}} \cap \mathit{locs}_{\mathrm{free}} = \emptyset \\ &\qquad \wedge\, \mathit{locs}_{\mathrm{active}} \cup \mathit{locs}_{\mathrm{free}} = \{x \mid 2 \leq x \leq \texttt{USER\_PGS} + 1\}.\end{aligned}$$

◀ DEFINITION 6.18
Abstraction relation from BS combined with validity

6.2 Property Transfer from Simpl to BS

In the previous section we presented the abstraction relation for the demand paging implementation in the big-step semantics towards an abstract PFH

configuration. In order to prove that the mentioned BS implementation is correct it remains to do the following.

First, we have to define a concrete version of the function which abstracts BS states towards Simpl states. As already mentioned such a function depends on the concrete Simpl state and cannot be defined generically to fit any state.

Next, we have to prove two lemmas that the valid abstraction relation from BS towards abstract PFH states implies the valid abstraction relation from Simpl towards abstract PFH states and vice verse. As we discussed in Section 3.4 we need the first implication to transfer preconditions respectively the second to transfer postconditions.

Further, we have to state specifications of the initialization code and the page-fault handler at the level of BS. Having this, we finally will be able to transfer implementation correctness of the initialization code and the page-fault handler down to the level of the big-step semantics.

6.2.1 Mapping Simpl States to BS States

In Section 3.4 we have mentioned the function $BS2H_{\text{state}}$ which abstracts big-step semantics states towards Simpl states. Since Simpl features a polymorphic state space and a shallow embedding this function could not be defined in Isabelle/HOL generically: the definition has to consider each single variable from the concrete state. With an example below we show how this function could be defined for some selected global and heap variables. We skip local variables as the part of the abstraction function for them is completely similar to those for global and heap variables.

For the global variables we choose the pointer to the free list and the big-page table as examples. As for the heap variables, the abstraction could only be defined for those BS heap values that are compatible with the types of values the demand paging implementation allocates on the heap (cf. Figure 7.10 of Schirmer's thesis [Sch06]). The demand paging implementation uses only values of two types in the heap memory: the page table-space and page descriptors. We define types ty_{pt} and ty_{pd} in the C0 big-step semantics for them below.

$$ty_{\text{pt}} = ArrT(\texttt{TOT\_PGS\_PT}, ArrT(\texttt{PTES\_PER\_PG}, UnsgndT))$$

$$ty_{\text{pd}} = StructT([(\texttt{pid}, UnsgndT), (\texttt{vpx}, UnsgndT), (\texttt{ppx}, UnsgndT), (\texttt{next}, PtrT(\texttt{pd\_t})), (\texttt{prev}, PtrT(\texttt{pd\_t}))])$$

For each heap location l in the BS state if the heap value $c_{\text{BS}}.heap(l)$ is compatible with the page table space type ty_{pt} then the heap function $c_{\text{H}}.pt_{\text{heap}}$ yields at the reference $loc2ref(l)$ a two-dimensional array constructed from the value $c_{\text{BS}}.heap(l)$. A similar constraint must hold for all heap locations compatible with the page descriptor type: the values obtained by the heap functions for page descriptor fields match the corresponding values from the BS state.

◀ DEFINITION 6.19
Abstraction of the state from BS towards Simpl

$$
\begin{aligned}
&BS2H_{\text{state}}(c_{\text{BS}}) = \\
&\{c_{\text{H}} \mid c_{\text{BS}}.gvars(\texttt{free}) = \lfloor Ref(c_{\text{H}}.free_{\text{glob}})\rfloor \\
&\quad \wedge\ c_{\text{BS}}.gvars(\texttt{bpt}) = \lfloor Arr(map(\lambda x : Prim(Unsgnd(x)), c_{\text{H}}.bpt_{\text{glob}}))\rfloor \\
&\quad \wedge\ \ldots \quad \text{(other global variables)} \\
&\quad \wedge\ (\forall l : \vdash_{\text{v}} \lVert\lfloor c_{\text{BS}}.heap(l)\rfloor\rVert :: ty_{\text{pt}} \longrightarrow \\
&\qquad c_{\text{BS}}.heap(l) = \lfloor Arr(map(\lambda x : Arr(map(\lambda y : Prim(Unsgnd(y)), x)), \\
&\qquad\qquad c_{\text{H}}.pt_{\text{heap}}(loc2ref(l))))\rfloor) \\
&\quad \wedge\ (\forall l : \vdash_{\text{v}} \lVert\lfloor c_{\text{BS}}.heap(l)\rfloor\rVert :: ty_{\text{pd}} \longrightarrow \\
&\qquad c_{\text{BS}}.heap(l) = \lfloor Struct([(\texttt{pid}, Prim(Unsgnd(c_{\text{H}}.pid_{\text{heap}}(loc2ref(l))))), \\
&\qquad\qquad (\texttt{vpx}, Prim(Unsgnd(c_{\text{H}}.vpx_{\text{heap}}(loc2ref(l))))), \\
&\qquad\qquad (\texttt{ppx}, Prim(Unsgnd(c_{\text{H}}.ppx_{\text{heap}}(loc2ref(l))))), \\
&\qquad\qquad (\texttt{next}, Ref(c_{\text{H}}.next_{\text{heap}}(loc2ref(l)))), \\
&\qquad\qquad (\texttt{prev}, Ref(c_{\text{H}}.prev_{\text{heap}}(loc2ref(l))))])\rfloor) \\
&\quad \wedge\ \ldots \quad \text{(local variables)}\}
\end{aligned}
$$

Now we have the abstraction relation between BS and Simpl states and can, finally, transfer correctness results of the demand paging implementation presented in Section 5 down to the level of the big-step semantics. For that we first show lemmas for the transfer of valid abstraction relations (Definitions 5.18, 6.18) from Simpl to BS and vice versa. These lemmas will be the main proof obligations for the transfer of the demand paging implementation correctness to the level of BS.

6.2.2 Transfer of Abstraction Relation

Transfer of the valid abstraction relation from Simpl to BS. The lemma below states that a valid abstraction relation from Simpl towards the abstract PFH state implies the valid abstraction relation from BS towards the abstract PFH state. Additional assumptions to the lemma are: (i) Simpl state c_{H} is drawn from the set of states obtained by abstraction of global and heap variables from BS state c_{BS}, and (ii) the conformance predicate hold for BS state c_{BS} with respect to type environment TE, heap typing HT, and global variables typing GT.

◀ LEMMA 6.20
Valid abstraction relation transfer from Simpl to BS

$$
\begin{aligned}
&abs_{\text{H}}\surd(c_{\text{H}}, c_{\text{PFH}}, refs_{\text{active}}, refs_{\text{free}}) \\
&\wedge\ c_{\text{H}} \in BS2H_{\text{state}}(c_{\text{BS}}) \\
&\wedge\ TE \vdash c_{\text{BS}} :: HT, GT, [\,] \\
&\longrightarrow abs_{\text{BS}}\surd(c_{\text{BS}}, c_{\text{PFH}}, HT, map(ref2loc, refs_{\text{active}}), map(ref2loc, refs_{\text{free}}))
\end{aligned}
$$

◀ PROOF

The idea of lemma's proof is as follows. From $abs_{\text{H}}\surd$ we know that Simpl variables from state c_{H} are correctly mapped to abstract PFH state c_{PFH}. Since c_{H} belongs to the set of states abstracted from big-step state c_{BS} we know that variables of c_{H} are also mapped to the variables of c_{BS}. Having these two facts, we conclude that the big-step state could be also mapped to the abstract PFH state. ■

Transfer of the valid abstraction relation from BS to Simpl. The lemma for the implication in the direction from BS to Simpl is quite similar to the previous one.

LEMMA 6.21 ►
Valid abstraction relation transfer from BS to Simpl

$$\begin{aligned} & abs_{\mathrm{BS}}\surd(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, HT, locs_{\mathrm{active}}, locs_{\mathrm{free}}) \\ \wedge\ & c_{\mathrm{H}} \in BS2H_{\mathrm{state}}(c_{\mathrm{BS}}) \\ \wedge\ & TE \vdash c_{\mathrm{BS}} :: HT, GT, [\,] \\ \longrightarrow\ & abs_{\mathrm{H}}\surd(c_{\mathrm{H}}, c_{\mathrm{PFH}}, map(loc2ref, locs_{\mathrm{active}}), map(loc2ref, locs_{\mathrm{free}})) \end{aligned}$$

6.2.3 Specification at the Level of BS

Pre- and postconditions to the functions of the demand paging implementation formulated at the level of the big-step semantics literally follow those at the level of Simpl. The only difference is data representation — we have already illustrated this while defining the abstraction relation from BS (Definition 6.17). Because of that we do not present formal definitions of pre- and postconditions but only declare respective predicates.

Initialization code. The precondition to the initialization code of demand paging at the level of BS is stated by predicate

$$PRE^{\mathrm{BS}}_{\mathrm{init}}?(c_{\mathrm{BS}}).$$

The postcondition at the level of BS is given by predicate

$$POST^{\mathrm{BS}}_{\mathrm{init}}?(c'_{\mathrm{BS}}, r),$$

where r is an artificial variable for storing the result of the initialization code. The definitions of both predicates follow Section 5.3.2. Most notably, the postcondition establishes the valid abstraction relation from BS (Definition 6.18) with the initial abstract PFH configuration:

$$abs_{\mathrm{BS}}\surd(c'_{\mathrm{BS}}, \mathit{init}\text{-}c_{\mathrm{PFH}}, \mathit{init}\text{-}HT, [\,], [\mathtt{USER\_PGS}{+}1..2]).$$

Above, *init-HT* is the initial value for the heap typing. The initialization code allocates the page table space at the heap location one and page descriptors of free and active lists at further `USER_PGS` locations.

DEFINITION 6.22 ►
Initial heap typing

$$\mathit{init}\text{-}HT(i) = \begin{cases} \lfloor \mathtt{pt\_t} \rfloor & \text{if } i = 1 \\ \lfloor \mathtt{pd\_t} \rfloor & \text{if } 1 < i \leq \mathtt{USER\_PGS}{+}1 \\ \bot & \text{otherwise} \end{cases}$$

We separate specifications for the extended state into independent predicates. By that we have the following advantage. Recall that the extended state is shared between the big-step and small-step semantics. Therefore, we can reuse the specifications for the extended state defined below while speaking about correctness of demand paging at the level of SS.

The precondition for the extended state is defined by the predicate

$$PRE^{\mathrm{X}}_{\mathrm{init}}?(c_{\mathrm{X}}).$$

It requires sub-typing of the extended state (Definition 4.8). The postcondition is stated by the predicate

$$POST^{\mathrm{X}}_{\mathrm{init}}?(c_{\mathrm{X}}, c'_{\mathrm{X}}).$$

It claims that the sub-typing is preserved and the page at page address ZFP is filled with zeros (Definition 4.30).

We define the set of names of global variables which are modified by the initialization code.

$$\mathit{modified}_{\mathrm{init}} = [\mathtt{kheap}, \mathtt{pt}, \mathtt{pcb}, \mathtt{active}, \mathtt{free}, \mathtt{bpfree} \\ \mathtt{pages\_used}, \mathtt{pages\_free}, \mathtt{bpages\_free}, \mathtt{ppx2pd}]$$

◀ DEFINITION 6.23
Global variables modified by the initialization code

Page-fault handler. The precondition of the page-fault handler function at the level of BS is specified by predicate

$$\mathit{PRE}_{\mathrm{ta}}^{\mathrm{BS}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \mathit{HT}, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}).$$

It explicitly takes values of the page-fault handler parameters as we evaluate them outside the precondition directly in the formulation of the correctness lemma at the level of BS (Lemma 6.30). The postcondition of the page-fault handler function is expressed by predicate

$$\mathit{POST}_{\mathrm{ta}}^{\mathrm{BS}}?(c'_{\mathrm{BS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \mathit{HT}, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}).$$

The formal definitions of both pre- and postconditions at the level of BS follow corresponding definitions at the level of Simpl (cf. Section 5.6.2). Most importantly, both pre- and postconditions contain the valid abstraction relation from BS towards abstract PFH state (Definition 6.18):

$$\mathit{abs}_{\mathrm{BS}}\surd(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, \mathit{HT}, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}).$$

In contrast to specification of the initialization code, the BS postcondition of the page=fault handler does not speak about the result variable r. That is because we introduce a separate term to describe the result in the formulation of the respective theorem. We will benefit from this style later on when dealing with the page-fault handler top-level correctness theorem. The result of the page fault handler on the level of BS is computed by the function

$$\mathit{pfh\text{-}ta\text{-}res}_{\mathrm{BS}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count})$$

which is defined following the cases described in Section 5.6.2: in case of a PTL exception the function returns an error constant INVALID_ADDR whereas in all other cases the function yields a physical memory address $\mathit{pma}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr})$ (Definition 4.14). The parameters *intent* and *count* are taken only to perform the case distinction. The additional parameter of the postcondition is an artificial variable r for the result.

The precondition for the extended state demanding only its sub-typing is:

$$\mathit{PRE}_{\mathrm{ta}}^{\mathrm{X}}?(c_{\mathrm{X}}).$$

The postcondition is stated by the predicate

$$\mathit{POST}_{\mathrm{ta}}^{\mathrm{X}}?(c_{\mathrm{X}}, c'_{\mathrm{X}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count})$$

which is defined by case splitting according to Section 5.6.2.

We define the set of names of global variables which are modified by the page-fault handler.

$$\mathit{modified}_{\mathrm{ta}} = [\mathtt{free}, \mathtt{active}, \mathtt{pages\_free}]$$

◀ DEFINITION 6.24
Global variables modified by page-fault handler

6.2.4 Correctness at the Level of BS

Demand paging program. Recall that Hoare triples on the big-step and small-step levels are defined with respect to some program Π and extended semantics environment *xsem*. In order to claim such Hoare triples for our demand paging functions we need to have concrete instances of the program and the extended semantics environment.

We denote the program of our demand paging implementation by Π_{PFH}. It contains the entries for (i) the initialization code `pfh_init`, (ii) the page-fault handler function `pfh_touch_addr`, (iii) the swapping routines `pfh_swap_out` and `pfh_swap_in`, and (iv) the functions of the doubly-linked list library used by the demand paging.

The extended semantics environment for the demand paging implementation is $xsem_{\text{PFH}}$. It contains entries for (i) the hard-disk drivers `read_from_disk` and `write_to_disk`, and (ii) the zero fill page function `zero_fill_page`. The formal definition of $xsem_{\text{PFH}}$ is introduced in Section 7.1 (Definition 7.6) where we discuss correctness of extended calls to those functions in details.

Having a concrete instance of the demand paging program Π_{PFH} we can define concrete version of different typings. Basically, the idea behind the following definition is to convert concepts like a type environment and global and local symbol tables of the program Π_{PFH} given as lists of pairs of names and types as defined in syntax of C0 and small-step semantics to partial mappings from names to types as used in big-step semantics.

DEFINITION 6.25 ▶ Typings w.r.t demand paging program

$$\begin{array}{rcl} TE_{\text{PFH}} & = & \mathit{map\text{-}of}(\Pi_{\text{PFH}}.te) \\ GT_{\text{PFH}} & = & \mathit{map\text{-}of}(\Pi_{\text{PFH}}.gst) \\ LT_{\text{init}} & = & \mathit{map\text{-}of}(f_{\text{pfh\_init}}.\mathit{params} \circ f_{\text{pfh\_init}}.\mathit{lvars}) \\ LT_{\text{ta}} & = & \mathit{map\text{-}of}(f_{\text{pfh\_touch\_addr}}.\mathit{params} \circ f_{\text{pfh\_touch\_addr}}.\mathit{lvars}) \end{array}$$

Above, $f_{\text{pfh\_init}}$ and $f_{\text{pfh\_touch\_addr}}$ are function definitions in the function table $\Pi_{\text{PFH}}.ft$.

Transfer of the initialization code correctness from Simpl to BS. The lemma stated belows claims correctness of the initialization code at the level of BS. It has only a single assumption. Some variable r to which we write the result of the call of the initialization code must reside in the context of caller's variables L. The lemma claims a valid big-step Hoare triple with respect to program Π_{PFH}, extended semantics environment $xsem_{\text{PFH}}$, and context of variables L. We need to express which variables are modified by the initialization code. For that we define below the predicate $\mathit{modifies}^{\text{BS}}_{\text{init}}?$ which compares two BS states c_{BS} and c'_{BS} with respect to a global typing GT and a context of caller's variables L and claims that (i) all variables outside the global typing GT remain unchanged, (ii) all variables from the global typing GT which do not belong to the list of modified global variables $\mathit{modified}_{\text{init}}$ remain unchanged, and (iii) all caller's variables from the context L except for the result variable r remain unchanged.

DEFINITION 6.26 ▶ Modification of BS variables by initialization code

$$\begin{array}{rl} \mathit{modifies}^{\text{BS}}_{\text{init}}?(c_{\text{BS}}, c'_{\text{BS}}, GT, L, r) = & \\ \quad (\forall v \notin \mathit{dom}(GT) : c'_{\text{BS}}.\mathit{gvars}(v) = c_{\text{BS}}.\mathit{gvars}(v)) & \\ \wedge \quad (\forall v \in \mathit{dom}(GT) : v \notin \mathit{modified}_{\text{init}} \longrightarrow c'_{\text{BS}}.\mathit{gvars}(v) = c_{\text{BS}}.\mathit{gvars}(v)) & \\ \wedge \quad (\forall v \in L : v \neq r \longrightarrow c'_{\text{BS}}.\mathit{lvars}(v) = c_{\text{BS}}.\mathit{lvars}(v)) & \end{array}$$

The precondition of the Hoare triple is the set of all big-step states c_{BS} which (i) conform with the type environment TE_{PFH}, global typing GT_{PFH}, and local typing LT_{init}, and (ii) respect the preconditions of the initialization code $PRE^{\text{BS}}_{\text{init}}?$ and $PRE^{\text{X}}_{\text{init}}?$.

The Hoare triple's postcondition is the set of all big-step states c'_{BS} for which (i) the postconditions $POST^{\text{BS}}_{\text{init}}?$ and $POST^{\text{X}}_{\text{init}}?$ are satisfied, (ii) the conformance predicate holds with respect to the initial heap typing *init-HT*, and (iii) modification of variables is described by the predicate $modifies^{\text{BS}}_{\text{init}}?$.

◀ LEMMA 6.27
Initialization code correctness at the level of BS

$$
\begin{aligned}
r \in L \;\longrightarrow\; \Pi_{\text{PFH}}, xsem_{\text{PFH}}, L \vDash^{\text{t}}_{\text{BS}} \\
&\{c_{\text{BS}} \mid c_{\text{BS}} = c^0_{\text{BS}} \\
&\quad \wedge\, TE_{\text{PFH}} \vdash c_{\text{BS}} :: [\,], GT_{\text{PFH}}, LT_{\text{init}} \\
&\quad \wedge\, PRE^{\text{BS}}_{\text{init}}?(c_{\text{BS}}) \wedge PRE^{\text{X}}_{\text{init}}?(c_{\text{BS}}.x)\} \\
&\quad\; SCall(r, \texttt{pfh\_init}, [\,]) \\
&\quad \{c'_{\text{BS}} \mid POST^{\text{BS}}_{\text{init}}?(c'_{\text{BS}}, r,) \wedge POST^{\text{X}}_{\text{init}}?(c^0_{\text{BS}}.xc'_{\text{BS}}.x) \\
&\quad\quad \wedge\, TE_{\text{PFH}} \vdash c'_{\text{BS}} :: \textit{init-HT}, GT_{\text{PFH}}, LT_{\text{init}} \\
&\quad\quad \wedge\, modifies^{\text{BS}}_{\text{init}}?(c^0_{\text{BS}}, c'_{\text{BS}}, GT_{\text{PFH}}, L, r)\}
\end{aligned}
$$

◀ PROOF

We apply the transfer theorem from Simpl to BS (Theorem 3.2) in which the Simpl Hoare triple is instantiated by the one proven correct for the initialization code (Theorem 5.20). The core goals of the proof comprise an implication from the big-step precondition of the function to the Simpl precondition, and an implication from the Simpl postcondition to the big-step postcondition. We use Lemmas 6.20 and 6.21, respectively, to conclude that goals. ■

Transfer of the page-fault handler correctness from Simpl to BS. Before stating the theorem of correctness results transfer from Simpl to BS for the page-fault handler let us formalize a few additional concepts.

As defined before, the BS specification of the page-fault handler takes values of the handler's call parameters. These values are obtained from the corresponding *expressions* passed as parameters. Since these expressions might be different depending on the place of the page-fault handler call in the CVM program, they appear as free variables in the formulation of the handler's correctness theorem at the level of BS and are subject to instantiation by the theorem's user. Our first goal, is to guarantee that the passed expressions are correctly evaluated. For this we define a predicate which ensures that expressions e_{pid}, e_{addr}, e_{intent}, and e_{count} are evaluated to the corresponding values *pid*, *addr*, *intent*, and *count* in a big-step semantics configuration c_{BS} with respect to a context of caller's variables L.

◀ DEFINITION 6.28
Evaluation of parameters to the page-fault handler at the level of BS

$$
\begin{aligned}
param^{\text{BS}}_{\text{ta}}?(c_{\text{BS}}, L, e_{pid}, e_{addr}, e_{intent}, e_{count}, pid, addr, intent, count) = \\
&eval_{\text{BS}}(L, c_{\text{BS}}, e_{pid}) = \lfloor Prim(\,Unsgnd(pid))\rfloor \\
\wedge\quad &eval_{\text{BS}}(L, c_{\text{BS}}, e_{addr}) = \lfloor Prim(\,Unsgnd(addr))\rfloor \\
\wedge\quad &eval_{\text{BS}}(L, c_{\text{BS}}, e_{intent}) = \lfloor Prim(\,Unsgnd(intent))\rfloor \\
\wedge\quad &eval_{\text{BS}}(L, c_{\text{BS}}, e_{count}) = \lfloor Prim(\,Unsgnd(count))\rfloor
\end{aligned}
$$

We define the predicate $modifies^{\text{BS}}_{\text{ta}}?$ which describes modification of variables done by the page-fault handler. The predicate compares two states c_{BS} and c'_{BS} with respect to a given global typing GT, heap typing HT, type environ-

ment TE, and context of caller's variables L. This predicate states that (i) all variables outside the global variables typing GT remain unchanged, (ii) all variables from the global typing GT which do not belong to the list $modified_{\text{ta}}$ remain unchanged, (iii) all variables from the context L except for r remain unchanged, (iv) domain of the big-step heap stays the same, (v) all heap locations of the types outside the type environment TE remain unchanged, and (vi) the heap might have changed only at the page table space (location one) and active and free lists (locations specified by $locs_{\text{active}}$ and $locs_{\text{free}}$).

DEFINITION 6.29 ▶
Modification of BS variables by the page-fault handler

$$
\begin{aligned}
&modifies_{\text{ta}}^{\text{BS}}?(c_{\text{BS}}, c'_{\text{BS}}, GT, HT, L, r, TE, locs_{\text{active}}, locs_{\text{free}}) = \\
&\quad (\forall v \notin dom(GT) : c'_{\text{BS}}.gvars(v) = c_{\text{BS}}.gvars(v)) \\
\wedge\ &(\forall v \in dom(GT) : v \notin modified_{\text{ta}} \longrightarrow c'_{\text{BS}}.gvars(v) = c_{\text{BS}}.gvars(v)) \\
\wedge\ &(\forall v \in L : v \neq r \longrightarrow c_{\text{BS}}.lvars(v) = c_{\text{BS}}.lvars(v)) \\
\wedge\ &dom(c'_{\text{BS}}.heap) = dom(c_{\text{BS}}.heap) \\
\wedge\ &(\forall l, tn : HT(l) = \lfloor tn \rfloor \wedge tn \notin dom(TE) \longrightarrow c'_{\text{BS}}.heap(l) = c_{\text{BS}}.heap(l)) \\
\wedge\ &(\forall tn, ty : TE(tn) = \lfloor ty \rfloor \longrightarrow \forall l : HT(l) = \lfloor tn \rfloor : \\
&\qquad (ty = ty_{\text{pt}} \wedge l \neq 1) \vee (ty = ty_{\text{pd}} \wedge l \notin locs_{\text{active}} \circ locs_{\text{free}}) \\
&\qquad \longrightarrow [\![c'_{\text{BS}}.heap(l)]\!] = [\![c_{\text{BS}}.heap(l)]\!])
\end{aligned}
$$

Analogously to Lemma 6.27 the lemma below assumes a result variable r to the be in the context of caller's variables L. The lemma concludes a valid BS Hoare triple with respect to a program Π_{PFH} and L. The precondition of the Hoare triple is the set of all big-step states c_{BS} which (i) conform with the type environment TE_{PFH}, local typing LT_{ta}, global typing GT_{PFH}, and some heap typing HT, (ii) correctly evaluates the page-fault handler parameters, and (iii) respect the preconditions of the handler $PRE_{\text{ta}}^{\text{BS}}?$ and $PRE_{\text{ta}}^{\text{X}}?$.

The Hoare triple's postcondition is the set of all big-step states c'_{BS} for which (i) the postconditions $POST_{\text{ta}}^{\text{BS}}?$ and $POST_{\text{ta}}^{\text{X}}?$ hold, (ii) the function's result is appropriately computed, (iii) the conformance judgment is preserved, and (iv) the variable modification is described by the predicate $modifies_{\text{ta}}^{\text{BS}}?$.

LEMMA 6.30 ▶
Page-fault handler correctness at the level of BS

$$
\begin{aligned}
&r \in L \longrightarrow \Pi_{\text{PFH}}, xsem_{\text{PFH}}, L \vDash_{\text{BS}}^{\text{t}} \\
&\{c_{\text{BS}} \mid c_{\text{BS}} = c_{\text{BS}}^0 \\
&\qquad \wedge\ TE_{\text{PFH}} \vdash c_{\text{BS}} :: LT_{\text{ta}}, GT_{\text{PFH}}, HT \\
&\qquad \wedge\ param_{\text{ta}}^{\text{BS}}?(c_{\text{BS}}, L, e_{pid}, e_{addr}, e_{intent}, e_{count}, pid, addr, intent, count) \\
&\qquad \wedge\ PRE_{\text{ta}}^{\text{BS}}?(c_{\text{BS}}, c_{\text{PFH}}, pid, addr, intent, count, HT, locs_{\text{active}}, locs_{\text{free}}) \\
&\qquad \wedge\ PRE_{\text{ta}}^{\text{X}}?(c_{\text{BS}}.x)\} \\
&\qquad\quad SCall(r, \texttt{pfh\_touch\_addr}, [e_{pid}, e_{addr}, e_{intent}, e_{count}]) \\
&\{c'_{\text{BS}} \mid POST_{\text{ta}}^{\text{BS}}?(c'_{\text{BS}}, c_{\text{PFH}}, pid, addr, intent, count, HT, locs_{\text{active}}, locs_{\text{free}}) \\
&\qquad \wedge\ POST_{\text{ta}}^{\text{X}}?(c_{\text{BS}}^0.x, c'_{\text{BS}}.x, c_{\text{PFH}}, pid, addr, intent, count) \\
&\qquad \wedge\ r = pfh\text{-}ta\text{-}res_{\text{SS}}(c_{\text{PFH}}, pid, addr, intent, count) \\
&\qquad \wedge\ TE_{\text{PFH}} \vdash c'_{\text{BS}} :: LT_{\text{ta}}, GT_{\text{PFH}}, HT \\
&\qquad \wedge\ modifies_{\text{ta}}^{\text{BS}}?(c_{\text{BS}}^0, c'_{\text{BS}}, GT_{\text{PFH}}, HT, L, r, TE_{\text{PFH}}, locs_{\text{active}}, locs_{\text{free}})\}
\end{aligned}
$$

PROOF ▶ We apply the transfer theorem from Simpl to BS (Theorem 3.2) in which the Simpl Hoare triple is instantiated by the one proven correct for the page-fault handler (Theorem 5.25). For the core proof goals — an implication from the big-step precondition of the function to the Simpl precondition, and an implication from the Simpl postcondition to the big-step postcondition — we

use Lemmas 6.20 and 6.21, respectively. ■

6.3 Program Context Extension

So far, we have correctness results of the demand paging implementation on the big-step level with respect to the program Π_{PFH} (Lemmas 6.27 and 6.30). The definition of Π_{PFH} contains only entries corresponding to the demand paging functions. However, we want to apply this correctness result in the context of a complete concrete kernel. That means that the correctness lemmas of the demand paging must also hold in the context of a program which contains entries for all functions of the CVM framework as well as the abstract kernel.

At first glance, it might seem that substituting Π_{PFH} by the program of the concrete kernel $\Pi_{\text{CK}}(\Pi_{\text{AK}})$ (Definition 2.17) may suffice. However, the concrete kernel program contains definitions of the hard-disk driver functions `read_from_disk` and `write_to_disk` as well as the zero fill page function `zero_fill_page`. These functions contain *inline assembly* code and are subroutines of the page-fault handler and its initialization code, respectively. As Section 3.5 points, the simulation theorems between Simpl and BS as well as BS and SS assume that the function subject to property transfer must not execute assembly statements.

In order to solve this problem we have to remove the hard-disk driver functions and the zero fill page routine from the definition of the concrete kernel program. Information about the removed functions will be stored instead in the extended semantics environment which is a parameter to correctness lemmas of demand paging on the small-step semantics level. Moreover, we have to replace all calls to the removed functions by the extended calls (XCalls). Next we define how this procedure works.

◀ DEFINITION 6.31
Replacement of calls by extended calls in statement tree

The function $\textit{repl-hdzfp}_{\text{stmt}}(s)$ analyzes the statement (tree) s and replaces all normal calls to the zero fill page and hard-disk driver functions by the corresponding extended calls. The function is defined by structural induction. For the zero fill page call $s = \textit{SCall}(vn, \texttt{zero\_fill\_page}, \textit{params})$ the result is defined as

$$\textit{XCall}(\texttt{zero\_fill\_page}, [\textit{VarAcc}(vn, \textit{IntegerT})], \textit{params}).$$

For the calls to the hard-disk driver functions $s = \textit{SCall}(vn, fn, \textit{params})$ with $fn \in \{\texttt{write\_to\_disk}, \texttt{read\_from\_disk}\}$ we define the results as

$$\textit{XCall}(fn, [\textit{VarAcc}(vn, \textit{BooleanT})], \textit{params}).$$

For the compositional, conditional, and loop statement the function goes one level deeper in the recursion tree. For all remaining statements the function is identity.

◀ DEFINITION 6.32
Replacement of calls by extended calls in function table

We define the function $\textit{repl-hdzfp}_{\text{ft}}(ft)$ which transforms the function table ft to xpt in two steps.

- It removes the definitions of hard-disk driver and zero fill page functions:

$$ft' = \textit{filter}(\lambda p : \textit{fst}(p) \notin \{\texttt{write\_to\_disk}, \texttt{read\_from\_disk}, \texttt{zero\_fill\_page}\}, ft).$$

- It replaces all calls to these functions by XCalls:

$$xpt = map(\lambda p : (fst(p), f'), ft')$$

where $f'.body = repl\text{-}hdzfp_{\text{stmt}}(snd(p).body)$.

Having this, we can define the *extended* program $\Pi_{\text{ext}}(\Pi_{\text{AK}})$, the one we will use instead of Π_{PFH}. The function table of this programs is defined by applying Definition 6.32 to the function table of the concrete kernel $ft_{\text{CK}}(\Pi_{\text{AK}})$. The type environment and the global symbol table are those of the concrete kernel: $te_{\text{CK}}(\Pi_{\text{AK}})$ and $gst_{\text{CK}}(\Pi_{\text{AK}})$ (cf. Definition 2.17).

DEFINITION 6.33 ► Extended program

$$\begin{array}{rcl} ft_{\text{ext}}(\Pi_{\text{AK}}) & = & repl\text{-}hdzfp_{\text{ft}}(ft_{\text{CK}}(\Pi_{\text{AK}})) \\ \Pi_{\text{ext}}(\Pi_{\text{AK}}).ft & = & ft_{\text{ext}}(\Pi_{\text{AK}}) \\ \Pi_{\text{ext}}(\Pi_{\text{AK}}).te & = & te_{\text{CK}}(\Pi_{\text{AK}}) \\ \Pi_{\text{ext}}(\Pi_{\text{AK}}).gst & = & gst_{\text{CK}}(\Pi_{\text{AK}}) \end{array}$$

Additionally, we define typings constructed from the extended program.

DEFINITION 6.34 ► Typings w.r.t extended program

$$\begin{array}{rcl} TE_{\text{ext}}(\Pi_{\text{AK}}) & = & map\text{-}of(te_{\text{CK}}(\Pi_{\text{AK}})) \\ GT_{\text{ext}}(\Pi_{\text{AK}}) & = & map\text{-}of(gst_{\text{CK}}(\Pi_{\text{AK}})) \end{array}$$

Note that since program extension does not affect local symbol tables we have no need to define local typings with respect to the extended program, but rather can keep using LT_{init} and LT_{ta}.

So far we have defined the extended program context for our demand paging implementation. Our next goal is to state and prove its correctness lemmas with respect to this extended program context. These proofs will use two following lemmas which have been proven by Schirmer.

The first lemma provides means to substitute a program Π used in a big-step step semantics Hoare triple by Π' provided that Π is a part of Π'.

LEMMA 6.35 ► Program context extension

Assume that (i) a big-step semantics Hoare triple $P\ s\ Q$ holds with respect to a program Π and an extended semantics environment $xsem$, (ii) for all function names fn that have an entry in the function table $\Pi.ft$ there is an entry in the function table $\Pi'.ft$, and (iii) for all type names ty that have an entry in the type environment $\Pi.te$ there is an entry in the type environment $\Pi'.te$, then the Hoare triple $P\ s\ Q$ holds with respect to Π':

$$\begin{array}{l} \Pi, xsem, L \vDash^{\text{t}}_{\text{BS}} P\ s\ Q \\ \wedge\ (\forall fn : map\text{-}of(\Pi.ft)(fn) = \lfloor x \rfloor \longrightarrow map\text{-}of(\Pi'.ft)(fn) = \lfloor x \rfloor) \\ \wedge\ (\forall ty : map\text{-}of(\Pi.te)(ty) = \lfloor x \rfloor \longrightarrow map\text{-}of(\Pi'.te)(ty) = \lfloor x \rfloor) \\ \longrightarrow \Pi', xsem, L \vDash^{\text{t}}_{\text{BS}} P\ s\ Q. \end{array}$$

The lemma is proven structural induction on the statement s.

The second lemma allows to substitute pre- and postconditions P and Q of a big-step step semantics Hoare triple by different pre- and postconditions P' and Q' provided that $P' \subseteq P$ and $Q \subseteq Q'$. The lemma is a nothing but a consequence rule for Hoare triples at the level of big-step semantics:

LEMMA 6.36 ► Consequence rule for BS Hoare triples

$$P' \subseteq P\ \wedge\ Q \subseteq Q'\ \wedge\ \Pi, xsem, L \vDash^{\text{t}}_{\text{BS}} P\ s\ Q \quad \longrightarrow \quad \Pi, xsem, L \vDash^{\text{t}}_{\text{BS}} P'\ s\ Q'.$$

Below we reformulate correctness lemmas of demand paging at the level of BS (Lemmas 6.27 and 6.30) with respect to the extended program.

$$\begin{array}{l} \mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \wedge r \in L \;\longrightarrow\; \Pi_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{xsem}_{\mathrm{PFH}}, L \vDash^{\mathrm{t}}_{\mathrm{BS}} \\ \quad \{c_{\mathrm{BS}} \mid c_{\mathrm{BS}} = c^0_{\mathrm{BS}} \\ \qquad \wedge\; \mathit{TE}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}) \vdash c_{\mathrm{BS}} :: [\,], \mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{LT}_{\mathrm{init}} \\ \qquad \wedge\; \mathit{PRE}^{\mathrm{BS}}_{\mathrm{init}}?(c_{\mathrm{BS}}) \wedge \mathit{PRE}^{\mathrm{X}}_{\mathrm{init}}?(c_{\mathrm{BS}}.x)\} \\ \\ \qquad \mathit{SCall}(r, \texttt{pfh\_init}, [\,]) \\ \\ \quad \{c'_{\mathrm{BS}} \mid \mathit{POST}^{\mathrm{BS}}_{\mathrm{init}}?(c'_{\mathrm{BS}}, r) \wedge \mathit{POST}^{\mathrm{X}}_{\mathrm{init}}?(c^0_{\mathrm{BS}}.x, c'_{\mathrm{BS}}.x) \\ \qquad \wedge\; \mathit{TE}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}) \vdash c'_{\mathrm{BS}} :: \mathit{init\text{-}HT}, \mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{LT}_{\mathrm{init}} \\ \qquad \wedge\; \mathit{modifies}^{\mathrm{BS}}_{\mathrm{init}}?(c^0_{\mathrm{BS}}, c'_{\mathrm{BS}}, \mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), L, r)\} \end{array}$$

◀ LEMMA 6.37
Initialization code correctness at the level of BS w.r.t. the extended program

◀ PROOF

First, we apply Lemma 6.35 where we instantiate the "small" program with Π_{PFH} and the "big" one with $\Pi_{\mathrm{ext}}(\Pi_{\mathrm{AK}})$. The first assumption of Lemma 6.35 is discharged by Lemma 6.27, the second and the third are proven by inspecting the definitions of small and big programs.

Second, we apply Lemma 6.36 in which P and Q are instantiated with pre- and postconditions of the Hoare triple from Lemma 6.27, whereas P' and Q' by those from the present lemma. This step leaves us two subgoals to prove: $P' \subseteq P$ and $Q \subseteq Q'$. Having the described instantiations, the difference between P and P' respectively Q and Q' is that the primed versions of the predicates are defined for the extended type environment $\mathit{TE}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})$ and global variables typing $\mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})$.

The subgoal $P' \subseteq P$ boils down to a weakening of a conforming BS state condition (cf. Definition 3.1). The main proof step is to show an implication $\mathit{dom}(\mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})) \subseteq \mathit{dom}(c_{\mathrm{BS}}.\mathit{gvars}) \longrightarrow \mathit{dom}(\mathit{GT}_{\mathrm{PFH}})) \subseteq \mathit{dom}(c_{\mathrm{BS}}.\mathit{gvars})$. It follows from the fact that $\mathit{GT}_{\mathrm{PFH}}$ is included in $\mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})$.

The subgoal $Q' \subseteq Q$ boils down to a strengthening of a conforming BS state condition (cf. Definition 3.1). The main proof step is to show an implication $\mathit{dom}(\mathit{GT}_{\mathrm{PFH}})) \subseteq \mathit{dom}(c_{\mathrm{BS}}.\mathit{gvars}) \longrightarrow \mathit{dom}(\mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})) \subseteq \mathit{dom}(c_{\mathrm{BS}}.\mathit{gvars})$. Here we need to show that the "difference" between $\mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})$ and $\mathit{GT}_{\mathrm{PFH}}$ has a counterpart in $c_{\mathrm{BS}}.\mathit{gvars}$. It follows from the condition $\forall v \notin \mathit{dom}(\mathit{GT}_{\mathrm{PFH}}) : c'_{\mathrm{BS}}.\mathit{gvars}(v) = c_{\mathrm{BS}}.\mathit{gvars}(v)$ of Definition 6.26. ■

$$\begin{array}{l} \mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \wedge r \in L \;\longrightarrow\; \Pi_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{xsem}_{\mathrm{PFH}}, L \vDash^{\mathrm{t}}_{\mathrm{BS}} \\ \{c_{\mathrm{BS}} \mid c_{\mathrm{BS}} = c^0_{\mathrm{BS}} \\ \quad \wedge\; \mathit{TE}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}) \vdash c_{\mathrm{BS}} :: \mathit{LT}_{\mathrm{ta}}, \mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{HT} \\ \quad \wedge\; \mathit{param}^{\mathrm{BS}}_{\mathrm{ta}}?(c_{\mathrm{BS}}, L, e_{pid}, e_{addr}, e_{intent}, e_{count}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) \\ \quad \wedge\; \mathit{PRE}^{\mathrm{BS}}_{\mathrm{ta}}?(c_{\mathrm{BS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \mathit{HT}, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) \\ \quad \wedge\; \mathit{PRE}^{\mathrm{X}}_{\mathrm{ta}}?(c_{\mathrm{BS}}.x)\} \\ \\ \quad \mathit{SCall}(r, \texttt{pfh\_touch\_addr}, [e_{pid}, e_{addr}, e_{addr}, e_{count}]) \\ \\ \{c'_{\mathrm{BS}} \mid \mathit{POST}^{\mathrm{BS}}_{\mathrm{ta}}?(c'_{\mathrm{BS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \mathit{HT}, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}})) \\ \quad \wedge\; \mathit{POST}^{\mathrm{X}}_{\mathrm{ta}}?(c^0_{\mathrm{BS}}.x, c'_{\mathrm{BS}}.x, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) \\ \quad \wedge\; r = \mathit{pfh\text{-}ta\text{-}res}_{\mathrm{SS}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) \\ \quad \wedge\; \mathit{TE}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}) \vdash c'_{\mathrm{BS}} :: \mathit{LT}_{\mathrm{ta}}, \mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{HT} \\ \quad \wedge\; \mathit{modifies}^{\mathrm{BS}}_{\mathrm{ta}}?(c^0_{\mathrm{BS}}, c'_{\mathrm{BS}}, \mathit{TE}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{HT}, L, r, \mathit{GT}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}})\} \end{array}$$

◀ LEMMA 6.38
Page-fault handler correctness at the level of BS w.r.t. the extended program

PROOF ► The proof is similar to the proof Lemma 6.37. We use Lemma 6.30 insted of Lemma 6.27. ■

6.4 Abstraction Relation from SS

This section introduces abstraction relation

$$abs_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}, gvars_{\mathrm{active}}, gvars_{\mathrm{free}})$$

which states that SS implementation state c_{SS} encodes abstract PFH configuration c_{PFH} with respect to type environment $te :: \textit{tenv-t}$ and lists of active and free pointer variables $gvars_{\mathrm{active}}, gvars_{\mathrm{free}} :: \textit{gvar-t}^*$ which specify the positions of the respective lists in memory. The definition of the predicate follows its counterparts for Simpl (Section 5.17) and big-step semantics (Section 6.17). Below we highlight the relevant differences in memory models of small-step and big-step semantics.

Differences in the BS and SS memory models

We have described the memory model of C0 big-step semantics in Section 3.3 and the memory model of C0 small-step semantics in Section 2.3. The key difference between these two model is values representation: big-step semantics features compound values whereas small-step semantics uses flat values.

On the big-step level global, local, and heap memories are modeled as separate partial mapping from variable names (locations) to compound values. These values are modeled by inductive data type *val-t* with individual constructors for primitive, array, and structure values. The primitive values are also modeled by an inductive data type whose constructors correspond to the primitive types supported by C0.

On the small-step level global, local, and heap memories are modeled by means of memory frames. A memory frame stores its symbol table which is a list of pairs of variable names and types, the set of initialized variables, and the content. The content is a mapping from natural numbers to memory cells. A memory cell stores a single primitive value and it is modeled by inductive data type *mcell-t* whose constructors correspond to the primitive C0 types. As the reader might observe, there are no special data types to model array and structure values. Thus, such compound values are flattened and stored within consecutive memory cells.

The variables are modeled within the small-step semantics by inductive data type *gvar-t*. It has constructors to model global, local, and heap variable as well as array elements and structure fields. Having a variable g we can compute its content in the C0 small-step memory configuration *mem* by means of the function $\textit{value}_{\mathrm{g}}(\textit{mem}, g)$ [Lei07, Definition 4.22]. The obtained content *ct* is a mapping from natural numbers to memory cells. This content computation function will be used in definitions of abstractions relations from SS for different variables from the demand paging implementation. The common idea behind these abstraction relations is to obtain the content of the respective SS variable by means of the function $\textit{value}_{\mathrm{g}}$ and to express the obtained result with the help of inductive constructors of data type *mcell-t* and the corresponding HOL values taken from the abstract PFH configuration.

6.4.1 Doubly-Linked Lists

Following ideas behind definitions of lists abstractions in Hoare logic (Definition 5.1) and big-step semantics (Definition 6.1) we introduce a list formalization in small-step semantics by predicate

$$\mathit{list}_{\mathrm{SS}}? :: C_{\mathrm{SS}} \times \mathit{tenv\text{-}t} \times \mathit{gvar\text{-}t}_{\perp} \times \mathbb{S} \times \mathit{gvar\text{-}t}^{*} \mapsto \mathbb{B}.$$

It takes the following parameters: (i) complete SS state c_{SS} and (ii) type environment te to evaluate expressions, (iii) pointer p to the first element of the list, (iv) field name *next* which corresponds to the list's "next"-field which yields successor elements, and (v) list of variables l specifying the list structure.

The list predicate in the small-step semantics means that in non-degenerate case p is the first pointer in list l and we can obtain by evaluation the content ct of p's successor which being converted to a pointer maintains the SS list predicate with the l's tail l':

◀ DEFINITION 6.39
List abstracted from SS

$$\mathit{list}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, p, \mathit{next}, l) = \begin{cases} p = \perp & \text{if } l = [\,] \\ p = a \wedge \exists\, \mathit{ds} : \mathit{eval}_{\mathrm{SS}}(te, c_{\mathrm{SS}}.\mathit{mem}, p') = \lfloor \mathit{ct} \rfloor & \\ \qquad \wedge\ \mathit{list}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, \mathit{mem2ptr}(\mathit{ct}(0)), \mathit{next}, l') & \text{if } l = a \circ l', \end{cases}$$

where $p' = \mathit{StructAcc}(\mathit{Deref}(\mathit{Lit}(\mathit{Prim}(\mathit{Addr}(\mathit{gvar2loc}(a))))), \mathit{next})$. Note that contents ct are mappings from natural numbers to memory cells. The result of evaluation in our case is stored at index zero: $\mathit{ct}(0)$. In the definition function $\mathit{mem2ptr}(m)$ above converts memory cell m to a pointer variable, and $\mathit{gvar2loc}(a)$ converts variable a to a location.

An SS doubly-linked list is formalized by application of the SS list predicate in both directions. Pointer q to the list's last element and the "predecessor"-field name *prev* are additionally involved:

$$\mathit{dlist}_{\mathrm{SS}}? :: C_{\mathrm{SS}} \times \mathit{tenv\text{-}t} \times \mathit{gvar\text{-}t}_{\perp} \times \mathit{gvar\text{-}t}_{\perp} \times \mathbb{S} \times \mathbb{S} \times \mathit{gvar\text{-}t}^{*} \mapsto \mathbb{B},$$

◀ DEFINITION 6.40
Doubly-linked list abstracted from SS

$$\mathit{dlist}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, p, q, \mathit{next}, \mathit{prev}, l) = \mathit{list}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, p, \mathit{next}, l) \wedge \mathit{list}_{\mathrm{SS}}?(c_{\mathrm{BS}}, te, q, \mathit{prev}, \mathit{rev}(l)).$$

An instantiation for lists of page descriptors is defined as follows:

$$\mathit{dlist}_{\mathrm{SS}}^{\mathrm{pd}}?(c_{\mathrm{SS}}, te, p, q, l) = \mathit{dlist}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, p, q, \texttt{next}, \texttt{prev}, l).$$

6.4.2 Page and Big-Page Management

In the same way we did for the abstraction relations from Simpl and and from BS we first introduce an abstraction relation for lists of page descriptors. The predicate below states that a lists of page descriptors starts at a variable of name *vn* in the memory of a small-step configuration c_{SS} with respect to a type environment te. The list's content is specified by a list of abstract page descriptors *pds* and its structure in the memory is given by a list of pointer variables l (cf. Definitions 5.3 and 6.3).

In the definition below the function $\mathit{eval}_{\mathrm{pd}}(\mathit{fn})$ is used to access the field with name *fn* of the doubly-linked list of page descriptors element pointed to by $l[i]$. This is achieved by evaluating the respective C0 expression for structure field access $\mathit{StructAcc}(\mathit{Deref}(\mathit{Lit}(\mathit{Prim}(\mathit{Addr}(\mathit{gvar2loc}(l[i]))))))$ with the function

$eval_{\mathrm{SS}}$. If succeeds, this evaluation function returns some content. We apply $eval_{\mathrm{pd}}$ for the page descriptor fields `pid`, `vpx`, and `ppx`. We claim that evaluation succeeds and we obtain the corresponding contents ct_{pid}, ct_{vpx}, ct_{ppx}. These contents contain at position zero the memory cells storing the respective values from the PFH abstraction: $pds[i].pid$, $pds[i].vpx$, and $pds[i].ppx$.

DEFINITION 6.41 ▶ List of page descriptors abstracted from SS

$$
\begin{aligned}
&\mathit{pds\text{-}abs}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, vn, l, pds) = \\
&\quad |l| = |pds| \\
&\wedge (\exists\, q : \mathit{dlist}^{\mathrm{pd}}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, \mathit{mem2ptr}(\mathit{data}(te, c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{gm}}(vn))), q, l)) \\
&\wedge \forall\, i < |l|, \mathit{eval}_{\mathrm{pd}} = \lambda\, fn : \mathit{eval}_{\mathrm{SS}}(te, c_{\mathrm{SS}}.\mathit{mem}, \mathit{StructAcc}(\mathit{Deref}(\\
&\qquad\qquad \mathit{Lit}(\mathit{Prim}(\mathit{Addr}(\mathit{gvar2loc}(l[i])))))), fn)) : \\
&\quad \exists\, ct_{\mathrm{pid}}, ct_{\mathrm{vpx}}, ct_{\mathrm{ppx}} : \\
&\qquad \mathit{eval}_{\mathrm{pd}}(\texttt{pid}) = \lfloor ct_{\mathrm{pid}} \rfloor \wedge \mathit{eval}_{\mathrm{pd}}(\texttt{vpx}) = \lfloor ct_{\mathrm{vpx}} \rfloor \wedge \mathit{eval}_{\mathrm{ct}}(\texttt{ppx}) = \lfloor ct_{\mathrm{ppx}} \rfloor \\
&\quad \wedge\, ct_{\mathrm{pid}}(0) = \mathit{mcell}_{\mathrm{nat}}(pds[i].pid) \\
&\quad \wedge\, ct_{\mathrm{vpx}}(0) = \mathit{mcell}_{\mathrm{nat}}(pds[i].vpx) \\
&\quad \wedge\, ct_{\mathrm{ppx}}(0) = \mathit{mcell}_{\mathrm{nat}}(pds[i].ppx).
\end{aligned}
$$

Substituting the list's name with `active` and its specification lists with $c_{\mathrm{PFH}}.\mathit{active}$ and $\mathit{gvars}_{\mathrm{active}}$ we obtain the abstraction relation for the active list cf. Definitions 5.4 and 6.4.

DEFINITION 6.42 ▶ Active list abstracted from SS

$$
\begin{aligned}
&\mathit{active\text{-}abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{gvars}_{\mathrm{active}}) = \\
&\qquad\qquad \mathit{pds\text{-}abs}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, \texttt{active}, \mathit{gvars}_{\mathrm{active}}, c_{\mathrm{PFH}}.\mathit{active}).
\end{aligned}
$$

Setting the lists's name to `free` and providing $c_{\mathrm{PFH}}.\mathit{free}$ and $\mathit{gvars}_{\mathrm{free}}$ as its specification we have the abstraction relation for the free list (cf. Definitions 5.5 and 6.5).

DEFINITION 6.43 ▶ Free list abstracted from SS

$$
\begin{aligned}
&\mathit{free\text{-}abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{gvars}_{\mathrm{free}}) = \\
&\qquad\qquad \mathit{pds\text{-}abs}_{\mathrm{SS}}?(c_{\mathrm{SS}}, te, \texttt{free}, \mathit{gvars}_{\mathrm{free}}, c_{\mathrm{PFH}}.\mathit{free}).
\end{aligned}
$$

Note that lists of active and free pointer variables $\mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}} :: \mathit{gvar}\text{-}t^*$ which specify the positions of the respective lists in memory are specified later in Definition 6.56.

The abstraction relation for the stack of free big pages states that data read from the SS implementation array `bpfree` in the memory configuration $c_{\mathrm{SS}}.\mathit{mem}$ matches the corresponding specification array (cf. Definitions 5.6 and 6.6).

DEFINITION 6.44 ▶ Stack of free big pages abstracted from SS

$$
\begin{aligned}
&\mathit{bpfree\text{-}abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}) = \forall\, i < |c_{\mathrm{PFH}}.\mathit{bpfree}| : \\
&\qquad \mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{arr}}(\mathit{gvar}_{\mathrm{gm}}(\texttt{bpfree}), i)) = \mathit{mcell}_{\mathrm{nat}}(c_{\mathrm{PFH}}.\mathit{bpfree}[i]).
\end{aligned}
$$

Recall that the function $\mathit{value}_{\mathrm{g}}(\mathit{mem}, g)$ [Lei07, Definition 4.22] yields the content of the variable g in the C0 small-step memory configuration mem. In this section, we are interested only in the memory cell which resides at index zero of the obtained content. Because of that, we allow ourselves to abuse notation and write $\mathit{value}_{\mathrm{g}}(\mathit{mem}, g)$ instead of $\mathit{value}_{\mathrm{g}}(\mathit{mem}, g)(0)$.

6.4.3 Page and Big-Page Tables

Notice a difference in numbering heap addresses in the small-step semantics and on the level of Simpl/BS. In the former we count heap addresses starting from zero whereas on the latter we start from one. That is why the abstraction relation for the page table space first claims that the data stored at the variable `pt` is a pointer to the heap memory cell at address zero. Reading data from this variable gives us a two dimensional array with values matching the page table space from the PFH abstract state (cf. Definitions 5.7 and 6.7).

$$\begin{aligned} &\textit{pt-abs}_{\text{SS}}?(te, c_{\text{SS}}, c_{\text{PFH}}) = \\ &\qquad \textit{value}_{\text{g}}(c_{\text{SS}}.\textit{mem}, \textit{gvar}_{\text{gm}}(\texttt{pt})) = \textit{mcell}_{\text{ptr}}(\textit{gvar}_{\text{hm}}(0)) \\ &\quad \wedge \forall\, i < |c_{\text{PFH}}.\textit{pt}|, j < |c_{\text{PFH}}.\textit{pt}[i]| : \\ &\qquad \textit{value}_{\text{g}}(c_{\text{SS}}.\textit{mem}, \textit{gvar}_{\text{arr}}(\textit{gvar}_{\text{arr}}(\textit{gvar}_{\text{hm}}(0), i), j)) = \\ &\qquad \textit{mcell}_{\text{nat}}(c_{\text{PFH}}.\textit{pt}[i][j]). \end{aligned}$$

◀ DEFINITION 6.45
Page table space abstracted from SS

The abstraction relation for the big-page table space checks whether the data read from the variable `bpt` matches the abstract big-page tabel space (cf. Definitions 5.8 and 6.8).

$$\begin{aligned} &\textit{bpt-abs}_{\text{SS}}?(te, c_{\text{SS}}, c_{\text{PFH}}) = \forall\, i < |c_{\text{PFH}}.\textit{bpt}| : \\ &\qquad \textit{value}_{\text{g}}(c_{\text{SS}}.\textit{mem}, \textit{gvar}_{\text{arr}}(\textit{gvar}_{\text{gm}}(\texttt{bpt}), i)) = \textit{mcell}_{\text{nat}}(c_{\text{PFH}}.\textit{bpt}[i]). \end{aligned}$$

◀ DEFINITION 6.46
Big-page table space abstracted from SS

6.4.4 Process Control Blocks

The abstraction relation for the page table origins claims that the data obtained at the index `PTO` of the field `ef` of the variable `pcb` matches the page table origin values from the abstract PFH state for all user processes (cf. Definitions 5.9 and 6.9).

$$\begin{aligned} &\textit{pto-abs}_{\text{SS}}?(te, c_{\text{SS}}, c_{\text{PFH}}) = \forall\, i : \textit{user-pid}?(i) \longrightarrow \\ &\quad \textit{value}_{\text{g}}(c_{\text{SS}}.\textit{mem}, \textit{gvar}_{\text{arr}}(\textit{gvar}_{\text{str}}(\textit{gvar}_{\text{arr}}(\textit{gvar}_{\text{gm}}(\texttt{pcb}), i), \texttt{ef}), \texttt{PTO})) = \\ &\qquad \textit{mcell}_{\text{int}}(c_{\text{PFH}}.\textit{pto}[i]). \end{aligned}$$

◀ DEFINITION 6.47
Page table origins abstracted from SS

The abstraction relation for the page table lengths $\textit{ptl-abs}_{\text{SS}}?(te, c_{\text{SS}}, c_{\text{PFH}})$ is defined similarly to the predicate above but `PTL` and $c_{\text{PFH}}.\textit{ptl}$ are used (cf. Definitions 5.10 and 6.10).

◀ DEFINITION 6.48
Page table lengths abstracted from SS

The abstraction relation for the big-page table origins states that the data read from the field `bpto` of the variable `pcb` matches the big-page table origin values from the abstract PFH configuration for all user processes (cf. Definitions 5.11 and 6.11).

$$\begin{aligned} &\textit{bpto-abs}_{\text{SS}}?(te, c_{\text{SS}}, c_{\text{PFH}}) = \forall\, i : \textit{user-pid}?(i) \longrightarrow \\ &\quad \textit{value}_{\text{g}}(c_{\text{SS}}.\textit{mem}, \textit{gvar}_{\text{str}}(\textit{gvar}_{\text{arr}}(\textit{gvar}_{\text{gm}}(\texttt{pcb}), i), \texttt{bpto})) = \\ &\qquad \textit{mcell}_{\text{int}}(c_{\text{PFH}}.\textit{bpto}[i]). \end{aligned}$$

◀ DEFINITION 6.49
Big-page table origins abstracted from SS

The abstraction relation for the big-page table lengths, denoted by $\textit{bptl-abs}_{\text{SS}}?(te, c_{\text{SS}}, c_{\text{PFH}})$, is defined similarly to the predicate above but `bptl` and $c_{\text{PFH}}.\textit{bptl}$ are used (cf. Definitions 5.12 and 6.12.

◀ DEFINITION 6.50
Big-page table lengths abstracted from SS

6.4.5 Miscellaneous

The abstraction relations for the number of free physical pages, free big pages, and used virtual pages are straightforward and follow Definitions 5.13–5.15 on the Simpl level and 6.13–6.15 on the big-step level.

DEFINITION 6.51 ► Free physical pages abstracted from SS

$$\mathit{pages\text{-}free\text{-}abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}) = \mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{gm}}(\texttt{page\_free})) = \mathit{mcell}_{\mathrm{nat}}(|c_{\mathrm{PFH}}.\mathit{free}|).$$

DEFINITION 6.52 ► Free big pages abstracted from SS

$$\mathit{bpages\text{-}free\text{-}abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}) = \mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{gm}}(\texttt{bpages\_free})) = \mathit{mcell}_{\mathrm{nat}}(|c_{\mathrm{PFH}}.\mathit{bpfree}|).$$

DEFINITION 6.53 ► Used virtual pages abstracted from SS

$$\mathit{pages\text{-}used\text{-}abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}) = \mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{gm}}(\texttt{pages\_used})) = \mathit{mcell}_{\mathrm{nat}}(\sum_{i=1}^{\texttt{MAX\_PID}-1} (c_{\mathrm{PFH}}.\mathit{ptl}[i] + 1)).$$

The abstraction relation for the reverse lookup array states that for all user physical page indices i the pointer which resides in the implementation array `ppx2pd` at index i belongs to the set of active and free pointer variables. By dereferencing this pointer at the field `ppx` we obtain the value of i (cf. Definitions 5.16 and 6.16).

DEFINITION 6.54 ► Reverse lookup array abstracted from SS

$$\begin{aligned}&\mathit{ppx2pd\text{-}abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, \mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}}) = \\ &\quad \forall\, i, \mathit{ptr}_{\mathrm{pd}} = \lambda\, j : \mathit{mem2ptr}(\mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{arr}}(\mathit{gvar}_{\mathrm{gm}}(\texttt{ppx2pd}), j)) : \\ &\qquad \mathit{user\text{-}ppx}?(i) \longrightarrow \mathit{ptr}_{\mathrm{pd}}(i) \in \mathit{gvars}_{\mathrm{active}} \circ \mathit{gvars}_{\mathrm{free}} \\ &\qquad\quad \wedge\, \mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{str}}(\mathit{ptr}_{\mathrm{pd}}(i), \texttt{ppx})) = \mathit{mcell}_{\mathrm{nat}}(i).\end{aligned}$$

6.4.6 Altogether

Analogously to the abstraction relations from Simpl and from BS we combined the individual components defined above into the single abstraction relation from SS.

DEFINITION 6.55 ► Abstraction relation from SS

The overall abstraction relation from the SS implementation towards abstract PFH configurations $\mathit{abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}})$ is a conjunction of Definitions 6.42–6.54.

We combine the overall abstraction relation together with validity properties of the abstract page-fault handler.

DEFINITION 6.56 ► Abstraction relation from SS combined with validity

$$\begin{aligned}&\mathit{abs}_{\mathrm{SS}}\surd(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}}) = \\ &\quad \mathit{abs}_{\mathrm{SS}}?(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}}) \\ &\quad \wedge\, \mathit{pfh}\surd(c_{\mathrm{PFH}}) \\ &\quad \wedge\, \mathit{gvars}_{\mathrm{active}} \cap \mathit{gvars}_{\mathrm{free}} = \emptyset \\ &\quad \wedge\, \mathit{gvars}_{\mathrm{active}} \cup \mathit{gvars}_{\mathrm{free}} = \{\mathit{gvar}_{\mathrm{hm}}(i) \mid 1 \leq i \leq \texttt{USER\_PGS}\}.\end{aligned}$$

6.5 Property Transfer from BS to SS

6.5.1 Mapping BS States to SS States

In Section 3.5 we considered a meta theorem for transfer correctness of function calls from the big-step to the small step semantics (Theorem 3.4). This theorem has free variables for BS and SS configurations which have to match each other. In order to instantiate these variables we need to define an abstraction function which constructs a big-step state from a small-step state. Unlike the state abstraction function from Simpl towards BS (cf. Definition 6.19) the abstraction function from SS towards BS can be defined generically. Its was defined formally in Isabelle/HOL by Schirmer.

First we define how values are abstracted with respect to BS and SS semantics, and then introduce the abstraction function for states.

Value abstraction. C0 small-step semantics values are abstracted to big-step semantics by means of function

$$SS2BS_{\text{val}} :: (\mathbb{N} \mapsto \textit{mcell-t}) \times \textit{ty-t} \times \mathbb{N} \mapsto \textit{val-t}_{\perp}$$

◀ DEFINITION 6.57
Abstraction of values from SS towards BS

It takes as arguments an SS content ct, type ty, and address within the content a and returns a corresponding BS value. The function is defined by induction on the type constructors. For primitive types ty listed in Table 6.1 the function is defined as follows:

$$SS2BS_{\text{val}}(ct, ty, a) = \begin{cases} \lfloor Prim(v) \rfloor & ct(a) = m \\ \perp & \text{otherwise} \end{cases}$$

with values v and memory cells m specified in Table 6.1.

Table 6.1: Abstraction of primitive values from SS to BS

Type ty	Value v	Memory cell m
$BooleanT$	$Bool(b)$	$mcell_{\text{bool}}(b)$
$IntegerT$	$Intg(i)$	$mcell_{\text{int}}(i)$
$CharT$	$Chr(c)$	$mcell_{\text{char}}(c)$
$UnsgndT$	$Unsgnd(n)$	$mcell_{\text{nat}}(n)$
$NullT$	$Null$	$mcell_{\text{ptr}}(\perp)$
$PtrT(tn)$	$Addr(gvar2loc(gv))$	$mcell_{\text{ptr}}(gv)$

For the array type the function $SS2BS_{\text{val}}(ct, ArrT(n, ty), a)$ is additionally recursive on the array size n. For $n = 0$ the function yields an empty array. For $n = m + 1$ the function abstracts the values of array's elements.

$$SS2BS_{\text{val}}(ct, ArrT(n, ty), a) = \begin{cases} \lfloor Arr([\,]) \rfloor & \text{if } n = 0 \\ \lfloor Arr(v \circ vs) \rfloor & \text{if } SS2BS_{\text{val}}(ct, ArrT(m, ty), a + size_{\text{t}}(ty)) = \lfloor Arr(vs) \rfloor \\ & \quad \wedge\ SS2BS_{\text{val}}(ct, ty, a) = \lfloor v \rfloor \wedge n = m+1 \\ \perp & \text{otherwise} \end{cases}$$

Above, the function $size_{\text{t}}(ty)$ [Lei07, Definition 4.1] computes the size of a type (number of elementary values in a flattened value of a type).

Finally, for the structure type the function $SS2BS_{\text{val}}(ct, Struct(fs), a)$ is defined quite similarly as for the array type. The definition is additionally recursive on the list fs of structure fields. For $fs = [\,]$ the obtained value is an empty structure. For $fs = x \circ xs$ the function abstract values of structure fields.

$$SS2BS_{\text{val}}(ct, Struct(fs), a) = \begin{cases} \lfloor Struct([\,]) \rfloor & \text{if } fs = [\,] \\ \lfloor Struct((fst(x), v) \circ vs) \rfloor & \text{if } SS2BS_{\text{val}}(ct, Struct(xs), a + size_{\text{t}}(snd(x))) = \\ & \quad \lfloor Struct(vs) \rfloor \\ & \quad \wedge SS2BS_{\text{val}}(ct, snd(x), a) = \lfloor v \rfloor \wedge fs = x \circ xs \\ \bot & \text{otherwise} \end{cases}$$

State abstraction. We denote the *type* of a variable vn in a given symbol table st by $type_{\text{v}}(st, vn)$ ([Lei07, Definition 4.14]). We denote the *base address* of a variable vn in a given symbol table st by $ba_{\text{v}}(st, vn)$ ([Lei07, Definition 4.13]).

States of the small-step semantics are abstracted towards big-step states by means of the function

DEFINITION 6.58 ▶
Abstraction of states from SS towards BS

$$SS2BS_{\text{state}} :: C_{\text{SS}} \mapsto C_{\text{BS}},$$
$$SS2BS_{\text{state}}(c_{\text{SS}}) = c_{\text{BS}}.$$

Below we describe how the abstracted c_{BS} state is obtained. Let us abbreviate different SS memory components: $gm = c_{\text{SS}}.mem.gm$, $lm = c_{\text{SS}}.mem.lm$, and $hm = c_{\text{SS}}.mem.hm$.

The BS heap memory component is obtained by abstraction of all heap variable from the SS heap symbol table:

$$c_{\text{BS}}.heap(l) = \begin{cases} SS2BS_{\text{val}}(hm.ct, snd(hm.st[l-1]), & \\ \qquad idx2addr(hm.st, l-1)) & \text{if } l \leq |hm.st| \\ \bot & \text{otherwise} \end{cases}$$

Above, with $idx2addr(st, i)$ we compute the base address of the i-th value according to the sizes of proceeding types in the symbol table st.

In oder to define the local variables of the big-step semantic configuration $c_{\text{BS}}.lvars(vn)$ we first check whether the local memory frame is empty $lm = [\,]$. If so, we assign $c_{\text{BS}}.lvars(vn) = \bot$. Otherwise, $lm = l \circ ls$ and we compute the type and base address of the variable vn in the local memory frame l. If we do not succeed with that, i.e., either $type_{\text{v}}(fst(l).st, vn) = \bot$ or $ba_{\text{v}}(fst(l).st, vn) = \bot$, we assign $c_{\text{BS}}.lvars(vn) = \bot$ alike. If this is not the case and $type_{\text{v}}(fst(l).st, vn) = \lfloor ty \rfloor$ and $ba_{\text{v}}(fst(l).st, vn) = \lfloor ba \rfloor$ we set

$$c_{\text{BS}}.lvars(vn) = \begin{cases} SS2BS_{\text{val}}(fst(l).ct, ty, ba) & \text{if } vn \in fst(l).init \\ \bot & \text{otherwise} \end{cases}$$

Finally, for the global variables component $c_{\text{BS}}.lvars(vn)$ we compute the type $type_{\text{v}}(gm.st, vn)$ and the base address $ba_{\text{v}}(gm.st, vn)$ of the variable vn in the global memory frame. If at least one of those is $\bot$ we assign $c_{\text{BS}}.gvars(vn) = \bot$. Otherwise, we have the type and base address values $\lfloor ty \rfloor$ and $\lfloor ba \rfloor$, respectively, and assign

$$c_{\text{BS}}.gvars(vn) = SS2BS_{\text{val}}(gm.ct, ty, ba).$$

Having the abstraction relation between SS and BS states we can transfer

correctness results of the demand paging implementation further to the level of small-step semantics. Similarly to a transfer from Simpl to BS we first show lemmas about transfer of a valid abstraction relation from BS to SS and vice versa. These lemmas will be essential arguments in the proofs of the demand paging implementation down to the SS level.

6.5.2 Transfer of Abstraction Relation

Transfer of the valid abstraction relation from BS to SS. In order to formulate a lemma that a valid abstraction relation can be transferred from the big-step to the small-step level we need to formalize the structure of the heap memory in the small-step semantics. Recall that, the demand paging implementation allocates the page table space and `USER_PGS` page descriptors on the heap. Moreover, no other part of the CVM implementation stores data on the heap. Below we define a constant hty_{CVM} which is a list of types of elements allocated by the demand paging code in the heap memory. By hst_{CVM} we denote a corresponding heap symbol table. Recall that a symbol table is a list of pairs consisting of variable names and their types. Since variables on the heap are nameless we use the polymorphic arbitrary constant A as the first component (variable name) of each element of hst_{CVM}. Additionally, we introduce a predicate which tests whether the list of second symbol table's components of a given heap memory frame hm starts with hty_{CVM}.

◀ DEFINITION 6.59
SS heap structure of the demand paging

$$\begin{aligned} hty_{\text{CVM}} &= ty_{\text{pt}} \circ rep(ty_{\text{pd}}, \texttt{USER\_PGS}) \\ hst_{\text{CVM}} &= map(\lambda x : (\mathsf{A}, x), hty_{\text{CVM}}) \\ \textit{prefix-}hst_{\text{CVM}}?(hm) &= \forall\, i \leq \texttt{USER\_PGS} : snd(hm.st) = hty_{\text{CVM}}[i] \end{aligned}$$

The lemma below states that a valid abstraction relation from BS towards the abstract PFH state implies the valid abstraction relation from SS towards the abstract PFH state. The big-step configuration is obtained by abstracting a given small-step configuration c_{SS} with the state abstraction function (Definition 6.58). Additional assumptions to the lemma are: (i) the small-step configuration is valid, (ii) the global symbol table of c_{SS} coincides with the linked symbol table of the concrete kernel, and (iii) the SS heap structure of the demand paging is respected. With *loc2gvar*(l) we convert a location l to a pointer variable.

◀ LEMMA 6.60
Valid abstraction relation transfer from BS to SS

$$\begin{aligned} & abs_{\text{BS}}\surd(SS2BS_{\text{state}}(c_{\text{SS}}), c_{\text{PFH}}, HT, locs_{\text{active}}, locs_{\text{free}}) \\ \wedge\ & c_{\text{SS}} \in valid_{\text{SS}}(te_{\text{CK}}(\Pi_{\text{AK}}), ft) \\ \wedge\ & gst(c_{\text{SS}}.mem) = gst_{\text{CK}}(\Pi_{\text{AK}}) \\ \wedge\ & \textit{prefix-}hst_{\text{CVM}}?(c_{\text{SS}}.mem.hm) \\ \longrightarrow\ & abs_{\text{SS}}\surd(te_{\text{CK}}(\Pi_{\text{AK}}), c_{\text{SS}}, c_{\text{PFH}}, map(loc2gvar, locs_{\text{active}}), \\ & \qquad\qquad map(loc2gvar, locs_{\text{free}})) \end{aligned}$$

◀ PROOF

The lemma's proof idea is quite similar to those of Lemma 6.20. The valid abstraction relation in the assumptions maps the BS state to the abstract PFH state. This BS state is obtained by abstracting the corresponding SS state. Therefore, we can also map the SS state to the abstract PFH configuration. ■

Transfer of the valid abstraction relation from SS to BS. The lemma for the implication in the direction from SS to BS is quite similar to the previous one, though, has two additional assumptions.

The first assumption is the abstract kernel properties $\mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}})$ (cf. Section 6.4 of Tsyban's thesis [Tsy09]).

The second assumption relates an SS type environment, an SS heap memory frame, and a BS heap typing. Heap locations in the small-step semantics are directly connected to the types, whereas in the big-step semantics there is a indirection via the type name stored in the heap typing. Since the mapping from type names to types specified by an SS type environment does not necessarily have to be injective we require that there must be at least one name that maps to the SS type according to the small-step heap symbol table. The described idea is formalized in the definition below. We consider all BS heap locations l. If the index number of such a location is bounded by the size of the SS heap symbol table $|\mathit{hst}(\mathit{mem})|$ we require existence of a type name tn such that (i) the heap typing maps to this type name at the location l, and (ii) this type name corresponds to the type $\mathit{snd}(\mathit{hst}(\mathit{mem})[l-1])$ in the SS type environment te. Note an index shift by one in referring to BS and SS heap locations: we start counting the former from one wheres the latter are counted starting from zero.

DEFINITION 6.61 ▶

$$\mathit{abs\text{-}heap\text{-}ty?}(\mathit{te}, \mathit{mem}, \mathit{HT}) = \forall l : \begin{cases} \exists \mathit{tn} : \mathit{HT}(l) = \lfloor \mathit{tn} \rfloor \wedge (\mathit{tn}, \mathit{snd}(\mathit{hst}(\mathit{mem})[l-1])) \in \mathit{te} & \text{if } l-1 < |\mathit{hst}(\mathit{mem})| \\ \mathit{HT}(l) = \bot & \text{otherwise} \end{cases}$$

LEMMA 6.62 ▶

Valid abstraction relation transfer from SS to BS

$$\begin{aligned} & \mathit{abs}_{\mathrm{SS}}\surd(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}}) \\ \wedge\ & c_{\mathrm{SS}} \in \mathit{valid}_{\mathrm{SS}}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}) \\ \wedge\ & \mathit{gst}(c_{\mathrm{SS}}.\mathit{mem}) = \mathit{gst}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}) \\ \wedge\ & \mathit{prefix\text{-}hst}_{\mathrm{CVM}}?(c_{\mathrm{SS}}.\mathit{mem}.\mathit{hm}) \\ \wedge\ & \mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \\ \wedge\ & \mathit{abs\text{-}heap\text{-}ty?}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}.\mathit{mem}) \\ \longrightarrow\ & \mathit{abs}_{\mathrm{BS}}\surd(\mathit{SS2BS}_{\mathrm{state}}(c_{\mathrm{SS}}), c_{\mathrm{PFH}}, \mathit{HT}, \mathit{map}(\mathit{gvar2loc}, \mathit{gvars}_{\mathrm{active}}), \\ & \qquad \mathit{map}(\mathit{gvar2loc}, \mathit{gvars}_{\mathrm{free}})) \end{aligned}$$

6.5.3 Specification at the Level of SS

Pre- and postconditions to the functions of the demand paging implementation formulated at the level of the small-step semantics express the same meaning as those formulated at the level of Simpl. Since the only difference between them is data representation — and we have already illustrated this difference while defining the abstraction relation from SS (Definition 6.55) — we do not present formal definitions of pre- and postconditions but only declare predicates for them.

Initialization code. The precondition of the initialization code at the level of SS is stated by the predicate

$$\mathit{PRE}^{\mathrm{SS}}_{\mathrm{init}}?(c_{\mathrm{SS}}, \mathit{te}).$$

Its arguments are a type environment te, an SS state c_{SS}. The postcondition of the initialization code is given by the predicate

$$\mathit{POST}^{\mathrm{SS}}_{\mathrm{init}}?(c'_{\mathrm{SS}}, r, \mathit{te}).$$

It additionally takes a result variable r and lists of active and free pointer variables. Formal definitions of the predicates follow Section 5.3.2. The essential feature of the specification it that the postcondition establishes the valid abstraction relation from SS (Definition 6.56) with the initial abstract PFH configuration:

$$abs_{\mathrm{SS}}\surd(te, c_{\mathrm{SS}}, \mathit{init}\text{-}c_{\mathrm{PFH}}, [\,], [\mathit{gvar}_{\mathrm{hm}}(\texttt{USER\_PGS}), \ldots, \mathit{gvar}_{\mathrm{hm}}(1)]).$$

The set of names of global variables changed by the functions is shared with the big-step specification (Definition 6.23).

Page-fault handler. The precondition of the page-fault handler at the level of SS is formulated as the predicate

$$PRE^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, te, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}})$$

which additionally to the discussed parameters takes values of the handler's call arguments. The postcondition of the page-fault handler in small-step semantics is given by the predicate

$$POST^{\mathrm{SS}}_{\mathrm{ta}}?(c'_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, te, \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}).$$

The result of the page fault handler on the level of SS is computed by the function

$$\mathit{pfh}\text{-}\mathit{ta}\text{-}\mathit{res}_{\mathrm{SS}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}).$$

Formal definitions of the predicates follow Section 5.6.2. We stress the fact that the most important conjunct of both pre- and postconditions is the valid abstraction relation from SS towards abstract PFH state (Definition 6.56):

$$abs_{\mathrm{SS}}\surd(te, c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}}).$$

The set of names of global and heap variables changed by the functions is shared with the big-step specification (Definition 6.24).

6.5.4 Correctness at the Level of SS

Transfer of the initialization code correctness from BS to SS. The lemma stated below describes the correctness of the demand paging initialization code at the level of small-step semantics. The lemma assumes the abstract kernel properties to hold and the result variable r to be in the context of caller's variables L. The lemma conclude an SS Hoare triple with respect to the type environments of the concrete kernel $te_{\mathrm{CK}}(\Pi_{\mathrm{AK}})$, the extended function table $ft_{\mathrm{ext}}(\Pi_{\mathrm{AK}})$, the extended semantics environment xsem, and the extended program $\Pi_{\mathrm{ext}}(\Pi_{\mathrm{AK}})$.

We define the predicate $\mathit{modifies}^{\mathrm{SS}}_{\mathrm{init}}?$ which compares two small-step states c_{SS} and c'_{SS} and describes which variables were modified by the initialization code. The predicate states that (i) all global variables of the concrete kernel except those that are in the set $\mathit{modified}_{\mathrm{init}}$ remain the same, and (ii) all initialized top-local variables except for r stay the same.

◀ DEFINITION 6.63
Modification of SS variables by initialization code

$$\begin{aligned}
&\mathit{modifies}^{\mathrm{SS}}_{\mathrm{init}}?(c_{\mathrm{SS}}, c'_{\mathrm{SS}}, r) = \\
&\quad (\forall v : v \in \mathit{map}(\mathit{fst}, \mathit{gst}_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \wedge v \notin \mathit{modified}_{\mathrm{init}} \\
&\qquad \longrightarrow \mathit{value}_{\mathrm{g}}(c'_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{gm}}(v)) = \mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{gm}}(v))) \\
&\wedge\ (\forall v : v \in \mathit{map}(\mathit{fst}, \mathit{lst}_{\mathrm{top}}(c_{\mathrm{SS}}.\mathit{mem})) \wedge v \in \mathit{lm}_{\mathrm{top}}(c_{\mathrm{SS}}.\mathit{mem}).\mathit{init} \wedge v \neq r \\
&\qquad \longrightarrow \mathit{value}_{\mathrm{g}}(c'_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{lm}}(n, v)) = \mathit{value}_{\mathrm{g}}(c_{\mathrm{SS}}.\mathit{mem}, \mathit{gvar}_{\mathrm{lm}}(n, v)))
\end{aligned}$$

Above, $n = |c_{\mathrm{SS}}.mem.lm| - 1$.

The precondition of the Hoare triple is a set of pairs of small-step configurations c_{SS} and extended states c_{X} such that (i) the global symbol table of c_{SS} coincides with the symbol table of the concrete kernel, (ii) the heap symbol table of c_{SS} is empty, and (iii) the preconditions $PRE^{\mathrm{SS}}_{\mathrm{init}}?$ and $PRE^{\mathrm{SS}}_{\mathrm{init}}?$ hold.

The Hoare triple's postcondition is stated as a set of pairs of small-step configurations c'_{SS} and extended states c'_{X} for which (i) the postconditions $POST^{\mathrm{SS}}_{\mathrm{init}}?$ and $POST^{\mathrm{X}}_{\mathrm{init}}?$ hold, (ii) the heap symbol table respects the SS heap structure of the demand paging code, and (iii) modification of the variables is described by the predicate $modifies^{\mathrm{SS}}_{\mathrm{init}}?$.

LEMMA 6.64 ▶
Initialization code correctness at the level of SS

$$
\begin{aligned}
&\mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \wedge r \in L \longrightarrow \\
&\Pi_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{xsem}_{\mathrm{PFH}}, L \vDash^{\mathrm{t}}_{\mathrm{SS}} \\
&\qquad \{(c_{\mathrm{SS}}, c_{\mathrm{X}}) \mid c_{\mathrm{SS}} = c^0_{\mathrm{SS}} \wedge c_{\mathrm{X}} = c^0_{\mathrm{X}} \\
&\qquad\qquad \wedge \mathit{gst}(c_{\mathrm{SS}}.mem) = \mathit{gst}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}) \wedge \mathit{hst}(c_{\mathrm{SS}}.mem) = [\,] \\
&\qquad\qquad \wedge PRE^{\mathrm{SS}}_{\mathrm{init}}?(c_{\mathrm{SS}}, te_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \wedge PRE^{\mathrm{X}}_{\mathrm{init}}?(c_{\mathrm{X}})\} \\
&\qquad\qquad \mathit{SCall}(r, \texttt{pfh\_init}, [\,]) \\
&\qquad \{(c'_{\mathrm{SS}}, c'_{\mathrm{X}}) \mid POST^{\mathrm{SS}}_{\mathrm{init}}?(c'_{\mathrm{SS}}, r, te_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \wedge POST^{\mathrm{X}}_{\mathrm{init}}?(c^0_{\mathrm{X}}, c'_{\mathrm{X}}) \\
&\qquad\qquad \wedge \mathit{hst}(c_{\mathrm{SS}}.mem) = \mathit{hst}_{\mathrm{CVM}} \\
&\qquad\qquad \wedge modifies^{\mathrm{SS}}_{\mathrm{init}}?(c^0_{\mathrm{SS}}, c'_{\mathrm{SS}}, r)\}
\end{aligned}
$$

PROOF ▶ We apply transfer theorem from BS to SS (Theorem 3.4) in which the big-step Hoare triple is instantiated with the one concluded in the correctness lemma of the initialization code at the BS level with respect to the extended program (Lemma 6.37). The major proof effort consists of an implication from SS to BS preconditions and from BS to SS postconditions. To conclude that, Lemmas 6.60 and 6.62 are used. ■

Below, we give this lemma an operational-semantics-style look. This reveals a number of terms hidden inside the SS Hoare triple notation. The (extended) validity of C0 states is a lemma's invariant (1,4). The lemma assumes the result variable r to be in the top-local memory frame. On the side of conclusions we additionally have a successful extended C0 computation (2,3) and the transition invariant (5) introduced in Section 3.5 (Definition 3.3).

LEMMA 6.65 ▶
Initialization code correctness at the level of SS: operational-semantics style

$$
\begin{aligned}
&\quad \mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \\
(1)\quad &\wedge\ c_{\mathrm{SS}} \in \mathit{valid}_{\mathrm{SS}}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})) \\
&\wedge\ c_{\mathrm{SS}}.\mathit{prog} = \mathit{SCall}(r, \texttt{pfh\_init}, [\,]) \\
&\wedge\ \mathit{gst}(c_{\mathrm{SS}}.mem) = \mathit{gst}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}) \wedge \mathit{hst}(c_{\mathrm{SS}}.mem) = [\,] \\
&\wedge\ PRE^{\mathrm{SS}}_{\mathrm{init}}?(c_{\mathrm{SS}}, te_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \wedge PRE^{\mathrm{X}}_{\mathrm{init}}?(c_{\mathrm{X}}) \\
&\wedge\ r \in \mathit{map}(\mathit{fst}, \mathit{lm}_{\mathrm{top}}(c_{\mathrm{SS}}.mem)) \\
(2)\quad &\longrightarrow\ \exists n, c'_{\mathrm{SS}}, c'_{\mathrm{X}} : \delta^n_{\mathrm{SSX}}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{xsem}, c_{\mathrm{SS}}, c_{\mathrm{X}}) = \lfloor (c'_{\mathrm{SS}}, c'_{\mathrm{X}}) \rfloor \\
(3)\quad &\qquad \wedge\ c'_{\mathrm{SS}}.\mathit{prog} = \mathit{Skip} \\
(4)\quad &\qquad \wedge\ c'_{\mathrm{SS}} \in \mathit{valid}_{\mathrm{SS}}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})) \\
(5)\quad &\qquad \wedge\ \mathit{trans\text{-}inv}?(c_{\mathrm{SS}}.\mathit{prog}, \mathit{hst}_{\mathrm{CVM}}, c_{\mathrm{SS}}, c'_{\mathrm{SS}}) \\
&\qquad \wedge\ POST^{\mathrm{SS}}_{\mathrm{init}}?(c'_{\mathrm{SS}}, r, te_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \\
&\qquad \wedge\ POST^{\mathrm{X}}_{\mathrm{init}}?(c_{\mathrm{X}}, c'_{\mathrm{X}}) \\
&\qquad \wedge\ modifies^{\mathrm{SS}}_{\mathrm{init}}?(c_{\mathrm{SS}}, c'_{\mathrm{SS}}, r)
\end{aligned}
$$

◀ PROOF

The lemma is proven using Theorem 3.5 which allows us to switch from the Hoare-triple-style formulation to the view of operational semantics. The SS Hoare triple is instantiated with the one concluded in Lemma 6.64. ■

Transfer of the page-fault handler correctness from BS to SS. First we define a predicate similar to Definition 6.28 which helps us to evaluate parameters of the call to the page-fault handler.

◀ DEFINITION 6.66
Evaluation of parameters to the page-fault handler at the level of SS

$$
\begin{aligned}
&param^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, te, e_{pid}, e_{addr}, e_{intent}, e_{count}, pid, addr, intent, count) = \\
&\exists ct_{pid}, ct_{addr}, ct_{intent}, ct_{count} : \\
&eval_{\mathrm{SS}}(te, c_{\mathrm{SS}}.mem, e_{pid}) = \lfloor ct_{pid} \rfloor \wedge ct_{pid}(0) = mcell_{\mathrm{nat}}(pid) \\
\wedge\ &eval_{\mathrm{SS}}(te, c_{\mathrm{SS}}.mem, e_{addr}) = \lfloor ct_{addr} \rfloor \wedge ct_{addr}(0) = mcell_{\mathrm{nat}}(addr) \\
\wedge\ &eval_{\mathrm{SS}}(te, c_{\mathrm{SS}}.mem, e_{intent}) = \lfloor ct_{intent} \rfloor \wedge ct_{intent}(0) = mcell_{\mathrm{nat}}(intent) \\
\wedge\ &eval_{\mathrm{SS}}(te, c_{\mathrm{SS}}.mem, e_{count}) = \lfloor ct_{count} \rfloor \wedge ct_{count}(0) = mcell_{\mathrm{nat}}(count)
\end{aligned}
$$

Modification of variables by the page-fault handler is described by the predicate $modifies^{\mathrm{SS}}_{\mathrm{ta}}?$ which states that (i) the size of the heap memory frame remains unchanged, (ii) all global variables of the concrete kernel except those that are allowed to be modified by the page-fault handler remain the same, (iii) all initialized top-local variables except for r stay the same, and (iv) all heap variable which lie after the page table space and `USER_PGS` page descriptors remain unchanged.

◀ DEFINITION 6.67
Modification of SS variables by page-fault handler

$$
\begin{aligned}
&modifies^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, c'_{\mathrm{SS}}, r) = \\
&\quad |c'_{\mathrm{SS}}.mem.hm| = |c_{\mathrm{SS}}.mem.hm| \\
\wedge\ &(\forall v : v \in map(fst, gst_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \wedge v \notin modified_{\mathrm{ta}} \\
&\qquad \longrightarrow value_{\mathrm{g}}(c'_{\mathrm{SS}}.mem, gvar_{\mathrm{gm}}(v)) = value_{\mathrm{g}}(c_{\mathrm{SS}}.mem, gvar_{\mathrm{gm}}(v))) \\
\wedge\ &(\forall v : v \in map(fst, lst_{\mathrm{top}}(c_{\mathrm{SS}}.mem)) \wedge v \in lm_{\mathrm{top}}(c_{\mathrm{SS}}.mem).init \wedge v \neq r \\
&\qquad \longrightarrow value_{\mathrm{g}}(c'_{\mathrm{SS}}.mem, gvar_{\mathrm{lm}}(n, v)) = value_{\mathrm{g}}(c_{\mathrm{SS}}.mem, gvar_{\mathrm{lm}}(n, v))) \\
\wedge\ &(\forall\ \texttt{USER\_PGS} < i < |hst(c_{\mathrm{SS}}.mem)| : \\
&\qquad \longrightarrow value_{\mathrm{g}}(c'_{\mathrm{SS}}.mem, gvar_{\mathrm{hm}}(i)) = value_{\mathrm{g}}(c_{\mathrm{SS}}.mem, gvar_{\mathrm{hm}}(i)))
\end{aligned}
$$

Above, $n = |c_{\mathrm{SS}}.mem.lm| - 1$.

The SS Hoare triple's precondition in the lemma below is a set of pairs of SS and extended states $(c_{\mathrm{SS}}, c_{\mathrm{X}})$ for which (i) the global symbol table of c_{SS} coincides with the symbol table of the concrete kernel, (ii) the SS heap structure of the demand paging is respected by c_{SS}, (iii) the parameters of the call are appropriately evaluated, (iv) the SS precondition of the page-fault handler $PRE^{\mathrm{SS}}_{\mathrm{ta}}?$ holds as well as the precondition over the extended state $PRE^{\mathrm{X}}_{\mathrm{ta}}?$.

The Hoare triple's postcondition is stated as a set of pairs of small-step configurations c'_{SS} and extended states c'_{X} for which (i) the SS postcondition $POST^{\mathrm{SS}}_{\mathrm{ta}}?$ of the page-fault handler is satisfied as well as the extended state postcondition $POST^{\mathrm{X}}_{\mathrm{ta}}?$, (ii) the result of the page-fault handler is appropriately computed, and (iii) modifications over the SS memory are described by the predicate $modifies^{\mathrm{SS}}_{\mathrm{ta}}?$.

LEMMA 6.68 ▶
Page-Fault handler correctness at the level of SS

$$
\begin{array}{l}
\mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \wedge r \in L \longrightarrow \\
\Pi_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{xsem}_{\mathrm{PFH}}, L \vDash^{\mathrm{t}}_{\mathrm{SS}} \\
\quad \{(c_{\mathrm{SS}}, c_{\mathrm{X}}) \mid c_{\mathrm{SS}} = c^0_{\mathrm{SS}} \wedge c_{\mathrm{X}} = c^0_{\mathrm{X}} \\
\qquad \wedge \mathit{gst}(c_{\mathrm{SS}}.\mathit{mem}) = \mathit{gst}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}) \wedge \mathit{prefix\text{-}hst}_{\mathrm{CVM}}?(c_{\mathrm{SS}}.\mathit{mem}) \\
\qquad \wedge \mathit{param}^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, \mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), e_{\mathit{pid}}, e_{\mathit{addr}}, e_{\mathit{intent}}, e_{\mathit{count}}, \\
\qquad\qquad\qquad \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) \\
\qquad \wedge \mathit{PRE}^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \\
\qquad\qquad \mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) \wedge \mathit{PRE}^{\mathrm{X}}_{\mathrm{ta}}?(c_{\mathrm{X}})\} \\
\\
\qquad \mathit{SCall}(r, \texttt{pfh\_touch\_addr}, [e_{\mathit{pid}}, e_{\mathit{addr}}, e_{\mathit{intent}}, e_{\mathit{count}}]) \\
\\
\quad \{(c'_{\mathrm{SS}}, c'_{\mathrm{X}}) \mid \mathit{POST}^{\mathrm{SS}}_{\mathrm{ta}}?(c'_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \\
\qquad\qquad \mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) \wedge \mathit{POST}^{\mathrm{X}}_{\mathrm{ta}}?(c^0_{\mathrm{X}}, c'_{\mathrm{X}}) \\
\qquad \wedge r = \mathit{pfh\text{-}ta\text{-}res}_{\mathrm{SS}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) \\
\qquad \wedge \mathit{modifies}^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, c'_{\mathrm{SS}}, r)\}
\end{array}
$$

PROOF ▶ We apply transfer theorem from BS to SS (Theorem 3.4) in which the big-step Hoare triple is instantiated with the one concluded in the correctness lemma of the page-fault at the BS level with respect to the extended program (Lemma 6.38). The major proof effort — implications from SS to BS preconditions and from BS to SS postconditions — are concluded using Lemmas 6.60 and 6.62. ■

Ultimately, we reformulate the above lemma in the style of operational semantics. The additional terms reveled by that are the same as in Lemma 6.65.

LEMMA 6.69 ▶
Page-fault handler correctness at the level of SS: operational-semantic style

$$
\begin{array}{l}
\quad \mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \wedge c_{\mathrm{SS}} \in \mathit{valid}_{\mathrm{SS}}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})) \\
\wedge\ c_{\mathrm{SS}}.\mathit{prog} = \mathit{SCall}(r, \texttt{pfh\_touch\_addr}, [\mathit{e\text{-}pid}, \mathit{e\text{-}addr}, \mathit{e\text{-}intent}, \mathit{e\text{-}count}]) \\
\wedge\ \mathit{gst}(c_{\mathrm{SS}}.\mathit{mem}) = \mathit{gst}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}) \wedge \mathit{prefix\text{-}hst}_{\mathrm{CVM}}?(c_{\mathrm{SS}}.\mathit{mem}) \\
\wedge\ \mathit{param}^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, \mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{e\text{-}pid}, \mathit{e\text{-}addr}, \mathit{e\text{-}intent}, \mathit{e\text{-}count}, \\
\qquad \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) \\
\wedge\ \mathit{PRE}^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, c_{\mathrm{X}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \\
\qquad \mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) \wedge \mathit{PRE}^{\mathrm{X}}_{\mathrm{ta}}?(c_{\mathrm{X}}) \\
\wedge\ r \in \mathit{map}(\mathit{fst}, \mathit{lm}_{\mathrm{top}}(c_{\mathrm{SS}}.\mathit{mem})) \\
\longrightarrow \exists n, c'_{\mathrm{SS}}, c'_{\mathrm{X}} : \delta^n_{\mathrm{SSX}}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), \mathit{xsem}, c_{\mathrm{SS}}, c_{\mathrm{X}}) = \lfloor c'_{\mathrm{SS}}, c'_{\mathrm{X}} \rfloor \\
\qquad \wedge\ c'_{\mathrm{SS}}.\mathit{prog} = \mathit{Skip} \wedge c'_{\mathrm{SS}} \in \mathit{valid}_{\mathrm{SS}}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{ext}}(\Pi_{\mathrm{AK}})) \\
\qquad \wedge\ \mathit{trans\text{-}inv}?(c_{\mathrm{SS}}.\mathit{prog}, [\,], c_{\mathrm{SS}}, c'_{\mathrm{SS}}) \\
\qquad \wedge\ \mathit{POST}^{\mathrm{SS}}_{\mathrm{ta}}?(c'_{\mathrm{SS}}, c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}, \\
\qquad\qquad \mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{locs}_{\mathrm{active}}, \mathit{locs}_{\mathrm{free}}) \wedge \mathit{POST}^{\mathrm{X}}_{\mathrm{ta}}?(c_{\mathrm{X}}, c'_{\mathrm{X}}) \\
\qquad \wedge\ r = \mathit{pfh\text{-}ta\text{-}res}_{\mathrm{SS}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) \\
\qquad \wedge\ \mathit{modifies}^{\mathrm{SS}}_{\mathrm{ta}}?(c_{\mathrm{SS}}, c'_{\mathrm{SS}}, r)
\end{array}
$$

PROOF ▶ We apply Theorem 3.5 where the SS Hoare triple is instantiated by the one concluded in Lemma 6.68. ■

CHAPTER 7

Integrating Results

In this chapter we prove top-level correctness theorems of demand paging: correctness of the initialization code and the page-fault handler. Up to this point we have shown correctness of the demand paging functions at the level of extended C0 small-step semantics. We still have unjustified XCalls to the hard-disk drivers. We plug in results from Alkassar's doctoral thesis [Alk09] in order to close this gap. First, we prove a theorem justifying correctness of the extended calls towards their ISA implementations and show how we apply this theorem in the context of top-level correctness theorems of demand paging. Next, we show the top-level correctness theorems paying attention on the proofs of the CVM relation for user processes as the latter essentially defines correctness of memory virtualization. Finally, we elaborate how the results achieved in this thesis are used in the CVM correctness proof.

7.1 Hard-Disk Drivers and Zero Fill Page

In the end of the previous chapter we obtained computations of demand paging functions at the level of C0 small-step semantics. Note that these computations (see Lemmas 6.65 and 6.69) are defined by means of the C0 transition function δ^n_{SSX} which is capable of treating extended calls. As we described in Section 6.3 we replaced all calls to the hard-disk driver functions `read_from_disk` and `write_to_disk` as well as the function `zero_fill_page` by corresponding XCalls. We also mentioned that the effects of these functions are expressed in an extended semantics environment.

However, the correctness theorem of CVM (Theorem 2.33) is formulated on the levels of VAMP ISA with devices and C0 small-step semantics without XCalls support. The ultimate goal of the demand paging correctness theorems is to be used in the CVM correctness proof. To achieve this we need to get rid of the mentioned XCalls by justifying their correctness: the extended semantics has to be respected by their implementations which contain inline assembly code.

The means to deal with the described problem are provided by Alkassar. In his thesis [Alk09] he proves a compiler correctness theorem with support of XCalls for the mentioned functions [Alk09, Theorem 11] for the case of `write_to_disk`. The remaining cases could be handled completely analogously. The target level of that theorem is VAMP assembly. However, we aim at formulating correctness of the demand paging in VAMP ISA semantics. For that, below we transfer the Alkassar's theorem down to the VAMP ISA level.

The definitions in this section are a product of collaboration between Alkassar and the author.

7.1.1 Extended Semantics Environment

Hard-disk drivers. The functions `write_to_disk` and `read_from_disk` are implemented as C0 functions but their bodies consist only of a single assembly portion. In Section 4.4 we specified the behavior of these function on the abstract level (Definitions 4.28 and 4.26). Now we consider under which preconditions their behavior occurs and specify the results of the function in small-step semantics.

Both functions take the start (page) address in the physical memory ma and the start (sector) address in the swap memory sa as parameters. They copy the specified page between the memories. The precondition to the functions restricts their parameters: (i) the memory address is page aligned; (ii) the page to be copied does not overlap with the devices memory region and the system memory part, and (iii) lies within the virtual address space; (iv) the disk sector address is page aligned, i.e., divisible by eight, and (v) lies outside the boot region of the hard disk.

DEFINITION 7.1 ► Preconditions to the hard-disk driver

$$
\begin{aligned}
PRE_{\mathrm{driver}}?(ma, sa) \quad = \quad & ma \bmod \texttt{PG\_SZ} = 0 \\
\wedge \quad & ma + \texttt{PG\_SZ} \leq \texttt{DEVICES\_BORDER} \\
\wedge \quad & \texttt{ZFP}\cdot\texttt{PG\_SZ} \leq ma \\
\wedge \quad & ma < \texttt{ZFP}\cdot\texttt{PG\_SZ} + (\texttt{USER\_PGS}+1)\cdot\texttt{PG\_SZ} \\
\wedge \quad & sa \bmod 8 = 0 \\
\wedge \quad & \texttt{BOOT\_PGS}\cdot 8 \leq sa \\
\wedge \quad & sa < \texttt{BOOT\_PGS}\cdot 8 + \texttt{TOT\_BIG\_PGS}\cdot\texttt{PGS\_PER\_BIG\_PG}\cdot 8
\end{aligned}
$$

The extended semantics of the functions takes as arguments a mapping from parameter names to their contents $args :: \mathbb{S} \mapsto (\mathbb{N} \mapsto mcell\text{-}t)_\perp$ and an extended state c_X. If the parameters fulfill the preconditions to the hard-disk drivers the returned results comprises the value T and an updated extended state obtained via functions *write-to-disk* respectively *read-from-disk* (Definitions 4.28 and 4.26). Otherwise, the computation gets stuck.

◀ DEFINITION 7.2
Extended semantics of write to disk

$$xsem_{\mathrm{write}}(args, c_\mathrm{X}) = \begin{cases} \lfloor([mcell_{\mathrm{bool}}(\mathsf{T})], write\text{-}to\text{-}disk(ma, sa, c_\mathrm{X}))\rfloor & \text{if } PRE_{\mathrm{driver}}?(ma, sa) \\ \perp & \text{otherwise} \end{cases}$$

$$\text{where } ma = m2u(\llbracket args(\texttt{mem\_addr})\rrbracket(0)),\quad sa = m2u(\llbracket args(\texttt{disk\_addr})\rrbracket(0)).$$

◀ DEFINITION 7.3
Extended semantics of read from disk

$$xsem_{\mathrm{read}}(args, c_\mathrm{X}) = \begin{cases} \lfloor([mcell_{\mathrm{bool}}(\mathsf{T})], read\text{-}from\text{-}disk(ma, sa, c_\mathrm{X}))\rfloor & \text{if } PRE_{\mathrm{driver}}?(ma, sa) \\ \perp & \text{otherwise} \end{cases}$$

$$\text{where } ma = m2u(\llbracket args(\texttt{mem\_addr})\rrbracket(0)),\quad sa = m2u(\llbracket args(\texttt{disk\_addr})\rrbracket(0)).$$

Above, $m2u(m)$ converts a memory cell to an unsigned integer.

Zero fill page. The body of the function `zero_fill_page` also consists only of assembly instructions. The function's parameter is an index *ppx* of the page to be filled with zeros. The preconditions to the function ensure that only the page at address `ZFP` of some user page might be accessed.

◀ DEFINITION 7.4
Preconditions to zero fill page

$$PRE_{\mathrm{zfp}}?(ppx) = \texttt{ZFP} \leq ppx < \texttt{ZFP} + \texttt{USER\_PGS} + 1$$

The extended semantics of the function updates the extended state via the function *zero-fill-page* (Definition 4.30) and returns zero as a result in case the preconditions are satisfied. Otherwise, the computation gets stuck.

◀ DEFINITION 7.5
Extended semantics of zero fill page

$$xsem_{\mathrm{zfp}}(args, c_\mathrm{X}) = \begin{cases} \lfloor([mcell_{\mathrm{int}}(0)], zero\text{-}fill\text{-}page(ppx, c_\mathrm{X}))\rfloor & \text{if } PRE_{\mathrm{zfp}}?(ppx) \\ \perp & \text{otherwise} \end{cases}$$

$$\text{where } ppx = m2u(\llbracket args(\texttt{ppx})\rrbracket(0)).$$

Altogether. The extended semantics environment for the demand paging $xsem_{\mathrm{PFH}}$ is a list with three elements. Each element is pair of a function name and a list of function's parameters together with its return type and the extended semantics.

◀ DEFINITION 7.6
Extended semantics environemnt for demand paging

$$\begin{aligned} &xsem_{\mathrm{PFH}} = \\ &\ [(\texttt{write\_to\_disk}, \\ &\quad([(\texttt{mem\_addr}, UnsgndT), (\texttt{disk\_addr}, UnsgndT)], [BooleanT], xsem_{\mathrm{write}})), \\ &\ (\texttt{read\_from\_disk}, \\ &\quad([(\texttt{mem\_addr}, UnsgndT), (\texttt{disk\_addr}, UnsgndT)], [BooleanT], xsem_{\mathrm{read}})), \\ &\ (\texttt{zero\_fill\_page}, ([(\texttt{ppx}, UnsgndT)], [IntegerT], xsem_{\mathrm{zfp}}))] \end{aligned}$$

7.1.2 Correctness of the Extended Calls

Mapping the extended state to the physical and swap memories. As our current goal is to get rid of XCalls, i.e., justify them by applying functional correctness of their implementations, we also have to get rid of the extended state by mapping it to the physical and the swap memories of the underlying ISA or assembly machine with devices. By that we will be able to project modifications described by the extended semantics over the extended state to real hardware memories.

The relation for the physical memory $xconsis_{\text{mem}}?$ is defined for both VAMP ISA c_{ISA} and assembly c_{ASM} configurations (we use the same function name). The first parameter of the relation is one of these configurations whereas the second parameter is the extended state of demand paging c_{X}. The relation states that each word taken from the physical memory component of c_{X} equals to the corresponding word read from the physical memory.

DEFINITION 7.7 ► Relation between physical memory and extended state

$$xconsis_{\text{mem}}?(c_{\text{ASM}}, c_{\text{X}}) = \forall\, k < |c_{\text{X}}.mem| :\ \forall\, i < \texttt{PG\_SZ\_WD} : \\ i2n(c_{\text{ASM}}.m(\texttt{PG\_SZ} \cdot (\texttt{ZFP} + k)/4 + i)) = c_{\text{X}}.mem[k][i]$$

$$xconsis_{\text{mem}}?(c_{\text{ISA}}, c_{\text{X}}) = \forall\, k < |c_{\text{X}}.mem| :\ \forall\, i < \texttt{PG\_SZ\_WD} : \\ \langle c_{\text{ISA}}.m_{\text{word}}(bin(\texttt{PG\_SZ} \cdot (\texttt{ZFP} + k) + 4 \cdot i))\rangle = c_{\text{X}}.mem[k][i]$$

The relation for the swap memory $xconsis_{\text{swap}}?$ is defined between a configuration of the hard disk c_{HD} and extended state c_{X}. It states that each word taken from the swap memory component of c_{X} is equal to the respective word read from the disk's swap memory.

DEFINITION 7.8 ► Relation between swap memory and extended state

$$xconsis_{\text{swap}}?(c_{\text{HD}}, c_{\text{X}}) = \forall\, k < |c_{\text{X}}.swap| :\ \forall\, i < \texttt{PG\_SZ\_WD} : \\ c_{\text{HD}}.sm[\texttt{PG\_SZ\_WD} \cdot (\texttt{BOOT\_PGS} + k) + i] = c_{\text{X}}.swap[k][i]$$

Adding relation for C0 states. We obtain a single *extended consistency relation* by combining relations $xconsis_{\text{mem}}?$ and $xconsis_{\text{swap}}?$ with the relation between a C0 configuration c_{SS} of the concrete kernel and its version c_{XSS} from which the extended calls to the hard-disk drivers and the zero fill page function are removed. The memories of these C0 machines are equal. The differences in their program rests are specified by the function $repl\text{-}hdzfp_{\text{stmt}}$ introduced in Section 6.3 (Definition 6.31).

DEFINITION 7.9 ► Extended consistency relation

$$xconsis?(c_{\text{SS}}, c_{\text{ISA}}, c_{\text{HD}}, c_{\text{XSS}}, c_{\text{X}}) = \\ \begin{aligned} & c_{\text{XSS}}.mem = c_{\text{SS}}.mem \\ \wedge\ & c_{\text{XSS}}.prog = repl\text{-}hdzfp_{\text{stmt}}(hd(s2l(c_{\text{SS}}.prog))) \\ \wedge\ & xconsis_{\text{mem}}?(c_{\text{ISA}}, c_{\text{X}}) \\ \wedge\ & xconsis_{\text{swap}}?(c_{\text{HD}}, c_{\text{X}}) \end{aligned}$$

Note that in this definition an assembly configuration c_{ASM} could used be instead of c_{ISA}.

Relation between function tables and extended semantics environment. The extended consistency relations ties together a complete C0 configuration of the concrete kernel and its variant with XCalls removed. We also need to express a relation between the function tables corresponding to these configurations.

The desired relation takes two functions tables *ft* and *xpt*, and some extended semantics environment *xsem*. It states that (i) *xsem* equals the extended semantics for demand paging $xsem_{\mathrm{PFH}}$, (ii) *xpt* is obtained from *ft* by removing the definitions of hard-disk drivers and zero fill page, and (iii) these functions are defined in the function table *ft*.

◀ DEFINITION 7.10
Relation between function tables and extended semantics

$$\begin{aligned} xconsis_{\mathrm{ft}}?(ft, xpt, xsem) \quad &= \quad xsem = xsem_{\mathrm{PFH}} \\ &\wedge \quad xpt = repl\text{-}hdzfp_{\mathrm{ft}}(ft) \\ &\wedge \quad renum^{\mathrm{even}}_{\mathrm{fun}}(\texttt{write\_to\_disk}, f_{\mathrm{write\_to\_disk}}) \in ft \\ &\wedge \quad renum^{\mathrm{even}}_{\mathrm{fun}}(\texttt{read\_from\_disk}, f_{\mathrm{read\_from\_disk}}) \in ft \\ &\wedge \quad renum^{\mathrm{even}}_{\mathrm{fun}}(\texttt{zero\_fill\_page}, f_{\mathrm{zero\_fill\_page}}) \in ft \end{aligned}$$

Above, f with subscripts corresponds to function definitions in a function table.

Next, we present two theorems justifying correctness of the extended calls. The target models of these theorems are VAMP assembly and ISA, respectively. The theorem for the assembly is an adaptation of Theorem 11 from Alkassar's thesis [Alk09] in order to make it more suitable for further transfer down to the ISA level. We use Alkassar's theorem to prove the theorem for assembly. The latter together with the simulation theorem of VAMP assembly by VAMP ISA [Tsy09, Theorem 4.8] is used to prove the theorem at the ISA level.

Note that below we speak about correctness of a single function, like the page-fault handler, and therefore the program rest consists of a single call statement to the respective function.

XCalls correctness towards VAMP assembly. The purpose of the theorem below is to justify correctness of XCalls occurring in a given extended C0 computation, i.e., a computation defined by the transition function δ^n_{SSX} between some start pair of C0 and extended start configurations $(c_{\mathrm{XSS}}, c_{\mathrm{X}})$ and the resulting states $(c'_{\mathrm{XSS}}, c'_{\mathrm{X}})$. The given extended C0 computation describes a function call, e.g., a call to the page-fault handler. The extended consistency relation between the extended C0 state c_{XSS} and a normal C0 configuration c_{SS} is assumed to hold at the beginning. The theorem delivers a resulting normal C0 configuration c'_{SS} under preservation of the extended consistency relation. Besides that, we state that the underlying assembly computation terminates.

Let us classify the theorem's assumptions and conclusions in detail. The theorem below assumes the following.

- Validity and size constraints of the involved models/concepts: C0 configuration and program (1,2), assembly configuration (7), execution sequence (9), hard-disk disk (10), and extended state (4).

- (Abstraction) relations between the mentioned concepts: simulation of assembly by C0 (3), extended consistency relation (5), and function table and extended semantics (6).

- Preconditions to executions: on the assembly level (8), and on the extended C0 level. The latter comprises absence of address-of operators and assembly and external call statements inside the program part to be executed (13) and dynamic (14) and static memory consumption constraints.

- Successful extended C0 execution: the result configuration is obtained (11), the call statement is removed from the program rest (12), and the heap borders are not violated (15,16).

On the side of conclusions the theorem provides a resulting normal C0 configuration, an updated C0 allocation function, a number of taken assembly steps, and an updated VAMP assembly with devices configuration, such that the following holds.

- The underlying assembly computation succeeds (17) and there is no communication with device except for the swap hard-disk (26). The latter is denoted by the predicate *non-interf-dev'*? which is defined by the equation from Definition 2.15 where the device with the identifier `SWAP_DID` is excluded from the universal quantification.

- The validity constraints are preserved over the model of C0 (18), assembly (21), and hard-disk (23).

- The consistency relation between the C0 and assembly is preserved (19) as well as the extended consistency relation (20).

- The assembly execution proceeds in the system mode (22) and absence of interrupts is guaranteed (25).

- The assembly memory remains unchanged at the two first addresses where the code of a jump-to-kernel resides (24).

THEOREM 7.11 ▶
Correctness of XCalls towards assembly level

$$
\begin{array}{lll}
(1) & & c_{\mathrm{SS}} \in C0'\surd(te, ft) \\
(2) & \wedge & (te, ft, gst(c_{\mathrm{SS}}.mem)) \in xltbl_{\mathrm{prog}} \\
(3) & \wedge & consis?(te, ft, c_{\mathrm{SS}}, alloc, c_{\mathrm{ASM+DS}}.cpu) \\
(4) & \wedge & subtyping_{\mathrm{X}}?(c_{\mathrm{X}}) \\
(5) & \wedge & xconsis?(c_{\mathrm{SS}}, c_{\mathrm{ASM+DS}}.cpu, the\text{-}hd(c_{\mathrm{ASM+DS}}.devs(\texttt{SWAP\_DID})), c_{\mathrm{XSS}}, c_{\mathrm{X}}) \\
(6) & \wedge & xconsis_{\mathrm{ft}}?(ft, xpt, xsem) \\
(7) & \wedge & asm\surd(c_{\mathrm{ASM+DS}}.cpu) \\
(8) & \wedge & sys\text{-}exec_{\mathrm{ASM}}?(c_{\mathrm{ASM+DS}}.cpu) \\
(9) & \wedge & seq\surd(seq, c_{\mathrm{ASM+DS}}.devs) \\
(10) & \wedge & hd\surd(c_{\mathrm{ASM+DS}}.devs) \\
(11) & \wedge & \delta_{\mathrm{SSX}}^{xc\text{-}steps}(te, xpt, c_{\mathrm{XSS}}, c_{\mathrm{X}}, xsem) = \lfloor (c'_{\mathrm{XSS}}, c'_{\mathrm{X}}) \rfloor \\
(12) & \wedge & is\text{-}SCall(c_{\mathrm{XSS}}.prog) \ \wedge \ is\text{-}Skip(c'_{\mathrm{XSS}}.prog) \\
(13) & \wedge & \forall\, p \in SCalls(xpt, called\text{-}func(c_{\mathrm{XSS}}.prog)) : \\
 & & \qquad noAddrOf\text{-}Asm\text{-}ESCall?(\llbracket map\text{-}of(xpt, p) \rrbracket.body) \\
(14) & \wedge & (ft, called\text{-}func(c_{\mathrm{XSS}}.prog), sz) \in SE \\
(15) & \wedge & \texttt{ABASE}_{\mathrm{lm}} + asize_{\mathrm{st^*}}(sc(c_{\mathrm{SS}}.mem).lst) + sz \leq \texttt{HEAP\_START} \\
(16) & \wedge & \texttt{HEAP\_START} + asize_{\mathrm{heap}}(hst(c'_{\mathrm{XSS}}.mem)) < \texttt{ZFP} \cdot \texttt{PG\_SZ} \\
 & \longrightarrow & \exists\, c'_{\mathrm{SS}}, alloc', asm\text{-}steps, c'_{\mathrm{ASM+DS}} :
\end{array}
$$

$$
\begin{array}{lll}
(17) & & \delta^{asm\text{-}steps}_{\mathrm{ASM+DS}}(seq, c_{\mathrm{ASM+DS}}) = c'_{\mathrm{ASM+DS}} \\
(18) & \wedge & c'_{\mathrm{SS}} \in C0'\surd(te, ft) \\
(19) & \wedge & consis?(te, ft, c'_{\mathrm{SS}}, alloc', c'_{\mathrm{ASM+DS}}.cpu) \\
(20) & \wedge & xconsis?(c'_{\mathrm{SS}}, c'_{\mathrm{ASM+DS}}.cpu, \\
(21) & & \qquad the\text{-}hd(c'_{\mathrm{ASM+DS}}.devs(\texttt{SWAP\_DID})), c'_{\mathrm{XSS}}, c'_{\mathrm{X}}) \\
(22) & \wedge & asm\surd(c'_{\mathrm{ASM+DS}}.cpu) \\
(23) & \wedge & sys\text{-}exec_{\mathrm{ASM}}?(c'_{\mathrm{ASM+DS}}.cpu) \\
(24) & \wedge & hd\surd(c'_{\mathrm{ASM+DS}}.devs) \\
(25) & \wedge & \forall\, i:\ 0 \le i \wedge i < 2 \longrightarrow c'_{\mathrm{ASM+DS}}.cpu.m(i) = c_{\mathrm{ASM+DS}}.cpu.m(i) \\
(26) & \wedge & dyn\text{-}prop_{\mathrm{DS}}(c_{\mathrm{ASM+DS}}, \texttt{PROGBASE}, \\
 & & \qquad csize_{\mathrm{prog}}(te, gst(c_{\mathrm{SS}}.mem), ft), seq, asm\text{-}steps) \\
 & \wedge & non\text{-}interf\text{-}dev'?(c_{\mathrm{ASM+DS}}.devs, c'_{\mathrm{ASM+DS}}.devs, seq, asm\text{-}steps)
\end{array}
$$

Let us clarify some definitions used in the theorem. The function $asize_{\mathrm{st}^*}$ is used to compute the size of the local memory stack. This function is defined as the sum of symbol table sizes:

$$asize_{\mathrm{st}^*}(lsts) = \sum_{i=0}^{i<|lsts|} asize_{\mathrm{st}}(lsts[i]).$$

The allocated size of a symbol table *st* is computed by the function $asize_{\mathrm{st}}(st)$ [Lei07, Definition 7.11]. The function $asize_{\mathrm{heap}}(hst)$ [Lei07, Definition 7.12] computes the allocated size of the heap for a heap symbol table *hst*. Condition (14) $(ft, called\text{-}func(c_{\mathrm{SS}}.prog), sz) \in SE$ [Tsy09, Definition 7.21] is used to estimate the stack consumption and means that starting from a call of the function $called\text{-}func(c_{\mathrm{SS}}.prog)$ the execution with respect to the function table *ft* consumes not more than *sz* bytes for the stack.

XCalls correctness towards VAMP ISA. The formulation of theorem with VAMP ISA as a target level literally follows the statement of the theorem above. We use an ISA with devices configuration instead of assembly. Therefore necessary validity constraints have to hold at the ISA level. Instead of consistency relation between C0 and assembly a simulation relation between C0 and ISA is used. Below we state the desired theorem omitting the repeating details.

◀ THEOREM 7.12
Correctness of XCalls towards ISA level

$$
\begin{array}{lll}
 & & \ldots \\
(3) & \wedge & C0\text{-}sim\text{-}isa?(te, ft, c_{\mathrm{SS}}, c_{\mathrm{ISA+DS}}.cpu) \ \wedge \ \cdots \\
(5) & \wedge & xconsis?(c_{\mathrm{SS}}, c_{\mathrm{ISA+DS}}.cpu, \\
 & & \qquad the\text{-}hd(c_{\mathrm{ISA+DS}}.devs(\texttt{SWAP\_DID})), c_{\mathrm{XSS}}, c_{\mathrm{X}}) \ \wedge \ \cdots \\
(7) & \wedge & isa\surd(c_{\mathrm{ISA+DS}}.cpu) \\
(8) & \wedge & sys\text{-}exec_{\mathrm{ISA}}?(c_{\mathrm{ISA+DS}}.cpu) \\
(9) & \wedge & seq\surd(seq, c_{\mathrm{ISA+DS}}.devs) \\
(10) & \wedge & hd\surd(c_{\mathrm{ISA+DS}}.devs) \ \wedge \ \cdots \\
 & \longrightarrow & \exists\, c'_{\mathrm{SS}}, isa\text{-}steps, c'_{\mathrm{ISA+DS}}: \\
(17) & & \quad \delta^{isa\text{-}steps}_{\mathrm{ISA+DS}}(seq, c_{\mathrm{ISA+DS}}) = c'_{\mathrm{ISA+DS}} \ \wedge \ \cdots \\
(19) & & \wedge \ C0\text{-}sim\text{-}isa?(te, ft, c'_{\mathrm{SS}}, c'_{\mathrm{ISA+DS}}.cpu) \\
(20) & & \wedge \ xconsis?(c'_{\mathrm{SS}}, c'_{\mathrm{ISA+DS}}.cpu, \\
 & & \qquad the\text{-}hd(c'_{\mathrm{ISA+DS}}.devs(\texttt{SWAP\_DID})), c'_{\mathrm{XSS}}, c'_{\mathrm{X}}) \\
(21) & & \wedge \ isa\surd(c'_{\mathrm{ISA+DS}}.cpu) \\
(22) & & \wedge \ sys\text{-}exec_{\mathrm{ISA}}?(c'_{\mathrm{ISA+DS}}.cpu) \\
(23) & & \wedge \ hd\surd(c'_{\mathrm{ISA+DS}}.devs) \\
(24) & & \wedge \ c'_{\mathrm{ISA+DS}}.cpu.m(rep(0, 29)) = c_{\mathrm{ISA+DS}}.cpu.m(rep(29, 0)) \\
(26) & & \wedge \ non\text{-}interf\text{-}dev'?(c_{\mathrm{ISA+DS}}.devs, c'_{\mathrm{ISA+DS}}.devs, seq, isa\text{-}steps)
\end{array}
$$

7.1.3 Applying Correctness of the Extended Calls

We will apply Theorem 7.12 twice: for the initialization code and for the page-fault handler. The first function has an XCall to the zero fill page whereas the second function additionally uses hard-disk drivers. The theorem application procedure is the same for these two cases. Because of that, we describe how the XCalls correctness theorem is applied before presenting the actual statements of top-level correctness theorems for the initialization code and the page-fault handler.

So far, we have transferred correctness results of the demand paging implementation down to the extended C0 level. Lemmas 6.65 and 6.69 give us extended C0 computations:

$$\delta_{\mathrm{SSX}}^{xc\text{-}steps}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), ft_{\mathrm{ext}}(\Pi_{\mathrm{AK}}), c_{\mathrm{XSS}}, c_{\mathrm{X}}, xsem) = \lfloor (c'_{\mathrm{XSS}}, c'_{\mathrm{X}}) \rfloor.$$

Moreover, these configurations respect functional pre- and postconditions of the demand paging.

On the other hand, demand paging top-level correctness theorems are supposed to be applied during concrete kernel executions within CVM. That is why they necessarily speak about some concrete kernel C0 configuration c_{SS} which respects the following properties:

$$\begin{array}{l} \mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}), \\ \mathit{gst}(c_{\mathrm{SS}}.\mathit{mem}) = \mathit{gst}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}). \end{array}$$

To apply Theorem 7.12 we instantiate the program with the linked kernel program, i.e., $te = te_{\mathrm{CK}}(\Pi_{\mathrm{AK}})$ and $ft = ft_{\mathrm{CK}}(\Pi_{\mathrm{AK}})$. We instantiate the extended C0 configuration with the configuration obtained from the normal C0 configuration by replacing the program rest by its first statement. This is handled by the function $\mathit{ss2xss}(c_{\mathrm{SS}}) = c_{\mathrm{XSS}}$:

$$\begin{array}{rcl} c_{\mathrm{XSS}}.\mathit{mem} & = & c_{\mathrm{SS}}.\mathit{mem}, \\ c_{\mathrm{XSS}}.\mathit{prog} & = & \mathit{hd}(\mathit{s2l}(c_{\mathrm{SS}}.\mathit{prog})). \end{array}$$

We construct the extended state from the physical memory and the hard-disk content of ISA with devices $c_{\mathrm{ISA+DS}}$. For this we define the function $\mathit{isa2x}(c_{\mathrm{ISA+DS}}) = c_{\mathrm{X}}$ which yields the extended state c_{X} by (i) converting the user part of the ISA memory to the physical memory component of the extended state with the function *mem2x* (cf. Figure 7.1), and (ii) converting the hard-disk swap memory part outside the boot region to the swap component of the extended state with the function *swap2x* (cf. Figure 7.2).

DEFINITION 7.13 ▶
Construction of extended state from ISA with devices

$$\begin{array}{rcl} c_{\mathrm{X}}.\mathit{mem} & = & \mathit{mem2x}(c_{\mathrm{ISA+DS}}.\mathit{cpu.m}) \\ c_{\mathrm{X}}.\mathit{swap} & = & \mathit{swap2x}(\mathit{the\text{-}hd}(c_{\mathrm{ISA+DS}}.\mathit{devs}(\mathtt{SWAP\_DID})).\mathit{sm}) \end{array}$$

$$\begin{array}{l} \mathit{mem2x}(m) = \mathit{map}(\lambda i:\ \mathit{map}(\lambda j:\ \langle m_{\mathrm{word}}(\mathit{bin}(\mathtt{PG\_SZ} \cdot (\mathtt{ZFP} + i) + 4 \cdot j))\rangle, \\ \qquad\qquad [0, \ldots, \mathtt{PG\_SZ\_WD} - 1]), \\ \qquad\qquad [0, \ldots, \mathtt{USER\_PGS}]) \\ \mathit{swap2x}(\mathit{sm}) = \mathit{map}(\lambda i:\ \mathit{map}(\lambda j:\ \mathit{sm}[\mathtt{PG\_SZ\_WD} \cdot (\mathtt{BOOT\_PGS} + i) + j], \\ \qquad\qquad [0, \ldots, \mathtt{PG\_SZ\_WD} - 1]), \\ \qquad\qquad [0, \ldots, \mathtt{TOT\_BIG\_PGS} \cdot \mathtt{PGS\_PER\_BIG\_PG} - 1]) \end{array}$$

The extended semantics environment is instantiated with the one for de-

Figure 7.1: Construction of the physical memory component

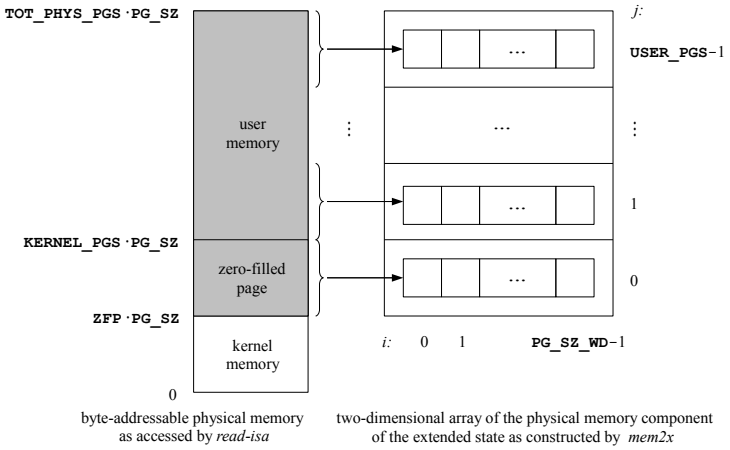
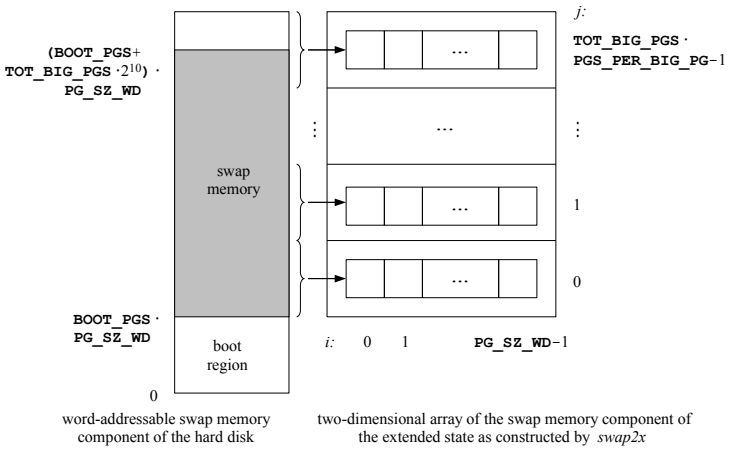


Figure 7.2: Construction of the swap memory component



mand paging $xsem_{\mathrm{PFH}}$. For memory consumption restrictions we compute the smallest number of bytes needed for the execution of each particular demand paging function. For the initialization code we set $sz = 52$ whereas for the the page-fault handler we use $sz = 136$.

Having these instantiations we justify the assumptions of Theorem 7.12 as follows.

Assumption 1: C0 configuration is valid. This is a CVM invariant.

Assumption 2: C0 program is translatable. Since our program is a concrete kernel obtained by linking we use Lemma 6.24 from Tsyban's thesis [Tsy09]:

$$\mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}}) \longrightarrow (te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), gst_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \in xltbl_{\mathrm{prog}}$$

Assumption 3: simulation relation between C0 and ISA configurations. This is a CVM invariant.

Assumption 4: sub-typing of the extended state. We need to show

$$\mathit{subtyping}_{\mathrm{X}}?(\mathit{isa2x}(c_{\mathrm{ISA+DS}})).$$

After unfolding Definition 4.8 we show four individual conjuncts. The length of the physical memory component is the user memory size plus one — for the zero filled page.

$$\begin{aligned} & |\mathit{mem2x}(c_{\mathrm{ISA+DS}}.\mathit{cpu.m})| \\ = \; & |\mathit{map}(\ldots, [0, \ldots, \texttt{USER\_PGS}])| \\ = \; & |[0, \ldots, \texttt{USER\_PGS}]| \\ = \; & \texttt{USER\_PGS} + 1 \end{aligned}$$

The length of the swap memory component is the number of pages contained in the total amount of big pages.

$$\begin{aligned} & |\mathit{swap2x}(\mathit{the\text{-}hd}(c_{\mathrm{ISA+DS}}.\mathit{devs}(\texttt{SWAP\_DID})).\mathit{sm})| \\ = \; & |\mathit{map}(\ldots, [0, \ldots, \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG} - 1])| \\ = \; & |[0, \ldots, \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG} - 1]| \\ = \; & \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG} \end{aligned}$$

Each element of the physical memory component is a list of PG_SZ_WD thirty-two-bit natural numbers.

$$\begin{aligned} \forall\, i < \texttt{USER\_PGS} + 1 : \quad & |\mathit{mem2x}(c_{\mathrm{ISA+DS}}.\mathit{cpu.m})[i]| \\ = \; & |\mathit{map}\ldots, [0, \ldots, \texttt{PG\_SZ\_WD} - 1]| \\ = \; & |[0, \ldots, \texttt{PG\_SZ\_WD} - 1]| \\ = \; & \texttt{PG\_SZ\_WD} \end{aligned}$$

The same is true for the swap memory component.

$$\begin{aligned} \forall\, i < \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG} : & \\ & |\mathit{swap2x}(\mathit{the\text{-}hd}(c_{\mathrm{ISA+DS}}.\mathit{devs}(\texttt{SWAP\_DID})).\mathit{sm})[i]| \\ = \; & |\mathit{map}(\ldots, [0, \ldots, \texttt{PG\_SZ\_WD} - 1])| \\ = \; & |[0, \ldots, \texttt{PG\_SZ\_WD} - 1]| \\ = \; & \texttt{PG\_SZ\_WD} \end{aligned}$$

Assumption 5: extended consistency relation. We need to prove

$$\begin{aligned} & \mathit{xconsis}?(c_{\mathrm{SS}}, c_{\mathrm{ISA+DS}}.\mathit{cpu}, \mathit{the\text{-}hd}(c_{\mathrm{ISA+DS}}.\mathit{devs}(\texttt{SWAP\_DID})), \\ & \quad \mathit{ss2xss}(c_{\mathrm{SS}}), \mathit{isa2x}(c_{\mathrm{ISA+DS}})). \end{aligned}$$

We unfold Definition 7.9 and conclude its individual conjuncts. The function *ss2xss* does not change C0 memory configurations.

$$\mathit{ss2xss}(c_{\mathrm{SS}}).\mathit{mem} \;=\; c_{\mathrm{SS}}.\mathit{mem}$$

The equation for the program rests holds because we handle a call to one of the demand paging functions.

$$\begin{aligned} & \mathit{ss2xss}(c_{\mathrm{SS}}).\mathit{prog} \\ = \; & \mathit{hd}(\mathit{s2l}(c_{\mathrm{SS}}.\mathit{prog})) \\ = \; & \mathit{repl\text{-}hdzfp}_{\mathrm{stmt}}(\mathit{hd}(\mathit{s2l}(c_{\mathrm{SS}}.\mathit{prog}))) \end{aligned}$$

For the physical memory component of the extended state using Definition 7.13 we have:

$$\begin{aligned}\forall\, k < \texttt{USER\_PGS} + 1 : \forall\, i < \texttt{PG\_SZ\_WD} :\\ & mem2x(c_{\text{ISA+DS}}.cpu.m)[k][i]\\ = \;& map(\ldots, map(\ldots, [0, \ldots, \texttt{USER\_PGS}]))[k][i]\\ = \;& map(\ldots(k), [0, \ldots, \texttt{PG\_SZ\_WD} - 1])[i]\\ = \;& \langle c_{\text{ISA+DS}}.cpu.m_{\text{word}}(bin(\texttt{PG\_SZ} \cdot (\texttt{ZFP} + k) + 4 \cdot i))\rangle\end{aligned}$$

Finally, for the swap memory component we conclude:

$$\begin{aligned}\forall\, k < \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG} : \forall\, i < \texttt{PG\_SZ\_WD} :\\ & swap2x(\textit{the-hd}(c_{\text{ISA+DS}}.devs(\texttt{SWAP\_DID})).sm)[k][i]\\ = \;& map(\ldots, map(\ldots, [0, \ldots, \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG} - 1]))[k][i]\\ = \;& map(\ldots(k), [0, \ldots, \texttt{PG\_SZ\_WD} - 1])[i]\\ = \;& \textit{the-hd}(c_{\text{ISA+DS}}.devs(\texttt{SWAP\_DID})).sm[\texttt{PG\_SZ\_WD} \cdot (\texttt{BOOT\_PGS} + k) + i].\end{aligned}$$

Assumption 6: relations between function tables and extended semantics. We need to show

$$xconsis_{\text{ft}}?(ft_{\text{CK}}(\Pi_{\text{AK}}), ft_{\text{ext}}(\Pi_{\text{AK}}), xsem_{\text{PFH}}).$$

Unfolding Definition 7.10 we see that the condition for the extended semantics of demand paging is trivial as well as the conjunct for the function tables (cf. Definition 6.33):

$$ft_{\text{ext}}(\Pi_{\text{AK}}) = \textit{repl-hdzfp}_{\text{ft}}(ft_{\text{CK}}(\Pi_{\text{AK}})).$$

For the last conjunct we use the fact that if the function does not contain any call statements then:

$$(fn, f) \in ft_{\text{CVM}} \longrightarrow renum^{\text{even}}_{\text{fun}}(fn, f) \in \{ft_{\text{CK}}(\Pi_{\text{AK}})\}.$$

This is true for $fn \in \{\texttt{write\_to\_disk}, \texttt{read\_from\_disk}, \texttt{zero\_fill\_page}\}$.

Assumptions 7-10: validity of models. These are CVM invariants.

Assumptions 11,12: execution on the extended C0 level. Delivered by correctness lemmas of demand paging at the small-step semantics level (Lemma 6.65 and 6.69).

Assumption 13: no address-of, assembly, or external calls. Using the concrete kernel function table we can show that during all possible executions of the page-fault handler no function of the abstract kernel is called. That means that no external calls might occur. As for the absence of the address-of operator and assembly statements, these conditions are shown by inspecting the demand paging code.

Assumption 14: static stack consumption. It is proven similarly by analyzing the defined program code:

$$(ft_{\text{CK}}(\Pi_{\text{AK}}), \texttt{pfh\_init}, 52) \in SE, \quad (ft_{\text{CK}}(\Pi_{\text{AK}}), \texttt{pfh\_touch\_addr}, 136) \in SE.$$

Assumption 15: dynamic stack consumption. This will be added as an assumption to the top-level correctness theorems of the page-fault handler since we do not know the size of the already consumed stack.

Assumption 16: heap consumption. In case of the initialization code we know that before the function call the heap was empty. Analyzing the initialization code we show that that the elements are allocated during the function execution are: (i) the page table space (a two-dimensional array of thirty-two-bit natural numbers with dimensions TOT_PGS_PT and PTES_PER_PG), and (ii) and USER_PGS page descriptors, each occupies 20 bytes. Therefore, we conclude

$$\texttt{HEAP\_START} + \texttt{TOT\_PGS\_PT} \cdot \texttt{PTES\_PER\_PG} \cdot 4 + \texttt{USER\_PGS} \cdot 20 < \texttt{ZFP} \cdot \texttt{PG\_SZ}.$$

In case of the page-fault handler we do not allocate any elements on the heap.

7.2 Initialization Code Top-Level Correctness

We have already defined all functional specifications as well as abstraction relations used in the formulation of demand paging top-level correctness theorems. These theorems also contain a number of technical pre- and postconditions necessary to apply the theorems in the context of CVM verification. Mainly, these conditions correspond to parts of either the validity of small-step configurations (for property transfer) $valid_{\mathrm{SS}}$ (Section 3.5) or the transition invariant *trans-inv*? (Definition 3.3). We group them below.

So far all our demand paging correctness lemmas assume the the program rest consists only of a single function call statement statement. According to C0 validity it means that the local stack consists only of one local frame of a callee. In order to adapt this to an application in the context of the CVM proof we switch to complete C0 configurations of the concrete kernel. This step is justified using the fact that an execution of the function call does not affect the local frames below the callee's one and it also does not affect the program rest tree behind the discussed call statement. Thus, below we write $hd(s2l(c_{\mathrm{SS}}.prog) = SCall(\ldots)$ instead of $c_{\mathrm{SS}}.prog = SCall(\ldots)$.

Technical preconditions. We impose the following restrictions on the C0 small-step semantics configuration c_{SS} of the concrete kernel at the point of the initialization code invocation: (i) validity of the C0 configuration, (ii) only pointers to heap locations in the memory are allowed, (iii) the top-most statement of the program rest is a call to the initialization code of the demand paging, (iv) the return variable is in the top-local memory frame, (v) the global symbol table of the C0 configuration is the global symbol table of the concrete kernel program, (vi) the heap symbol table is empty, and (vii) the local stack is appropriately bounded. Formally these requirements are collected in the following predicate.

DEFINITION 7.14 ▶ Technical precondition of initialization code

$$\begin{aligned} PRE^{\mathrm{tech}}_{\mathrm{init}}?(\Pi_{\mathrm{AK}}, c_{\mathrm{SS}}) = \\ & c_{\mathrm{SS}} \in C0'\surd(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \\ \wedge\ & \textit{only-heap-pointer?}(c_{\mathrm{SS}}.mem) \\ \wedge\ & hd(s2l(c_{\mathrm{SS}}.prog)) = SCall(r, \texttt{pfh\_init}, [\,]) \\ \wedge\ & r \in map(fst, lst_{\mathrm{top}}(c_{\mathrm{SS}}.mem)) \\ \wedge\ & gst(c_{\mathrm{SS}}.mem) = gst_{\mathrm{CK}}(\Pi_{\mathrm{AK}}) \\ \wedge\ & \textit{glob-init?}(c_{\mathrm{SS}}.mem) \\ \wedge\ & hst(c_{\mathrm{SS}}.mem) = [\,] \\ \wedge\ & asize_{\mathrm{st}^*}(sc(c_{\mathrm{SS}}.mem).lst) + 52 \leq \texttt{ABASE}_{\mathrm{hm}} - \texttt{ABASE}_{\mathrm{lm}} \end{aligned}$$

Technical postconditions. After the concrete kernel configuration c_{SS} transits to c'_{SS} by executing the cal to the initialization code the following technical postconditions hold: (i) the validity of the C0 configuration, (ii) only pointers to heap locations in the memory are allowed, (iii) every type in the heap has a proper name in the type environment, (iv) the program rest is updated by removing the top-most statement, (v) the global and heap memories are unchanged at all variables/locations except for those that are allowed to be modified by the initialization code, (vi) the C0 memory invariant (described below). We collect these statement into a single predicate.

◀ DEFINITION 7.15
Technical postcondition of initialization code

$$
\begin{aligned}
&POST^{\mathrm{tech}}_{\mathrm{init}}?(\Pi_{\mathrm{AK}}, c_{\mathrm{SS}}, c'_{\mathrm{SS}}) = \\
&\qquad c'_{\mathrm{SS}} \in \mathit{C0}'\surd(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \\
&\quad \wedge \quad \mathit{only\text{-}heap\text{-}pointer}?(c'_{\mathrm{SS}}.\mathit{mem}) \\
&\quad \wedge \quad \mathit{named\text{-}ty}?(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{hst}(c'_{\mathrm{SS}}.\mathit{mem})) \\
&\quad \wedge \quad c'_{\mathrm{SS}}.\mathit{prog} = \mathit{rem\text{-}1st\text{-}stmt}(c_{\mathrm{SS}}.\mathit{prog}) \\
&\quad \wedge \quad \mathit{unchanged}_{\mathrm{GM,HM}}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}.\mathit{mem}, c'_{\mathrm{SS}}.\mathit{mem}, \\
&\qquad\qquad \mathit{modified}_{\mathrm{init}}, [0..\texttt{USER\_PGS}]) \\
&\quad \wedge \quad \mathit{C0mem\text{-}inv}?(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}.\mathit{mem}, c'_{\mathrm{SS}}.\mathit{mem}, \\
&\qquad\qquad \mathit{hst}_{\mathrm{CVM}}, r, \mathit{mcell}_{\mathrm{int}}(0)).
\end{aligned}
$$

Let mc and mc' be memory configuration before and after execution of some call statement, let st_{heap} be the additional part of the heap symbol table which has been allocated during this call, and let *ret-var* and *ret-val* be a name and a value of the return variable of the call, respectively. The *C0 memory invariant* [Tsy09, Definition 7.31] $\mathit{C0mem\text{-}inv}?(te, mc, mc', st_{\mathrm{heap}}, \mathit{ret\text{-}var}, \mathit{ret\text{-}val})$ is a conjunction of the following facts: (i) global symbol tables of both memory configurations are equal whereas the set of initialized global variables of mc is included in that of mc', (ii) the lengths of local memory stacks are equal and these stack differ only at the topmost element, (iii) top return destination and top local symbol tables of memory configurations mc and mc' are equal whereas the set of initialized local variables of mc is a subset of the initialized local variables of mc' — the call initializes the result variable, (iv) all initialized top local variables have the same values in both memory configurations, (v) the value of the return variable *ret-var* is *ret-val*, and (vi) the heap symbol table of mc' is composed of the heap symbol table of mc and st_{heap}.

The predicate $\mathit{unchanged}_{\mathrm{GM,HM}}(te, mc, mc', \mathit{vars}_{\mathrm{gm}}, \mathit{ind\text{-}set}_{\mathrm{heap}})$ [Tsy09, Definition 7.29] compares two C0 memory configurations and asserts that they differ only at given global and heap variable names (locations) specified by $\mathit{vars}_{\mathrm{gm}}$ and $\mathit{ind\text{-}set}_{\mathrm{heap}}$, respectively.

With the function $\mathit{rem\text{-}1st\text{-}stmt}(c_{\mathrm{SS}}.\mathit{prog})$ [Tsy09, Definition 7.32] we remove the top-most statement in the program state.

Correctness theorem. The top-level correctness theorem of the initialization code has the following assumptions.

- Validity of involved models/concepts: underlying VAMP ISA configuration (5), hard disk (3), and execution sequence (3).
- Simulation relation between a C0 configuration of the concrete kernel and the ISA configuration (4).

- Preconditions to execution on different levels: abstract kernel properties (1), ISA code invariants (6), system-mode execution (7), technical C0 preconditions (8), and functional preconditions of the initialization code expressed in C0 small-step semantics (9).

On the side of conclusions the theorem claims the existence of an updated VAMP ISA configuration achieved in T steps and an updated C0 configuration of the concrete kernel such that the following holds.

- The ISA computation succeeds and all devices except for the swap hard disk are untouched (10).
- Validity of the hard disk (11) and ISA (13), simulation of C0 by ISA (12), as well as ISA code invariants (14) and system-mode execution condition (15) are preserved.
- Technical (17) and functional (18) postconditions as well as the zero-filled page condition (16) hold.
- The abstraction relation for user processes is established with the initial configuration of user processes (19).

THEOREM 7.16 ▶ Top-level correctness of initialization code

$$
\begin{aligned}
&(1) && \mathit{abs\text{-}kernel\text{-}props}(\Pi_{\mathrm{AK}})\\
&(2) && \wedge\ \mathit{seq}\surd(\mathit{seq}, c_{\mathrm{ISA+DS}}.\mathit{devs})\\
&(3) && \wedge\ \mathit{hd}\surd(c_{\mathrm{ISA+DS}}.\mathit{devs})\\
&(4) && \wedge\ \mathit{C0\text{-}sim\text{-}isa?}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}, c_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(5) && \wedge\ \mathit{isa}\surd(c_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(6) && \wedge\ \mathit{code\text{-}inv?}(\Pi_{\mathrm{AK}}, c_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(7) && \wedge\ \mathit{sys\text{-}exec}_{\mathrm{ISA}}?(c_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(8) && \wedge\ PRE_{\mathrm{init}}^{\mathrm{tech}}?(\Pi_{\mathrm{AK}}, c_{\mathrm{SS}})\\
&(9) && \wedge\ PRE_{\mathrm{init}}^{\mathrm{SS}}?(c_{\mathrm{SS}}, te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}))\\
& && \longrightarrow\ \exists\, T, c'_{\mathrm{ISA+DS}}, c'_{\mathrm{SS}} : \delta^{T}_{\mathrm{ISA+DS}}(c_{\mathrm{ISA+DS}}, \mathit{seq}) = c'_{\mathrm{ISA+DS}}\\
&(10) && \quad \wedge\ \mathit{non\text{-}interf\text{-}dev'?}(c_{\mathrm{ISA+DS}}.\mathit{devs}, c'_{\mathrm{ISA+DS}}.\mathit{devs}, \mathit{seq}, T)\\
&(11) && \quad \wedge\ \mathit{hd}\surd(c'_{\mathrm{ISA+DS}}.\mathit{devs})\\
&(12) && \quad \wedge\ \mathit{C0\text{-}sim\text{-}isa?}(te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), ft_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c'_{\mathrm{SS}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(13) && \quad \wedge\ \mathit{isa}\surd(c'_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(14) && \quad \wedge\ \mathit{code\text{-}inv?}(\Pi_{\mathrm{AK}}, c'_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(15) && \quad \wedge\ \mathit{sys\text{-}exec}_{\mathrm{ISA}}?(c'_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(16) && \quad \wedge\ \mathit{zfp\text{-}cond?}(c'_{\mathrm{ISA+DS}}.\mathit{cpu})\\
&(17) && \quad \wedge\ POST_{\mathrm{init}}^{\mathrm{tech}}?(\Pi_{\mathrm{AK}}, c_{\mathrm{SS}}, c'_{\mathrm{SS}})\\
&(18) && \quad \wedge\ POST_{\mathrm{init}}^{\mathrm{SS}}?(c'_{\mathrm{SS}}, r, te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{gvars}_{\mathrm{active}}, \mathit{gvars}_{\mathrm{free}})\\
&(19) && \quad \wedge\ \mathcal{B}(\mathit{ups}_{\mathrm{init}}, c'_{\mathrm{ISA+DS}})
\end{aligned}
$$

PROOF ▶ We apply Lemma 6.65 in order to discharge the functional postcondition and obtain an extended C0 computation. By this we obtain a description of physical and swap memories in terms of the extended state. We apply Theorem 7.12 as described in Section 7.1.3 to map modifications done over the extended state to real physical ISA and hard-disk memories.

Having this, the most important proof step is to establish the abstraction relation for user processes $\mathcal{B}$. Here we exploit two facts from the initialization code specification (Section 5.3.2). All registers except PTO and PTL are initialized with zeros as claimed by the specification (Definition 5.19). The PTL

registers are initialized with -1 for all user processes: no virtual memory is allocated for any proces. This gives the term describing the virtual memory equality for free (cf. Definition 2.29). The remaining proof goal is to show that the values of PTO and PTL registers taken from the ISA memory correspond to those specified by the small-step semantics postcondition of the initialization code. We use the simulation relation between C0 and ISA *C0-sim-isa*? to conclude that. ■

7.3 Page-Fault Handler Top-Level Correctness

Like we did for the initialization code we define technical pre- and postconditions for the page-fault handler top-level correctness theorem. The respective predicates additionally take as arguments the values *pid*, *addr*, *intent* and *count* of parametrs to the handler's call.

Technical preconditions. At the moment the concrete kernel calls the page-fault handler there exist expressions for the call parameters and the following technical preconditions have to hold: (i) validity of the C0 configuration, (ii) only pointers to heap locations in the memory are allowed, (iii) every type in the heap has a proper name in the type environment, (iv) the top-most statement of the program rest is a call to the page-fault handler with corresponding parameter expressions, (v) the return variable is in the top-local memory frame, (vi) the global symbol table of the C0 configuration is the global symbol table of the concrete kernel program, (vii) all global variables of the concrete kernel are initialized, (viii) the heap symbol table of the CVM implementation lies at the beginning of the concrete kernel heap symbol table, (ix) the local stack is appropriately bounded, (x) the heap is appropriately bounded, and (xi) the parameter expressions are evaluated to the specified values.

◀ DEFINITION 7.17
Technical precondition of page-fault handler

$$
\begin{aligned}
&PRE^{\text{tech}}_{\text{ta}}?(\Pi_{\text{AK}}, c_{\text{SS}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) = \\
&\qquad \exists e_{\mathit{pid}}, e_{\mathit{addr}}, e_{\mathit{intent}}, e_{\mathit{count}} : \\
&\qquad c_{\text{SS}} \in \mathit{C0}'\surd(te_{\text{CK}}(\Pi_{\text{AK}}), ft_{\text{CK}}(\Pi_{\text{AK}})) \\
&\quad \wedge \quad \mathit{only\text{-}heap\text{-}pointer?}(c_{\text{SS}}.\mathit{mem}) \\
&\quad \wedge \quad \mathit{named\text{-}ty?}(te_{\text{CK}}(\Pi_{\text{AK}}), \mathit{hst}(c_{\text{SS}}.\mathit{mem})) \\
&\quad \wedge \quad \mathit{hd}(\mathit{s2l}(c_{\text{SS}}.\mathit{prog})) = \mathit{SCall}(r, \texttt{pfh\_touch\_addr}, \\
&\qquad\qquad\qquad [e_{\mathit{pid}}, e_{\mathit{addr}}, e_{\mathit{intent}}, e_{\mathit{count}}]) \\
&\quad \wedge \quad r \in \mathit{map}(\mathit{fst}, \mathit{lst}_{\text{top}}(c_{\text{SS}}.\mathit{mem})) \\
&\quad \wedge \quad \mathit{gst}(c_{\text{SS}}.\mathit{mem}) = \mathit{gst}_{\text{CK}}(\Pi_{\text{AK}}) \\
&\quad \wedge \quad \mathit{glob\text{-}init?}(c_{\text{SS}}.\mathit{mem}) \\
&\quad \wedge \quad \forall\, i < |\mathit{hst}_{\text{CVM}}| : \ \mathit{hst}(c_{\text{SS}}.\mathit{mem})[i] = \mathit{hst}_{\text{CVM}}[i] \\
&\quad \wedge \quad \mathit{asize}_{\text{st}^*}(\mathit{sc}(c_{\text{SS}}.\mathit{mem}).\mathit{lst}) + 136 \leq \texttt{ABASE}_{\text{hm}} - \texttt{ABASE}_{\text{lm}} \\
&\quad \wedge \quad \mathit{asize}_{\text{heap}}(\mathit{hst}(c_{\text{SS}}.\mathit{mem})) < \texttt{ASIZE}^{\text{max}}_{\text{hm}} \\
&\quad \wedge \quad \mathit{param}^{\text{SS}}_{\text{ta}}?(c_{\text{SS}}, te_{\text{CK}}(\Pi_{\text{AK}}), e_{\mathit{pid}}, e_{\mathit{addr}}, e_{\mathit{intent}}, e_{\mathit{count}}, \\
&\qquad\qquad \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count})
\end{aligned}
$$

Technical postconditions. On the side of technical postconditions of the page-fault handler top-level theorem we have the following: (i) the validity of the C0 configuration, (ii) only pointers to heap locations in the memory are allowed, (iii) every type in the heap has a proper name in the type environment, (iv) the

program rest is updated by removing the top-most statement, (v) the global and heap memories are unchanged at all variables/locations except for those that are allowed to be modified by the page-fault handler, (vi) the C0 memory invariant.

DEFINITION 7.18 ▶
Technical postcondition of page-fault handler

$$
\begin{aligned}
&\mathit{POST}^{\mathrm{tech}}_{\mathrm{ta}}?(\Pi_{\mathrm{AK}}, c_{\mathrm{SS}}, c'_{\mathrm{SS}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count}) = \\
&\quad c'_{\mathrm{SS}} \in \mathit{C0'}\surd(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{ft}_{\mathrm{CK}}(\Pi_{\mathrm{AK}})) \\
&\wedge\ \mathit{only\text{-}heap\text{-}pointer?}(c'_{\mathrm{SS}}.\mathit{mem}) \\
&\wedge\ \mathit{named\text{-}ty?}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), \mathit{hst}(c'_{\mathrm{SS}}.\mathit{mem})) \\
&\wedge\ c'_{\mathrm{SS}}.\mathit{prog} = \mathit{rem\text{-}1st\text{-}stmt}(c_{\mathrm{SS}}.\mathit{prog}) \\
&\wedge\ \mathit{unchanged}_{\mathrm{GM,HM}}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}.\mathit{mem}, c'_{\mathrm{SS}}.\mathit{mem}, \\
&\qquad\qquad \mathit{modified}_{\mathrm{ta}}, [0..\texttt{USER\_PGS}]) \\
&\wedge\ \mathit{C0mem\text{-}inv?}(\mathit{te}_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), c_{\mathrm{SS}}.\mathit{mem}, c'_{\mathrm{SS}}.\mathit{mem}, \\
&\qquad\qquad [\,], r, \mathit{mcell}_{\mathrm{nat}}(\mathit{pfh\text{-}ta\text{-}res}_{\mathrm{SS}}(c_{\mathrm{PFH}}, \mathit{pid}, \mathit{addr}, \mathit{intent}, \mathit{count})).
\end{aligned}
$$

Weak relation for user processes. Instead of the relation for user processes $\mathcal{B}$ the top-level correctness theorem of the page-fault handler concludes its *weak version*.

Basically, the weak version of the relation $\mathcal{B}$, denoted by $\mathcal{B}'$, states that relation $\mathcal{B}$ holds for all virtual pages but one in the system. Recall that when the page fault handler is called for a pair (*pid*, *addr*) with an intention *intent* = `OVERWRITE` we know that the kernel is going to overwrite the entire data stored in the physical page corresponding to (*pid*, *addr*). An example of that is the primitive `cvm_copy` [Tsy09, Section 9.3] which copies data between two processes: some portions of the target process's memory will be overwritten. Because of that an optimization was introduced in the page-fault handler: in case *intent* = `OVERWRITE` we do not swap in the data for the touched page but only update its page table entry (consider lines 50–58 of Listing 5.4). The latter includes setting the valid bit. Since the page table entry corresponding to the pair (*pid*, *addr*) is validated while the respective physical page is still filled with the old data the relation for user processes is violated for that particular page. Note the relation $\mathcal{B}$ will be regained during the kernel run as soon as the kernel puts the right data at the discussed page. However, immediately after the page-fault handler only the relation $\mathcal{B}'$ holds.

In order to define $\mathcal{B}'$ formally let us first adapt the assembly equality predicate *asm-equal*? (Definition 2.29). The *weak assembly equality*, denoted by the predicate *asm-equal'*?, is additionally parametrized by a boolean flag *flag* and a virtual page index *vpx*. The new predicate differs from *asm-equal*? in the memory conjunct: virtual memories of two assembly configurations are equal for all addresses a outside the page specified by *vpx* and only if the flag is not raised.

DEFINITION 7.19 ▶
Assembly configurations equality: weak version

$$
\begin{aligned}
&\mathit{asm\text{-}equal'}?(c^1_{\mathrm{ASM}}, c^2_{\mathrm{ASM}}, \mathit{vm\text{-}size}, \mathit{flag}, \mathit{vpx}) = \\
&\quad c^1_{\mathrm{ASM}}.\mathit{dpc} = c^2_{\mathrm{ASM}}.\mathit{dpc} \\
&\wedge\ c^1_{\mathrm{ASM}}.\mathit{pc} = c^2_{\mathrm{ASM}}.\mathit{pc} \\
&\wedge\ \mathit{tl}(c^1_{\mathrm{ASM}}.\mathit{gpr}) = \mathit{tl}(c^2_{\mathrm{ASM}}.\mathit{gpr}) \\
&\wedge\ \mathit{stored\text{-}spr}(c^1_{\mathrm{ASM}}.\mathit{spr}) = \mathit{stored\text{-}spr}(c^2_{\mathrm{ASM}}.\mathit{spr}) \\
&\wedge\ \forall a < \mathit{vm\text{-}size} : \neg(\mathit{flag} \wedge \mathit{px}(a) = \mathit{vpx}) \longrightarrow c^1_{\mathrm{ASM}}.m(a/4) = c^2_{\mathrm{ASM}}.m(a/4)
\end{aligned}
$$

Having this, it is fairly easy to define the relation $\mathcal{B}'$. Its definition extends the parameters of $\mathcal{B}$ (Definition 2.30) with a boolean flag *flag*, a process identifier *pid*, and a virtual page index *vpx*. The new relation claims for all user processes p the weak assembly equality between their implementation $vm(c_{\text{ISA+DS}}, p)$ and specification $ups(p)$ such that the parameter *flag* of *asm-equal′*? is strengthened with a condition $p = pid$.

◀ DEFINITION 7.20
Relation for user processes: weak version

$$\begin{aligned}&\mathcal{B}'(ups, c_{\text{ISA+DS}}, flag, pid, vpx) = \\ &\quad \forall\, 0 < p < \texttt{MAX\_PID} : asm\text{-}equal'?(vm(c_{\text{ISA+DS}}, p), ups(p), \\ &\qquad\qquad (ptl_{\text{CVM}}(c_{\text{ISA+DS}}.cpu, p) + 1) \cdot 2^{12}, \\ &\qquad\qquad flag \wedge (p = pid), vpx)\end{aligned}$$

By setting the flag parameter of $\mathcal{B}'$ to $intent = \texttt{OVERWRITE}$ we can express the relation for user processes which holds after the page-fault hander:

$$\mathcal{B}'(ups, c_{\text{ISA+DS}}, intent = \texttt{OVERWRITE}, pid, px(addr)).$$

Correctness theorem. Technically, the formulation of the page-fault handler top-level correctness theorem is close to Theorem 7.16 (the top-level theorem of the initialization code). Therefore we list only those assumptions and conclusions that differ.

On the side of assumptions the theorem has the zero-filled page condition (1) and the relation for user processes (2). Both are maintained throughout CVM and the page-fault handler top-level correctness theorem is supposed to be applied during the concrete kernel executions. Further, we assume technical (3) and functional (4) preconditions of the page-fault handler. As for the conclusions, the theorem claims that technical (5) and functional (6) page-fault handler postconditions hold as well as the weak version of the relation for user processes.

◀ THEOREM 7.21
Top-level correctness of page-fault handler

$$\begin{aligned}
& & & abs\text{-}kernel\text{-}props(\Pi_{\text{AK}}) \\
& & \wedge\ & seq\surd(seq, c_{\text{ISA+DS}}.devs) \\
& & \wedge\ & hd\surd(c_{\text{ISA+DS}}.devs) \\
& & \wedge\ & C0\text{-}sim\text{-}isa?(te_{\text{CK}}(\Pi_{\text{AK}}), ft_{\text{CK}}(\Pi_{\text{AK}}), c_{\text{SS}}, c_{\text{ISA+DS}}.cpu) \\
& & \wedge\ & isa\surd(c_{\text{ISA+DS}}.cpu) \\
& & \wedge\ & code\text{-}inv?(\Pi_{\text{AK}}, c_{\text{ISA+DS}}.cpu) \\
& & \wedge\ & sys\text{-}exec_{\text{ISA}}?(c_{\text{ISA+DS}}.cpu) \\
& (1) & \wedge\ & zfp\text{-}cond?(c_{\text{ISA+DS}}.cpu) \\
& (2) & \wedge\ & \mathcal{B}(ups, c_{\text{ISA+DS}}) \\
& (3) & \wedge\ & PRE^{\text{tech}}_{\text{ta}}?(\Pi_{\text{AK}}, c_{\text{SS}}, pid, addr, intent, count) \\
& (4) & \wedge\ & PRE^{\text{SS}}_{\text{ta}}?(c_{\text{SS}}, c_{\text{PFH}}, pid, addr, intent, count, \\
& & & \qquad te_{\text{CK}}(\Pi_{\text{AK}}), locs_{\text{active}}, locs_{\text{free}}) \\
& \longrightarrow & \exists\, T, c'_{\text{ISA+DS}}, c'_{\text{SS}} : \delta^{T}_{\text{ISA+DS}}(c_{\text{ISA+DS}}, seq) = c'_{\text{ISA+DS}} \\
& & \quad \wedge\ & non\text{-}interf\text{-}dev'?(c_{\text{ISA+DS}}.devs, c'_{\text{ISA+DS}}.devs, seq, T) \\
& & \quad \wedge\ & hd\surd(c'_{\text{ISA+DS}}.devs) \\
& & \quad \wedge\ & C0\text{-}sim\text{-}isa?(te_{\text{CK}}(\Pi_{\text{AK}}), ft_{\text{CK}}(\Pi_{\text{AK}}), c'_{\text{SS}}, c'_{\text{ISA+DS}}.cpu) \\
& & \quad \wedge\ & isa\surd(c'_{\text{ISA+DS}}.cpu) \\
& & \quad \wedge\ & code\text{-}inv?(\Pi_{\text{AK}}, c'_{\text{ISA+DS}}.cpu) \\
& & \quad \wedge\ & sys\text{-}exec_{\text{ISA}}?(c'_{\text{ISA+DS}}.cpu) \\
& & \quad \wedge\ & zfp\text{-}cond?(c'_{\text{ISA+DS}}.cpu) \\
& (5) & \quad \wedge\ & POST^{\text{tech}}_{\text{ta}}?(\Pi_{\text{AK}}, c_{\text{SS}}, c'_{\text{SS}}, pid, addr, intent, count)
\end{aligned}$$

$$\wedge \quad POST^{\mathrm{SS}}_{\mathrm{ta}}?(c'_{\mathrm{SS}}, c_{\mathrm{PFH}}, pid, addr, intent, count, r, te_{\mathrm{CK}}(\Pi_{\mathrm{AK}}), locs_{\mathrm{active}}, locs_{\mathrm{free}}) \tag{6}$$

$$\wedge \quad \mathcal{B}'(ups, c'_{\mathrm{ISA+DS}}, intent = \texttt{OVERWRITE}, pid, px(addr)). \tag{7}$$

PROOF ► We apply Lemma 6.69 in order to discharge the functional postcondition and obtain an extended C0 computation. By this we obtain a description of physical and swap memories in terms of the extended state. We apply Theorem 7.12 as described in Section 7.1.3 to obtain relations between the extended state and real physical ISA and hard-disk memories. The relation for user processes constitutes the major proof effort.

The page-fault handler does not change registers of user processes. Hence, the resister conjuncts of the relation are easy to show. As for the virtual memories of the processes, its contents stay the same. Actually, because of these facts we use the same user-processes components *ups* in the assumptions and conclusions of the theorem.

The only condition affected by the page-fault handler is the place where a certain virtual memory page lies: in the physical memory or on the hard disk. Formally, this boils down to investigating the address translation mechanism.

For the user-processes relation proof we have to map the changes specified for the extended state to the real machine memory and the hard disk. These two are related by the predicates $xconsis_{\mathrm{mem}}?$ and $xconsis_{\mathrm{swap}}?$ which are part of the relation $xconsis?$.

Let us consider as an example the case of legal user page fault in the situation where some page has to be swapped out (this turns out to be the most complicated case of page-fault handler execution). Let *mem* and *swap* be the components of the extended state before the function execution. Suppose a page fault occurs for the pair (pid_{in}, va_{in}). In order to treat this page fault the handler has to swap out the page corresponding to some other pair (pid_{out}, va_{out}). The extended state components after the handler execution are as follows (cf. Figure 7.3).

$$mem'[i] = \begin{cases} swap[offs_{\mathrm{swap}}(spx_{\mathrm{PFH}}(c_{\mathrm{PFH}}, px(va_{in}), pid_{in}))] \\ \qquad\qquad \text{if } i = offs_{\mathrm{mem}}(px(pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_{out}, va_{out}))) \\ mem[i] \qquad \text{otherwise} \end{cases}$$

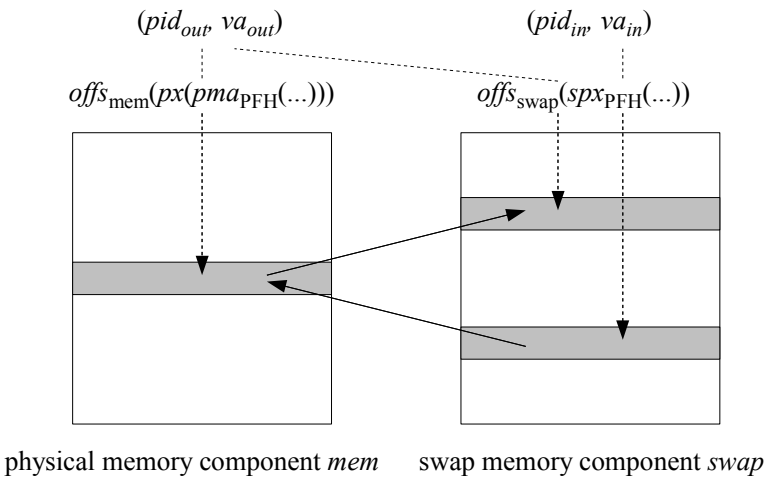
$$swap'[i] = \begin{cases} mem[offs_{\mathrm{mem}}(px(pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_{out}, va_{out})))] \\ \qquad\qquad \text{if } i = offs_{\mathrm{swap}}(spx_{\mathrm{PFH}}(c_{\mathrm{PFH}}, px(va_{out}), pid_{out})) \\ swap[i] \qquad \text{otherwise} \end{cases}$$

Note that since the extended state models only parts of physical and swap memories the corresponding offset functions $offs_{\mathrm{swap}}$ and $offs_{\mathrm{mem}}$ are applied (cf. Definitions 4.24 and 4.25).

We instantiate the extended state component before the function execution with $mem2x(c_{\mathrm{ISA}}.m)$ and $swap2x(c_{\mathrm{HD}}.sm)$ and by unfolding $xconsis_{\mathrm{mem}}?$ and $xconsis_{\mathrm{swap}}?$ obtain the following.

$$\begin{aligned} &\forall\, ad : \texttt{PG\_SZ} \cdot \texttt{ZFP} \leq ad \\ &\quad \wedge\, ad < \texttt{PG\_SZ} \cdot (\texttt{ZFP} + \texttt{USER\_PGS} + 1) \\ &\quad \longrightarrow \langle c'_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(ad)) \rangle = \end{aligned}$$

Figure 7.3: Page movement during page-fault handling



$$\begin{cases} c_{\text{HD}}.sm[(\texttt{PG\_SZ} \cdot spx_{\text{PFH}}(c_{\text{PFH}}, px(va_{in}), pid_{in}) + bx(ad))/4] \\ \qquad\qquad \text{if } px(ad) = px(pma_{\text{PFH}}(c_{\text{PFH}}, pid_{out}, va_{out})) \\ \langle c_{\text{ISA}}.m_{\text{word}}(bin(ad))\rangle \qquad \text{otherwise} \end{cases}$$

$$\begin{aligned} &\forall\ ad : \texttt{PG\_SZ\_WD} \cdot \texttt{BOOT\_PGS} \leq ad \\ &\quad \wedge\ ad < \texttt{PG\_SZ\_WD} \cdot (\texttt{BOOT\_PGS} + \texttt{TOT\_BIG\_PGS} \cdot \texttt{PGS\_PER\_BIG\_PG}) : \\ &\quad \longrightarrow c'_{\text{HD}}.sm[ad] = \\ &\begin{cases} \langle c_{\text{ISA}}.m_{\text{word}}(bin(\texttt{PG\_SZ} \cdot px(pma_{\text{PFH}}(c_{\text{PFH}}, pid_{out}, va_{out})) + bx(4 \cdot ad)))\rangle \\ \qquad\qquad \text{if } px(4 \cdot ad) = spx_{\text{PFH}}(c_{\text{PFH}}, px(va_{out}), pid_{out}) \\ c_{\text{HD}}.sm[ad] \qquad \text{otherwise} \end{cases} \end{aligned}$$

Note that the physical memory address and the swap page index computations over the abstract PFH state are semantically equivalent to those which are defined in the CVM theories and used in the definition of the relation $\mathcal{B}$ (Definitions 2.22 and 2.26).

At the end of the day, we want to show that the memory relation conjunct of the relation for user processes holds for the same abstract user processes configurations as before the function call. This could be reduced to the claim that for every user process *pid* and address *va* which belongs to the set of allocated addresses for that process the value constructed by the $\mathcal{B}$-relation before the function call is the same as after. Here we distinguish three cases.

Case 1. The pair (pid, va) corresponds to the same page table entry as the pair (pid_{in}, va_{in}), i.e., $pid = pid_{in}$ and $px(va) = px(va_{in})$. From the functional postcondition it follows that the valid bit for that page was not set before the function call and is set after the call. So, for this pair we should read the value from swap memory before the call and from the physical memory after the call.

The value before the call is

$$c_{\mathrm{HD}}.sm[(spx_{\mathrm{PFH}}(c_{\mathrm{PFH}}, px(va), pid) \cdot \mathtt{PG\_SZ} + bx(va))/4].$$

The page index of the physical memory address is

$$\begin{array}{rl} & px(pma_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, pid, va)) \\ = & px(pte_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, pid_{in}, px(va_{in}))) \\ = & px(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_{out}, px(va_{out}))) \\ = & px(pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_{out}, va_{out})). \end{array}$$

So, the value after the call is

$$\begin{array}{rl} & \langle c'_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(pma_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, pid, va)))\rangle \\ = & c_{\mathrm{HD}}.sm[(\mathtt{PG\_SZ} \cdot spx_{\mathrm{PFH}}(c_{\mathrm{PFH}}, px(va_{in}), pid_{in}) \\ & + bx(pma_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, pid, va)))/4] \\ = & c_{\mathrm{HD}}.sm[(\mathtt{PG\_SZ} \cdot spx_{\mathrm{PFH}}(c_{\mathrm{PFH}}, px(va), pid) + bx(va))/4]. \end{array}$$

Case 2. The pair (pid, va) corresponds to the same page table entry as the pair (pid_{out}, va_{out}), i.e., $pid = pid_{out}$ and $px(va) = px(va_{out})$. From the functional postcondition it follows that the valid bit for that page was set before the function call and is not set after the call. So, for this pair we should read the value from physical memory before the call and from the swap memory after the call. Value before the call is

$$\langle c_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, va)))\rangle.$$

The page index of the swap memory address is

$$\begin{array}{rl} & px(\mathtt{PG\_SZ} \cdot spx_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, px(va), pid) + bx(va)) \\ = & spx_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, px(va), pid) \\ = & spx_{\mathrm{PFH}}(c_{\mathrm{PFH}}, px(va_{out}), pid_{out}) \end{array}$$

So, the value after the call is

$$\begin{array}{rl} & c'_{\mathrm{HD}}.sm[(spx_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, px(va), pid) \cdot \mathtt{PG\_SZ} + bx(va))/4] \\ = & \langle c_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(\mathtt{PG\_SZ} \cdot px(pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_{out}, va_{out})) + \\ & bx(spx_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, px(va), pid) \cdot \mathtt{PG\_SZ} + bx(va))))\rangle \\ = & \langle c_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(\mathtt{PG\_SZ} \cdot px(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_{out}, px(va_{out}))) + bx(va))))\rangle \\ = & \langle c_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(\mathtt{PG\_SZ} \cdot px(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, px(va))) + bx(va))))\rangle \\ = & \langle c_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, va)))\rangle. \end{array}$$

Case 3. The pair (pid, va) does not correspond to any of the handled pages. In this case the page corresponding to (pid, va) resides at the same place before and after the function call. The "if" conditions in the equations presented in the beginning of this proof are not satisfied. Therefore we conclude

$$\langle c'_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(pma_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, pid, va)))\rangle = \langle c_{\mathrm{ISA}}.m_{\mathrm{word}}(bin(pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid, va)))\rangle$$

and

$$c'_{\mathrm{HD}}.sm[(spx_{\mathrm{PFH}}(c'_{\mathrm{PFH}}, px(va), pid) \cdot \mathtt{PG\_SZ} + bx(va))/4] = c_{\mathrm{HD}}.sm[(spx_{\mathrm{PFH}}(c_{\mathrm{PFH}}, px(va), pid) \cdot \mathtt{PG\_SZ} + bx(va))/4].$$

■

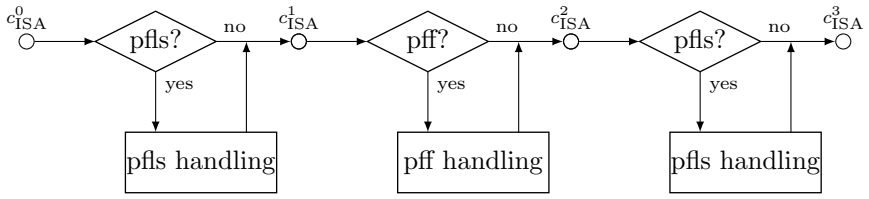


Figure 7.4: Verification diagram of user page fault handling.

7.4 Using Results in the CVM Proof

CVM invokes the initialization code of demand paging while initializing its data structures after a reset. Section 9.2.2 of Tsyban's thesis [Tsy09] describes verification of kernel initialization after a reset and gives details in which context the top-level correctness theorem of the initialization code (Theorem 7.16) is applied.

The page-fault handler is called by CVM in two situations: when page-fault exceptions occur during user steps and when the kernel executes CVM primitives that access user memory. In the second case the handler simulates address translation for CVM, which runs untranslated, and makes sure that the corresponding memory page is swapped in. In this section we briefly review these two cases.

7.4.1 Handling User Page Faults

During a single user step up to two page faults might occur: a page fault on fetch, and a page fault on load/store. Our page-fault handler is designed in a way that it guarantees that no more than two page faults occur while processing a single instruction. Hence, the following five situations are possible regarding page faults: (i) no page faults, (ii) only a page fault on fetch, (iii) only a page fault on load/store, (iv) a page fault on fetch followed by a page fault on load/store, and (v) a page fault on load/store followed by a page fault on fetch. Diagram 7.4 depicts all these cases.

Configurations c^0_{ISA}, c^1_{ISA}, c^2_{ISA}, and c^3_{ISA} are mapped to a single user process state at which the user attempts to make a step. In state c^1_{ISA} it is guaranteed that there is no page fault on fetch. In state c^2_{ISA} it is guaranteed that there is no page fault on load/store. In case there was a page fault on fetch we also can state its absence at this point. However, if there was no page fault on fetch, it could happen during the handling of the page fault on load/store, which will be signaled in the next step. Hence, generally we could not claim anything about the instruction page fault. Only in state c^3_{ISA} it is guaranteed that there both kinds of page-faults are absent.

The mentioned scenario is expressed as a formal lemma in Tsyban's thesis [Tsy09, Lemma 10.1]. The proof of this lemma essentially boils down to a triple application of the page-fault handler correctness theorem (Theorem 7.21). In order to guarantee liveness of the system it is necessary to argue that during the next call to the page-fault handler the page that was swapped in this time will not be swapped out.

The page-fault handler implementation respects the mentioned property. Suppose we want to claim that the page corresponding to a pair (pid, va) will still be in the physical memory after handling of the next page fault. If the page fault has occurred because the needed page was not in the physical memory, the handler has to load it from the swap memory. In case there is not a single vacant page in the physical memory, which is indicated by an empty free list, some page has to be evicted. According to our page-replacement strategy a page at the beginning of the active list has to be swapped out. The formula below claims that the page associated with the pair (pid, va) will not be evicted, hence, will not be swapped out during the next call to the handler:

$$c_{\mathrm{PFH}}.free = [\,] \longrightarrow \begin{pmatrix} c_{\mathrm{PFH}}.active[0].pid \\ c_{\mathrm{PFH}}.active[0].vpx \end{pmatrix} \neq \begin{pmatrix} pid \\ px(va) \end{pmatrix}.$$

In Section 9.2.5 of her doctoral thesis [Tsy09] Tsyban gives this predicate a meaning on the ISA level and formally proves that it holds after the page-fault handler call using the fact that we have at least two user pages in the physical memory.

7.4.2 Address Translation in CVM Primitives

When invoked in CVM primitives the page-fault handler is used for address translation and guaranteeing that specified pages reside in the physical memory. A good example of that is a primitive `cvm_copy`. It copies a specified amount of virtual memory from one process pid_1 at virtual address va_1 to another process pid_2 at virtual address va_2. Correctness of the primitive is shown in Section 9.3 of Tsyban's thesis [Tsy09]

In order to proceed with copying the primitive translates virtual addresses to physical ones by invoking the page-fault handler. An important property for the correctness proof of the primitive is the process memory isolation: two different pairs of process identifiers and virtual addresses $(pid_1, va_1) \neq (pid_2, va_2)$ are translated into two different physical addresses $pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_1, va_1) \neq pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_2, va_2)$. This property is respected by the page-fault handler and below we prove a lemma justifying it. Note that this lemma is also applied in similar cases of the CVM proof where it has to be shown that user processes operate in separate address space, e.g., the correctness lemma of a user step without interrupts [Tsy09, Lemma 10.2].

The property could be shown only for the valid page-fault handler configurations, user process identifiers, and appropriately bounded virtual addresses. The latter has to be less than the virtual memory amount of the respective process. Note that some virtual addresses are mapped to the zero-filled page. Without loss of generality we consider a situation when one of the addresses does not translate to the zero-filled page whereas the second address either translates to `ZFP` or to some other user page.

LEMMA 7.22 ► Correctness of address translation

$$\begin{aligned}
& (pid_1, va_1) \neq (pid_2, va_2) \\
\wedge\ & pfh\surd(c_{\mathrm{PFH}}) \\
\wedge\ & \mathit{user\text{-}pid?}(pid_1) \ \wedge\ \mathit{user\text{-}pid?}(pid_2) \\
\wedge\ & va_1 < (c_{\mathrm{PFH}}.ptl[pid_1] + 1) \cdot \mathtt{PG_SZ} \ \wedge\ va_2 < (c_{\mathrm{PFH}}.ptl[pid_2] + 1) \cdot \mathtt{PG_SZ} \\
\wedge\ & \mathit{valid?}(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_1, px(va_1))) \\
\wedge\ & (\mathit{valid?}(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_2, px(va_2))) \vee \mathit{zfp?}(c_{\mathrm{PFH}}, pid_2, px(va_2))) \\
\longrightarrow\ & pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_1, va_1) \neq pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_2, va_2)
\end{aligned}$$

We prove lemma by splitting cases according to the disjunction in the premises. ◀ PROOF

Case 1. The second pair of process identifier and virtual address corresponds to the zero-filled page:

$$\mathit{zfp?}(c_{\mathrm{PFH}}, \mathit{pid}_2, \mathit{px}(\mathit{va}_2)).$$

Using the page-fault handler validity conjunct *user-ppx-pt*? (Definition 4.47) which states that page table entries refer only to the user memory we reason as follows:

$$\begin{array}{rl} & \mathit{pma}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_2, \mathit{va}_2) \\ = & \mathtt{ZFP} \cdot \mathtt{PG\_SZ} + \mathit{bx}(\mathit{va}_2) \\ < & \mathtt{ZFP} \cdot \mathtt{PG\_SZ} + \mathtt{PG\_SZ} \\ = & \mathtt{KERNEL\_PGS} \cdot \mathtt{PG\_SZ} \\ \leq & \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_1, \mathit{px}(\mathit{va}_1))) \cdot \mathtt{PG\_SZ} \\ \leq & \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_1, \mathit{px}(\mathit{va}_1))) \cdot \mathtt{PG\_SZ} + \mathit{bx}(\mathit{va}_1) \\ = & \mathit{pma}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_1, \mathit{va}_1). \end{array}$$

From that we conclude that translated addresses are different.

Case 2. The second pair of process identifier and virtual address corresponds to the user memory:

$$\mathit{valid?}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_2, \mathit{px}(\mathit{va}_2))).$$

From the page-fault handler validity conjunct *valid-pte-descr-active*? (Definition 4.46) we conclude that two active lists's elements a_1 and a_2 (possibly equal) correspond to the given page table entries:

$$\begin{array}{ll} a_1 \in c_{\mathrm{PFH}}.\mathit{active} & a_2 \in c_{\mathrm{PFH}}.\mathit{active} \\ \wedge\ a_1.\mathit{pid} = \mathit{pid}_1 & \wedge\ a_2.\mathit{pid} = \mathit{pid}_2 \\ \wedge\ a_1.\mathit{vpx} = \mathit{px}(\mathit{va}_1) & \wedge\ a_2.\mathit{vpx} = \mathit{px}(\mathit{va}_2) \\ \wedge\ a_1.\mathit{ppx} = & \wedge\ a_2.\mathit{ppx} = \\ \quad \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_1, \mathit{px}(\mathit{va}_1))), & \quad \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_2, \mathit{px}(\mathit{va}_2))). \end{array}$$

Case 2.1: $a_1 = a_2$. In this case the corresponding process identifier as well as virtual and physical page indices are equal:

$$\begin{array}{rcl} \mathit{pid}_1 & = & \mathit{pid}_2, \\ \mathit{px}(\mathit{va}_1) & = & \mathit{px}(\mathit{va}_2), \\ \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_1, \mathit{px}(\mathit{va}_1))) & = & \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_2, \mathit{px}(\mathit{va}_2))). \end{array}$$

From the assumptions it follows that if the process identifiers are equal then the virtual addresses are different $\mathit{va}_1 \neq \mathit{va}_2$:

$$\begin{array}{rl} & (\mathit{va}_1 \neq \mathit{va}_2) \\ = & (\mathit{bx}(\mathit{va}_1) \neq \mathit{bx}(\mathit{va}_2)) \\ = & (\mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_1, \mathit{px}(\mathit{va}_1))) \cdot \mathtt{PG\_SZ} + \mathit{bx}(\mathit{va}_1) \\ & \quad \neq \mathit{px}(\mathit{pte}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_2, \mathit{px}(\mathit{va}_2))) \cdot \mathtt{PG\_SZ} + \mathit{bx}(\mathit{va}_2)) \\ = & (\mathit{pma}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_1, \mathit{va}_1) \neq \mathit{pma}_{\mathrm{PFH}}(c_{\mathrm{PFH}}, \mathit{pid}_2, \mathit{va}_2)). \end{array}$$

Case 2.2: $a_1 \neq a_2$. From the validity predicate *dstnct-ppx-active-free*? (Definition 4.44) we get that two different elements of the active list have different physical page indices:

$$a_1.\mathit{ppx} \neq a_2.\mathit{ppx}.$$

Without loss of generality, assume that $a_1.ppx < a_2.ppx$. Then we have:

$$\begin{aligned}
& pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_1, va_1) \\
= \;& px(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_1, px(va_1))) \cdot \texttt{PG_SZ} + bx(va_1) \\
= \;& a_1.ppx \cdot \texttt{PG_SZ} + bx(va_1) \\
< \;& a_1.ppx \cdot \texttt{PG_SZ} + \texttt{PG_SZ} \\
\leq \;& a_2.ppx \cdot \texttt{PG_SZ} \\
\leq \;& a_2.ppx \cdot \texttt{PG_SZ} + bx(va_2) \\
= \;& px(pte_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_2, px(va_2))) \cdot \texttt{PG_SZ} + bx(va_2) \\
= \;& pma_{\mathrm{PFH}}(c_{\mathrm{PFH}}, pid_2, va_2).
\end{aligned}$$

From that we conclude that translated addresses are different.

■

CHAPTER 8

Summary and Future Work

And that's all I have to say about that.

– *Forrest Gump*

The future is never clear.

– *Warren Buffet*

This thesis presents to the best of our knowledge the first example of pervasive formal verification of demand paging, a crucial component of every modern operating system. We for the first time applied the Verisoft's semantics stack which allows to combine results from sequential Hoare-logics-style reasoning about systems software like the page-fault handler on the low-level concurrent machine model. The results achieved within this thesis are successfully integrated into the correctness theorem of CVM, a verified framework for microkernel programmers.

At the time we started verification the implementation of demand paging has been intensively tested on the VAMP simulator. However, the verification disclosed the following errors.

- The initialization code of demand paging lacked a call to the zero fill page function. In fact, the zero-filled page was never initialized in the system.
- The page-fault handler lacked compatibility with executable bits of page table entries supported by the VAMP hardware.
- Constant `TOT_BIG_PGS` = 1152 was incorrectly assigned the value of 1024.
- In case the page-fault handler was invoked in order to prevent swapping out of a specified page, the page was identified only by its virtual page index instead of a pair `(pid,vpx)`. Only the pair uniquely defines the physical page.

Formal theories in Isabelle/HOL developed in the scope of this work comprise more than 450 definitions and 1500 lemmas proven in up to 32000 steps. Finally, we point out possible directions of future work.

- Verification of memory managements CVM primitives `cvm_alloc` and `cvm_free`. They share data structures with the demand paging code and hence one can re-use the valid abstraction relations of the PFH state in different semantics as well as equivalence proofs between them. This should significantly ease the verification effort.

- In the present work we focused on the integration of demand paging into CVM and because of that assumed that the caller might request the page-fault handler to guarantee the simultaneous presence of at most two specified pages in the physical memory. However, in the implementation the corresponding parameter `count` is not limited by such means but rather by the total number of user-available physical pages. One might adapt the page-fault handler specification and proofs to support the latter.

- This thesis considers a relatively simple page-fault handler which implements a FIFO page replacement strategy. Among its disadvantages is the fact that it only considers the swap-in time and omits to keep track on the access frequency for pages while in main memory. A good method to approximate the optimal page replacement algorithm is so-called *FIFO with second chance* [Tan01] which itself approximates a better but rather computationally expensive *least recently used* (LRU) strategy. A page-fault handler implementing a FIFO with second chance has been developed within Verisoft [Con06]. This implementation is a good candidate to illustrate formal verification of more advanced demand paging software. Since the advanced implementation only extends the page-fault handler state with an additional page management list and some variables the core parts of specifications and proofs presented in this thesis might be reused.

- Transferring of correctness results from Simpl to C0 big-step semantics requires defining many variable lookup functions as well as the state abstraction relation for concrete program states. One might add features to the code translation tool for generation of such functions automatically.

- Although we used mostly interactive verification techniques we believe there is a room for automation. One might profit from methods of automated verification while proving functional correctness of the source code. As yet, we have unsatisfactory experience in application of a software model checking tool [DMSS05] to justify expression guards in the Hoare logic proofs: those guards that were proven by the tool automatically could alike be proven by the built-in Isabelle simplifier.

Appendix A: Macros from the Implementation

```
 1  #define PX(x)           ((x) >> 12u)
 2  #define BX(x)           ((x) & 4095u)
 3
 4
 5
 6  #define BPX(page)       ((page) >> 10u)
 7  #define BBX(page)       ((page) & 1023u)
 8
 9
10
11  #define BPTE(pid,bpx)   bpt[unsigned(pcb[pid].bpto) + bpx]
12  #define PTO_PT(pid)    (((unsigned(pcb[pid].ef[PTO]) << 12u) - PT_START)\
13                                                  / (PTES_PER_PG * 4u))
14  #define PTE(pid,page)  ((*pt)[(PTO_PT(pid) + (page / PTES_PER_PG))]\
15                             [(page & (PTES_PER_PG - 1u))])
16
17
18
19  #define VALID_MASK     2048u
20  #define INVALID_MASK   (~VALID_MASK)
21  #define VALID(x)       (((x) & VALID_MASK) != 0u)
22  #define PROT_MASK      1024u
23  #define PROT(x)        (((x) & PROT_MASK) != 0u)
24  #define EXEC_MASK      512u
25  #define PPX_MASK       4294963200u
```

Appendix B: Loop Invariants and Ranking Functions

Initialization code

First Loop

$$
\begin{aligned}
&INV^1_{\text{init}}?(c_{\text{H}}) = \\
&\quad c_{\text{H}}.kheap_{\text{glob}} = \texttt{PT\_START} \\
&\wedge\ |c_{\text{H}}.pcb\text{-}bpto_{\text{glob}}| = \texttt{MAX\_PID} \wedge |c_{\text{H}}.pcb\text{-}bptl_{\text{glob}}| = \texttt{MAX\_PID} \\
&\wedge\ |c_{\text{H}}.pcb\text{-}ef_{\text{glob}}| = \texttt{MAX\_PID} \wedge (\forall\ i < \texttt{MAX\_PID} : |c_{\text{H}}.pcb\text{-}ef_{\text{glob}}[i]| = \texttt{EF\_DIM}) \\
&\wedge\ |c_{\text{H}}.ppx2pd_{\text{glob}}| = \texttt{TOT\_PHYS\_PGS} \wedge |c_{\text{H}}.bpfree_{\text{glob}}| = \texttt{TOT\_BIG\_PGS} \\
&\wedge\ pt\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}) \wedge bpt\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}) \\
&\wedge\ init\text{-}pcb\text{-}ef?(c_{\text{H}}) \\
&\wedge\ 0 < c_{\text{H}}.n_{\text{loc}} \leq \texttt{MAX\_PID} \\
&\wedge\ (\forall\ 0 < i < c_{\text{H}}.n_{\text{loc}} : c_{\text{H}}.pcb\text{-}ef_{\text{glob}}[i][\texttt{PTO}] = 1024 + (i-1) \cdot 9 \\
&\qquad\qquad \wedge\ c_{\text{H}}.pcb\text{-}ef_{\text{glob}}[i][\texttt{PTL}] = -1 \\
&\qquad\qquad \wedge\ c_{\text{H}}.pcb\text{-}bpto_{\text{glob}}[i] = 0 \wedge c_{\text{H}}.pcb\text{-}bptl_{\text{glob}}[i] = -1) \\
&\wedge\ c_{\text{H}}.alloc = [1] \wedge c_{\text{H}}.free \geq \texttt{USER\_PGS} \cdot \texttt{PD\_SZ} \\
&\wedge\ c_{\text{H}}.x = zero\text{-}fill\text{-}page(\texttt{ZFP}, c^0_{\text{H}}.x) \\
&\wedge\ pres(\{c_{\text{H}}.bpt_{\text{glob}}\})
\end{aligned}
$$

$$RANK^1_{\text{init}}(c_{\text{H}}) = \texttt{MAX\_PID} - c_{\text{H}}.n_{\text{loc}}$$

Second Loop

$$
\begin{aligned}
&INV^2_{\text{init}}?(c_{\text{H}}) = \\
&c_{\text{H}}.kheap_{\text{glob}} = \texttt{PT\_START} \\
\wedge\ &|c_{\text{H}}.ppx2pd_{\text{glob}}| = \texttt{TOT\_PHYS\_PGS} \wedge |c_{\text{H}}.bpfree_{\text{glob}}| = \texttt{TOT\_BIG\_PGS} \\
\wedge\ &active\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}, [\,]) \\
\wedge\ &pt\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}) \wedge bpt\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}) \\
\wedge\ &pto\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \wedge ptl\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \\
\wedge\ &bpto\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \wedge bptl\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \\
\wedge\ &init\text{-}pcb\text{-}ef?(c_{\text{H}}) \\
\wedge\ &\texttt{KERNEL\_PGS} < c_{\text{H}}.n_{\text{loc}} \leq \texttt{TOT\_PHYS\_PGS} \\
\wedge\ &(\exists\, q : dlist_{\text{H}}?(c_{\text{H}}.free_{\text{glob}}, c_{\text{H}}.next_{\text{heap}}, c_{\text{H}}.prev_{\text{heap}}, q, [c_{\text{H}}.n_{\text{loc}} - \texttt{KERNEL\_PGS} + 1..2]) \\
&\qquad \wedge (\forall\, i < c_{\text{H}}.n_{\text{loc}} - \texttt{KERNEL\_PGS} : c_{\text{H}}.pid_{\text{heap}}(c_{\text{H}}.n_{\text{loc}} - \texttt{KERNEL\_PGS} - i + 1) = 0 \\
&\qquad\qquad\qquad \wedge\ c_{\text{H}}.vpx_{\text{heap}}(c_{\text{H}}.n_{\text{loc}} - \texttt{KERNEL\_PGS} - i + 1) = 0 \\
&\qquad\qquad\qquad \wedge\ c_{\text{H}}.ppx_{\text{heap}}(c_{\text{H}}.n_{\text{loc}} - \texttt{KERNEL\_PGS} - i + 1) = n - i - 1) \\
&\qquad \wedge (\forall\, \texttt{KERNEL\_PGS} \leq i < c_{\text{H}}.n_{\text{loc}} : c_{\text{H}}.ppx2pd_{\text{glob}}[i] = 2 + i - \texttt{KERNEL\_PGS})) \\
\wedge\ &c_{\text{H}}.alloc = [c_{\text{H}}.n_{\text{loc}} - \texttt{KERNEL\_PGS} + 1..1] \\
\wedge\ &\texttt{TOT\_PHYS\_PGS} \cdot \texttt{PD\_SZ} \leq c_{\text{H}}.free + c_{\text{H}}.n_{\text{loc}} \cdot \texttt{PD\_SZ} \\
\wedge\ &c_{\text{H}}.x = zero\text{-}fill\text{-}page(\texttt{ZFP}, c^0_{\text{H}}.x) \\
\wedge\ &pres(\{c_{\text{H}}.bpt_{\text{glob}}\})
\end{aligned}
$$

$$RANK^2_{\text{init}}(c_{\text{H}}) = \texttt{TOT\_PHYS\_PGS} - c_{\text{H}}.n_{\text{loc}}$$

Third Loop

$$
\begin{aligned}
&INV^3_{\text{init}}?(c_{\text{H}}) = \\
&c_{\text{H}}.kheap_{\text{glob}} = \texttt{PT\_START} \\
\wedge\ &|c_{\text{H}}.bpfree_{\text{glob}}| = \texttt{TOT\_BIG\_PGS} \\
\wedge\ &active\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}, [\,]) \wedge free\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}, [\texttt{USER\_PGS} + 1..2]) \\
\wedge\ &pt\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}) \wedge bpt\text{-}abs_{\text{H}}?(c_{\text{H}}, init\text{-}c_{\text{PFH}}) \\
\wedge\ &pages\text{-}free\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \\
\wedge\ &pto\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \wedge ptl\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \\
\wedge\ &bpto\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \wedge bptl\text{-}abs_{\text{H}}?(c_{\text{H}}, c_{\text{PFH}}) \\
\wedge\ &init\text{-}pcb\text{-}ef?(c_{\text{H}}) \\
\wedge\ &0 < c_{\text{H}}.n_{\text{loc}} \leq \texttt{TOT\_BIG\_PGS} \\
\wedge\ &(\forall\, i < c_{\text{H}}.bpfree_{\text{glob}}[i] = \texttt{TOT\_BIG\_PGS} - 1 - i) \\
\wedge\ &c_{\text{H}}.alloc = [\texttt{USER\_PGS} + 1..1] \\
\wedge\ &c_{\text{H}}.x = zero\text{-}fill\text{-}page(\texttt{ZFP}, c^0_{\text{H}}.x) \\
\wedge\ &pres(\{c_{\text{H}}.bpt_{\text{glob}}\})
\end{aligned}
$$

$$RANK^3_{\text{init}}(c_{\text{H}}) = \texttt{TOT\_BIG\_PGS} - c_{\text{H}}.n_{\text{loc}}$$

Page-Fault Handler

First Loop

$$
\begin{aligned}
&INV^1_{\text{ta}}?(c_{\text{H}}, c_{\text{PFH}}, \mathit{refs}_{\text{active}}, \mathit{refs}_{\text{free}}) = \\
&\quad c^0_{\text{H}}.\mathit{count}_{\text{loc}} = 1 \wedge c_{\text{H}}.\mathit{count}_{\text{loc}} \leq 1 \\
&\wedge\ c_{\text{H}}.\mathit{pages\text{-}free}_{\text{glob}} = 0 \\
&\wedge\ (c_{\text{H}}.\mathit{count}_{\text{loc}} = 1 \longrightarrow c_{\text{H}}.\mathit{vict}_{\text{loc}} = c_{\text{H}}.\mathit{active}_{\text{glob}}) \\
&\wedge\ (c_{\text{H}}.\mathit{count}_{\text{loc}} = 0 \longrightarrow c_{\text{H}}.\mathit{vict}_{\text{loc}} = c_{\text{H}}.\mathit{next}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) \\
&\qquad\qquad \wedge\ (c_{\text{H}}.\mathit{vpx}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) \neq \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\qquad\qquad \vee\ c_{\text{H}}.\mathit{pid}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) \neq c_{\text{H}}.\mathit{pid}_{\text{loc}})) \\
&\wedge\ c_{\text{H}}.\mathit{vpx}_{\text{loc}} = \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\wedge\ c_{\text{H}}.\mathit{intent}_{\text{loc}} \neq \texttt{SWAP\_IN} \\
&\wedge\ c_{\text{H}}.\mathit{pte}_{\text{loc}} = \\
&\quad c_{\text{H}}.\mathit{pt}_{\text{heap}}(c_{\text{H}}.\mathit{pt}_{\text{glob}})[(c_{\text{H}}.\mathit{pcb\text{-}ef}_{\text{glob}}[c_{\text{H}}.\mathit{pid}_{\text{loc}}][\texttt{PTO}] \cdot \texttt{PG\_SZ} - \texttt{PT\_START})/\texttt{PG\_SZ} \\
&\qquad\qquad + \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}})/\texttt{PTES\_PER\_PG}][\mathit{ptea}_2(c_{\text{H}}.\mathit{addr}_{\text{loc}})] \\
&\wedge\ \mathit{user\text{-}pid}?(c_{\text{H}}.\mathit{pid}_{\text{loc}}) \wedge c_{\text{H}}.\mathit{addr}_{\text{loc}} < \texttt{TOT\_PGS} \cdot \texttt{PG\_SZ} \wedge c_{\text{PFH}}.\mathit{ptl}[c_{\text{H}}.\mathit{pid}_{\text{loc}}] \geq 0 \\
&\wedge\ \neg \mathit{ptlexcp}_{\text{PFH}}?(c_{\text{PFH}}, c_{\text{H}}.\mathit{pid}_{\text{loc}}, c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\wedge\ \neg \mathit{pf}_{\text{PFH}}?(c_{\text{PFH}}, c_{\text{H}}.\mathit{pid}_{\text{loc}}, c_{\text{H}}.\mathit{addr}_{\text{loc}}, c_{\text{H}}.\mathit{intent}_{\text{loc}}) \\
&\wedge\ \mathit{abs}_{\text{H}}\surd(c_{\text{H}}, c_{\text{PFH}}, \mathit{refs}_{\text{active}}, \mathit{refs}_{\text{free}}) \\
&\wedge\ c_{\text{H}}.x = c^0_{\text{H}}.x \\
&\wedge\ \mathit{pres}(\{c_{\text{H}}.\mathit{active}_{\text{glob}}, c_{\text{H}}.\mathit{free}_{\text{glob}}, c_{\text{H}}.\mathit{bpfree}_{\text{glob}}, c_{\text{H}}.\mathit{ppx2pd}_{\text{glob}}, c_{\text{H}}.\mathit{pt}_{\text{glob}}, \\
&\qquad c_{\text{H}}.\mathit{bpt}_{\text{glob}}, c_{\text{H}}.\mathit{pages\text{-}used}_{\text{glob}}, c_{\text{H}}.\mathit{pages\text{-}free}_{\text{glob}}, c_{\text{H}}.\mathit{bpages\text{-}free}_{\text{glob}}, \\
&\qquad c_{\text{H}}.\mathit{pcb\text{-}ef}_{\text{glob}}, c_{\text{H}}.\mathit{pcb\text{-}bpto}_{\text{glob}}, c_{\text{H}}.\mathit{pcb\text{-}bpto}_{\text{glob}}, c_{\text{H}}.\mathit{pcb\text{-}bptl}_{\text{glob}}, \\
&\qquad c_{\text{H}}.\mathit{pid}_{\text{loc}}, c_{\text{H}}.\mathit{addr}_{\text{loc}}, c_{\text{H}}.\mathit{intent}_{\text{loc}}, \\
&\qquad c_{\text{H}}.\mathit{pt}_{\text{heap}}, c_{\text{H}}.\mathit{pid}_{\text{heap}}, c_{\text{H}}.\mathit{vpx}_{\text{heap}}, c_{\text{H}}.\mathit{ppx}_{\text{heap}}, c_{\text{H}}.\mathit{next}_{\text{heap}}, c_{\text{H}}.\mathit{prev}_{\text{heap}}\})
\end{aligned}
$$

$$RANK^1_{\text{ta}}(c_{\text{H}}) = c_{\text{H}}.\mathit{count}_{\text{loc}}$$

Second Loop

$$
\begin{aligned}
&INV^2_{\text{ta}}?(c_{\text{H}}, c_{\text{PFH}}, \mathit{refs}_{\text{active}}, \mathit{refs}_{\text{free}}) = \\
&\quad c^0_{\text{H}}.\mathit{count}_{\text{loc}} = 1 \wedge c_{\text{H}}.\mathit{count}_{\text{loc}} \leq 1 \\
&\wedge\ c_{\text{H}}.\mathit{pages\text{-}free}_{\text{glob}} = 0 \\
&\wedge\ (c_{\text{H}}.\mathit{count}_{\text{loc}} = 1 \longrightarrow c_{\text{H}}.\mathit{active\text{-}tail}_{\text{loc}} = c_{\text{H}}.\mathit{active}_{\text{glob}} \\
&\qquad\qquad \wedge\ c_{\text{H}}.\mathit{vpx}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) = \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\qquad\qquad \wedge\ c_{\text{H}}.\mathit{pid}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) = c_{\text{H}}.\mathit{pid}_{\text{loc}}) \\
&\wedge\ (c_{\text{H}}.\mathit{count}_{\text{loc}} = 0 \longrightarrow c_{\text{H}}.\mathit{active\text{-}tail}_{\text{loc}} = c_{\text{H}}.\mathit{next}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}})) \\
&\wedge\ (c_{\text{H}}.\mathit{vpx}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) = \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\qquad \wedge\ c_{\text{H}}.\mathit{pid}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) = c_{\text{H}}.\mathit{pid}_{\text{loc}} \\
&\quad \longrightarrow c_{\text{H}}.\mathit{vict}_{\text{loc}} = c_{\text{H}}.\mathit{active}_{\text{glob}}) \\
&\wedge\ (c_{\text{H}}.\mathit{vpx}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) \neq \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\qquad \vee\ c_{\text{H}}.\mathit{pid}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) \neq c_{\text{H}}.\mathit{pid}_{\text{loc}} \\
&\quad \longrightarrow c_{\text{H}}.\mathit{vict}_{\text{loc}} = c_{\text{H}}.\mathit{next}_{\text{heap}}(c_{\text{H}}.\mathit{active}_{\text{glob}}) \\
&\wedge\ c_{\text{H}}.\mathit{vpx}_{\text{loc}} = \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\wedge\ c_{\text{H}}.\mathit{intent}_{\text{loc}} \neq \texttt{SWAP\_IN} \\
&\wedge\ c_{\text{H}}.\mathit{pte}_{\text{loc}} = \\
&\quad c_{\text{H}}.\mathit{pt}_{\text{heap}}(c_{\text{H}}.\mathit{pt}_{\text{glob}})[(c_{\text{H}}.\mathit{pcb\text{-}ef}_{\text{glob}}[c_{\text{H}}.\mathit{pid}_{\text{loc}}][\texttt{PTO}] \cdot \texttt{PG\_SZ} - \texttt{PT\_START})/\texttt{PG\_SZ} \\
&\qquad\qquad + \mathit{px}(c_{\text{H}}.\mathit{addr}_{\text{loc}})/\texttt{PTES\_PER\_PG}][\mathit{ptea}_2(c_{\text{H}}.\mathit{addr}_{\text{loc}})] \\
&\wedge\ \mathit{user\text{-}pid}?(c_{\text{H}}.\mathit{pid}_{\text{loc}}) \wedge c_{\text{H}}.\mathit{addr}_{\text{loc}} < \texttt{TOT\_PGS} \cdot \texttt{PG\_SZ} \wedge c_{\text{PFH}}.\mathit{ptl}[c_{\text{H}}.\mathit{pid}_{\text{loc}}] \geq 0 \\
&\wedge\ \neg \mathit{ptlexcp}_{\text{PFH}}?(c_{\text{PFH}}, c_{\text{H}}.\mathit{pid}_{\text{loc}}, c_{\text{H}}.\mathit{addr}_{\text{loc}}) \\
&\wedge\ \neg \mathit{pf}_{\text{PFH}}?(c_{\text{PFH}}, c_{\text{H}}.\mathit{pid}_{\text{loc}}, c_{\text{H}}.\mathit{addr}_{\text{loc}}, c_{\text{H}}.\mathit{intent}_{\text{loc}}) \\
&\wedge\ \mathit{abs}_{\text{H}}\surd(c_{\text{H}}, c_{\text{PFH}}, \mathit{refs}_{\text{active}}, \mathit{refs}_{\text{free}}) \\
&\wedge\ c_{\text{H}}.x = c^0_{\text{H}}.x \\
&\wedge\ \mathit{pres}(\{c_{\text{H}}.\mathit{active}_{\text{glob}}, c_{\text{H}}.\mathit{free}_{\text{glob}}, c_{\text{H}}.\mathit{bpfree}_{\text{glob}}, c_{\text{H}}.\mathit{ppx2pd}_{\text{glob}}, c_{\text{H}}.\mathit{pt}_{\text{glob}}, \\
&\qquad c_{\text{H}}.\mathit{bpt}_{\text{glob}}, c_{\text{H}}.\mathit{pages\text{-}used}_{\text{glob}}, c_{\text{H}}.\mathit{pages\text{-}free}_{\text{glob}}, c_{\text{H}}.\mathit{bpages\text{-}free}_{\text{glob}}, \\
&\qquad c_{\text{H}}.\mathit{pcb\text{-}ef}_{\text{glob}}, c_{\text{H}}.\mathit{pcb\text{-}bpto}_{\text{glob}}, c_{\text{H}}.\mathit{pcb\text{-}bpto}_{\text{glob}}, c_{\text{H}}.\mathit{pcb\text{-}bptl}_{\text{glob}}, \\
&\qquad c_{\text{H}}.\mathit{pid}_{\text{loc}}, c_{\text{H}}.\mathit{addr}_{\text{loc}}, c_{\text{H}}.\mathit{intent}_{\text{loc}}, \\
&\qquad c_{\text{H}}.\mathit{pt}_{\text{heap}}, c_{\text{H}}.\mathit{pid}_{\text{heap}}, c_{\text{H}}.\mathit{vpx}_{\text{heap}}, c_{\text{H}}.\mathit{ppx}_{\text{heap}}, c_{\text{H}}.\mathit{next}_{\text{heap}}, c_{\text{H}}.\mathit{prev}_{\text{heap}}\})
\end{aligned}
$$

$$
RANK^2_{\text{ta}}(c_{\text{H}}) = c_{\text{H}}.\mathit{count}_{\text{loc}}
$$

Appendix C: Mapping to Formal Names in Isabelle/HOL

Name	Formal name in Isabelle/HOL
Definition 2.2	VAMPasm2isaSystem/config_correct::is_dlx_conft
Definition 2.4	VAMPisa/mem_spec::compute_pa
Definition 2.7	VAMPisa/dlxifspec::ipf
Definition 2.8	VAMPisa/dlxifspec::dpf
Definition 2.9	VAMPasm/Config::is_ASMcore
Definition 2.10	VAMPisaDevices/dlxifspec_dev_hd::is_valid_idle_hd
Definition 2.11	VAMPisaDevices/dlxifspec_dev_hd::swap_disk_exists
Definition 2.12	VAMPisaDevices/dlxifspec_dev_hd::{live_input_seq_isa, efis_welltyped}
Definition 2.13	VAMPisaDevices/dlxifspec_dev_hd::proc_step_number
Definition 2.14	VAMPisaDevices/dlxifspec_dev_hd::dev_step_times, dev_step_number
Theorem 2.16	C0SS2VAMPisaSystem/mini_stack_wo_dev::C0_mini_stack_isa
Definition 2.17	cvm/config/c0_config::{linked_tt, linked_pt, linked_st}
Definition 2.18	cvm/map/B_relation::get_p_vm
Definition 2.19	cvm/map/B_relation::{page_index, byte_index}
Definition 2.20	cvm/map/B_relation::{page_table_origin, page_table_length}
Definition 2.21	cvm/map/B_relation::page_table_entry
Definition 2.22	cvm/map/B_relation::physical_memory_address
Definition 2.23	cvm/map/B_relation::{big_page_index, big_byte_index}
Definition 2.24	cvm/map/B_relation::big_page_table_origin
Definition 2.25	cvm/map/B_relation::big_page_table_entry
Definition 2.26	cvm/map/B_relation::swap_memory_address
Definition 2.27	cvm/map/B_relation::valid_bit
Definition 2.28	cvm/map/B_relation::get_mm
Definition 2.29	cvm/map/B_relation::ASMcore_equality
Definition 2.30	cvm/map/B_relation:B_relation
Definition 2.31	cvm/config/isa_config::code_invariant_isa
Definition 2.32	pfh/Transfer/zfpCondition::zfp_condition
Theorem 2.33	cvm/cvm_correct/cvm_correct::cvm_correct
Definition 3.1	C0BS/ConformT::conforms
Theorem 3.2	HoareToBigStep/HoareToBigStep::validt_hoare_to_C0_aux
Theorem 3.4	C0BSSSequiv/PropertyTransferBStoSS:: ultimate_valid_bs_to_valid_ss_transfer_total
Theorem 3.5	C0BSSSequiv/PropertyTransferBStoSS::adapt_hoare_redex'
Definition 4.1	pfh/Validity::subtyping_descriptor

Definition 4.2	pfh/Validity::{subtyping_active, subtyping_free}
Definition 4.3	pfh/Validity::subtyping_bpfree_stack
Definition 4.4	pfh/Validity::subtyping_ptspace
Definition 4.5	pfh/Validity::subtyping_bptspace
Definition 4.6	pfh/Validity::subtyping_pcbs
Definition 4.7	pfh/Validity::subtyping_pfh
Definition 4.8	pfh/pfhX::valid_x
Definition 4.9	pfh/Utilities::ptspace_origin
Definition 4.10	pfh/Utilities::ptspace_length
Definition 4.11	pfh/Utilities::pte_dim1
Definition 4.12	pfh/Utilities::pte_dim2
Definition 4.13	pfh/Utilities::ptspace_entry
Definition 4.14	pfh/Utilities::translate_addr
Definition 4.15	pfh/Utilities::ptl_excp
Definition 4.16	pfh/Utilities::invalid_access
Definition 4.17	pfh/Utilities::zero_protection
Definition 4.18	pfh/Utilities::page_fault
Definition 4.19	pfh/Utilities::bpx_of_vpx
Definition 4.20	pfh/Utilities::bbx_of_vpx
Definition 4.23	pfh/Utilities::compute_disk_addr
Definition 4.24	pfh/Utilities::adjust_swap_addr
Definition 4.25	pfh/Utilities::adjust_mem_addr
Definition 4.26	pfh/pfhX::read_from_disk
Definition 4.27	pfh/pfhX::swap_in
Definition 4.28	pfh/pfhX::write_to_disk
Definition 4.29	pfh/pfhX::swap_out
Definition 4.30	pfh/pfhX::zero_fill_page
Def. 4.31,4.32,4.33	pfh/Configurations::abs_pfh_after_handling
Definition 4.34	pfh/pfhX::fill_page
Definition 4.35	pfh/pfhX::pfh_touch_addr_POST_modifying_X_state
Definition 4.36	pfh/Configurations::abs_pfh_after_handling_non_pf
Definition 4.37	pfh/Validity::is_used_pid_active
Definition 4.38	pfh/Validity::{valid_ppx_active, valid_ppx_free}
Definition 4.39	pfh/Validity::valid_bit_active
Definition 4.40	pfh/Validity::not_protection_bit_active
Definition 4.41	pfh/Validity::page_index_active
Definition 4.42	pfh/Validity::{ppx_active_not_zfp, ppx_free_not_zfp}
Definition 4.43	pfh/Validity::distinct_pid_vpx_active
Definition 4.44	pfh/Validity::distinct_ppx_active_free
Definition 4.45	pfh/Validity::vpx_inside_ptl_active
Definition 4.46	pfh/Validity::active_describes_valid_pte
Definition 4.47	pfh/Validity::valid_ppx_ptspace
Definition 4.48	pfh/Validity::pt_not_overlap
Definition 4.49	pfh/Validity::pto_plus_ptl_inside_ptspace
Definition 4.50	pfh/Validity::ppx_zfp_protected
Definition 4.51	pfh/Validity::pte_valid_exec
Definition 4.52	pfh/Validity::ppx_zfp_valid
Definition 4.53	pfh/Validity::number_of_bpages_eq_free_and_used
Definition 4.54	pfh/Validity::bpte_distinct
Definition 4.55	pfh/Validity::bpto_bptl_correct

Definition 4.56	pfh/Validity::pto_mono
Definition 4.57	pfh/Validity::ptl_pid_alloc_active
Definition 4.58	pfh/Validity::pto_ptl_correct
Definition 4.59	pfh/Validity::bptl_correct
Definition 4.60	pfh/Validity::valid_pfh_pcbs
Theorem 4.61	pfh/pfhValidityLemmas::valid_init_pfh_init_pcbs
Theorem 4.62	pfh/pfhValidityLemmas::valid_pfh_pcbs_abs_pfh_after_handling
Theorem 4.63	pfh/pfhValidityLemmas::valid_pfh_pcbs_abs_pfh_after_handling_non_pf
Definition 5.1	Hoare/HeapList::List
Definition 5.2	dList/HeapdList::dList
Definition 5.3	pfh/Simpl/simplMapping::dList_of_pds_map
Definition 5.4	pfh/Simpl/simplMapping::active_map
Definition 5.5	pfh/Simpl/simplMapping::free_map
Definition 5.6	pfh/Simpl/simplMapping::bpfree_stack_map
Definition 5.7	pfh/Simpl/simplMapping::ptspace_map
Definition 5.8	pfh/Simpl/simplMapping::bptspace_map
Definition 5.9	pfh/Simpl/simplMapping::ptos_map
Definition 5.10	pfh/Simpl/simplMapping::ptls_map
Definition 5.11	pfh/Simpl/simplMapping::bptos_map
Definition 5.12	pfh/Simpl/simplMapping::bptls_map
Definition 5.13	pfh/Simpl/simplMapping::number_free_pages_map
Definition 5.14	pfh/Simpl/simplMapping::free_big_pages_map
Definition 5.15	pfh/Simpl/simplMapping::pages_used_map
Definition 5.16	pfh/Simpl/simplMapping::ppx2pd_map
Definition 5.17	pfh/Simpl/ValidSimplMapping::pfh_pcbs_map
Definition 5.18	pfh/Simpl/ValidSimplMapping::valid_pfh_pcbs_map
Definition 5.19	pfh/Simpl/pfhInitSpec::initialized_ef_except_pto_ptl
Theorem 5.20	pfh/Simpl/pfhInitTotal::{pfh_init_spec, pfh_init_modifies}
Theorem 5.21	pfh/Simpl/pfhSwapInTotal::{pfh_swap_in_spec, pfh_swap_in_modifies}
Theorem 5.22	pfh/Simpl/pfhSwapOutTotal::{pfh_swap_out_spec, pfh_swap_out_modifies}
Definition 5.23	pfh/Utilities::no_ptl_excp_unless_swap_in
Theorem 5.25	pfh/Simpl/pfhTouchAddrTotal:: {pfh_touch_addr_spec, pfh_touch_addr_modifies}
Definition 6.1	pfh/Transfer/bsList::List$_{bs}$
Definition 6.2	pfh/Transfer/bsList::dList$_{bs}$
Definition 6.3	pfh/Transfer/bsMapping::dList_of_pds_map$_{bs}$
Definition 6.4	pfh/Transfer/bsMapping::active_map$_{bs}$
Definition 6.5	pfh/Transfer/bsMapping::free_map$_{bs}$
Definition 6.6	pfh/Transfer/bsMapping::bpfree_stack_map$_{bs}$
Definition 6.7	pfh/Transfer/bsMapping::ptspace_map$_{bs}$
Definition 6.8	pfh/Transfer/bsMapping::bptspace_map$_{bs}$
Definition 6.9	pfh/Transfer/bsMapping::ptos_map$_{bs}$
Definition 6.10	pfh/Transfer/bsMapping::ptls_map$_{bs}$
Definition 6.11	pfh/Transfer/bsMapping::bptos_map$_{bs}$
Definition 6.12	pfh/Transfer/bsMapping::bptls_map$_{bs}$
Definition 6.13	pfh/Transfer/bsMapping::number_free_pages_map$_{bs}$
Definition 6.14	pfh/Transfer/bsMapping::free_big_pages_map$_{bs}$
Definition 6.15	pfh/Transfer/bsMapping::pages_used_map$_{bs}$
Definition 6.16	pfh/Transfer/bsMapping::ppx2pd_map$_{bs}$
Definition 6.17	pfh/Transfer/bsMapping::pfh_pcbs_map$_{bs}$

Definition 6.18	pfh/Transfer/bsMapping::valid_pfh_pcbs_map$_{bs}$
Definition 6.19	HoareToBigStep/HoareToBigStep::abs$_{bs}$
Lemma 6.20	pfh/Transfer/MappingTransferSimplToBS:: valid_pfh_pcbs_map_impl_valid_pfh_pcbs_map$_{bs}$
Lemma 6.21	pfh/Transfer/MappingTransferBSToSimpl:: valid_pfh_pcbs_map$_{bs}$_impl_valid_pfh_pcbs_map
Definition 6.22	pfh/Transfer/pfhInitTransferSimplToBS::initial_heap_typing
Definition 6.23	pfh/Transfer/pfhCode::pfh_init_gm_changed_vars
Definition 6.24	pfh/Transfer/pfhCode::pfh_touch_addr_gm_changed_vars
Lemma 6.27	pfh/Transfer/pfhInitTransferSimplToBS::pfh_init_hoare_to_bs'
Lemma 6.30	pfh/Transfer/pfhTouchAddrTransferSimplToBS::pfh_touch_addr_hoare_to_bs''
Definition 6.31	pfh/xSem::HDdriverXCall_stmt
Definition 6.32	pfh/xSem::pt2xpt
Definition 6.33	pfh/pfhCode::pfh'bs'prog'
Lemma 6.35	C0BS/C0BSHoareValidity::c0validt_extend_program
Lemma 6.36	C0BS/C0BSHoareValidity::c0validt_conseq
Lemma 6.37	pfh/Transfer/pfhInitTransferSimplToBS::pfh_init_hoare_to_bs''''
Lemma 6.38	pfh/Transfer/pfhTouchAddrTransferSimplToBS::pfh_touch_addr_hoare_to_bs''''
Definition 6.39	pfh/Transfer/ssList::List$_{ss}$
Definition 6.40	pfh/Transfer/ssList::dList$_{ss}$
Definition 6.41	pfh/Transfer/ssMapping::dList_of_pds_map$_{ss}$
Definition 6.42	pfh/Transfer/ssMapping::active_map$_{ss}$
Definition 6.43	pfh/Transfer/ssMapping::free_map$_{ss}$
Definition 6.44	pfh/Transfer/ssMapping::bpfree_stack_map$_{ss}$
Definition 6.45	pfh/Transfer/ssMapping::ptspace_map$_{ss}$
Definition 6.46	pfh/Transfer/ssMapping::bptspace_map$_{ss}$
Definition 6.47	pfh/Transfer/ssMapping::ptos_map$_{ss}$
Definition 6.48	pfh/Transfer/ssMapping::ptls_map$_{ss}$
Definition 6.49	pfh/Transfer/ssMapping::bptos_map$_{ss}$
Definition 6.50	pfh/Transfer/ssMapping::bptls_map$_{ss}$
Definition 6.51	pfh/Transfer/ssMapping::number_free_pages_map$_{ss}$
Definition 6.52	pfh/Transfer/ssMapping::free_big_pages_map$_{ss}$
Definition 6.53	pfh/Transfer/ssMapping::pages_used_map$_{ss}$
Definition 6.54	pfh/Transfer/ssMapping::ppx2pd_map$_{ss}$
Definition 6.55	pfh/Transfer/ssMapping::pfh_pcbs_map$_{ss}$
Definition 6.56	pfh/Transfer/ssMapping::valid_pfh_pcbs_map$_{ss}$
Definition 6.57	C0BSSSequiv/ValueAbs::absVal
Definition 6.58	C0BSSSequiv/StateAbs::absState
Definition 6.59	pfh/Transfer/PfhBigStepToSmallStep::allocated_cvm_heap_in_heap
Lemma 6.60	pfh/Transfer/MappingTransferBSToSS:: valid_pfh_pcbs_map$_{bs}$_impl_valid_pfh_pcbs_map$_{ss}$
Definition 6.61	C0BSSSequiv/EquivStmt::absHeapType
Lemma 6.62	pfh/Transfer/MappingTransferSSToBS:: valid_pfh_pcbs_map$_{ss}$_impl_valid_pfh_pcbs_map$_{bs}$
Lemma 6.64	pfh/Transfer/pfhInitTransferBSToSS::pfh_init_bs_to_ss
Lemma 6.65	pfh/Transfer/pfhInitTransferBSToSS::pfh_init_ss_adapt
Lemma 6.68	pfh/Transfer/pfhTouchAddrTransferBSToSS::pfh_touch_addr_bs_to_ss
Lemma 6.69	pfh/Transfer/pfhTouchAddrTransferBSToSS::pfh_touch_addr_bs_to_ss_adapt
Definition 7.1	pfh/pfhX::{read_from_disk_PRE, write_to_disk_PRE}
Definition 7.2	pfh/xSem::XWtdSem'ss

Definition 7.3	pfh/xSem::XRfdSem'ss
Definition 7.4	pfh/pfhX::zero_fill_page_PRE
Definition 7.5	pfh/xSem::XZfpSem'ss
Definition 7.6	pfh/xSem::pfh_SS'xdefns
Definition 7.7	pfh/NoXCall/driverCorrectnessAsm::pfh_memConsis
Definition 7.8	pfh/NoXCall/driverCorrectnessAsm::pfh_swapConsis
Definition 7.9	pfh/NoXCall/driverCorrectnessAsm::driver_xConsis
Definition 7.10	pfh/NoXCall/driverCorrectnessAsm::driver_pt_xpt_xsem_relation
Theorem 7.11	pfh/NoXCall/driverCorrectnessAsm::XCall_driver_correct_asm
Theorem 7.12	pfh/NoXCall/driverCorrectnessIsa::XCall_driver_correct_isa
Definition 7.13	pfh/NoXCall/driverCorrectnessCvm::{mem_to_pfhX, swap_to_pfhX}
Theorem 7.16	pfh/NoXCall/pfhInitTopLevel::pfh_init_correct
Definition 7.19	cvm/map/B_relation::b_ASMcore_equality
Definition 7.20	cvm/map/B_relation::b_relation
Theorem 7.21	pfh/NoXCall/pfhTouchAddrTopLevel::pfh_touch_addr_correct
Lemma 7.22	cvm/config/pfh_conditions::ptspace_entry_plus_byte_index_diff

Bibliography

[ABP09] E. Alkassar, S. Bogan, and W. Paul. Proving the correctness of client/server software. 34:145–192, 2009.

[AHK+07] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev. Formal device and programming model for a serial interface. In B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, pages 4–20. CEUR-WS Workshop Proceedings, 2007.

[AHL+08] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The verisoft approach to systems verification. In *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.

[AHL+09] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. In *Journal of Automated Reasoning: Special Issue on Operating Systems Verification*. Springer, 2009.

[Alk09] Eyad Alkassar. *OS Verification Extended. On the Formal Verification of Device Drivers and the Correctness of Client/Server Software.* PhD thesis, Saarland University, Computer Science Department, 2009.

[ASS08] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In Juris Hartmanis Gerhard Goos and Jan van Leeuwen, editors, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[Bev87] W. R. Bevier. *A Verified Operating System Kernel.* PhD thesis, University of Texas at Austin, 1987.

[Bev89] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[Bey05] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP.* PhD thesis, Saarland University, Computer Science Department, March 2005.

[BHMY89a] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An approach to systems verification. 5(4):411–428, December 1989.

[BHMY89b] W.R. Bevier, W.A. Hunt, J. Strother Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[BHW06] Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification — experiences from the verisoft email client. In *Proceedings of the Workshop on Empirical Succesfully Computerized Reasoning (ESCoR 2006)*, 2006.

[BJK+03] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W.J. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the vamp. In *CHARME 2003*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BJK+06] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006.

[Bog08] Sebastian Bogan. *Formal Specification of a Simple Operating System.* PhD thesis, Saarland University, Computer Science Department, 2008.

[Bur72] R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.

[CKS08] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Mu noz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 2008.

[Con06] Cosmin Condea. Design and implementation of a page-fault handler in c0. Master's thesis, Saarland University, 2006.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units.* PhD thesis, Saarland University, Computer Science Department, July 2006.

[DDB08] Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70, 2008.

[DDSW08] Matthias Daum, Jan Dörrenbächer, Mareike Schmidt, and Burkhart Wolff. A verification approach for system-level concurrent programs. In *Verified Software: Theories, Tools, and Experiments*, volume 5295/2008 of *LNCS*, pages 161–176. Springer, 2008.

[DDW09] Matthias Daum, Jan Drrenbcher, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Journal of Automated Reasoning: Special Issue on Operating System Verification*. Springer, 2009.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725, pages 301–316. Springer, 2005.

[DMSS05] M. Daum, S. Maus, N. Schirmer, and M.N. Seghir. Integration of a software model checker into isabelle. In *LPAR*, volume 3835 of *LNCS*, pages 381–395. Springer, 2005.

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In Jim Woodcock and Natarajan Shankar, editors, *Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2008)*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, October 2008. Springer.

[End05] Endrawaty. Verification of the fiasco IPC implementation, 2005.

[FN79] R. Feiertag and P. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the National Computer Conference 48*, pages 329–334, 1979.

[FSDG08] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, New York, NY, USA, June 2008. ACM.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603, pages 1–16. Springer, 2005.

[HEK+07] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S.M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *Operating Systems Review*, 41(4):3–11, 2007.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Computer Science Department, June 2005.

[HIP05] Mark Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD '05*, pages 309–316. IEEE Computer Society, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HP07] M. A. Hillebrand and W. J. Paul. On the architecture of system verification environments. In *Haifa Verification Conference 2007, October 23-25, 2007, Haifa, Israel*, LNCS. Springer, 2007.

[HT03] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. Technical Report No. 187 Institut für Informatik Universität Freiburg, url = citeseer.ist.psu.edu/article/hohmuth03semantics.html.

[HT05] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *2nd ECOOP Workshop on Program Languages and Operating Systems (ECOOP-PLOS'05)*, 2005.

[IdRT08] T. In der Rieden and A. Tsyban. Cvm - a verified framework for microkernel programmers. In *3rd International Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[In 09] Tomas In der Rieden. *Verifying CVM - The Kernel Parts. To appear.* PhD thesis, Saarland University, Computer Science Department, 2009.

[JSM04] E. Northup S. Sridhar J. Shapiro, M. S. Doerrie and M. Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification*, October 2004.

[KDE09] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: sel4 — formally verifying a high-performance microkernel. In *Proc. 2009 ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 91–96. ACM, August 2009.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[KELS62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, 11(2):223–235, 1962.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.

[KHPS61] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner. Atlas operating system part i: Internal organization. *The Computer Journal*, 4(3):222–225, 1961.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[Kna08] Steffen Knapp. *The Correctness of a Distributed Real-Time System*. PhD thesis, Saarland University, Computer Science Department, 2008.

[KR68] C.J. Kuehner and B. Randell. Demand paging in perspective. In *Fall Joint Computer Conference*, 1968.

[Lei07] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Department, 2007.

[LP08] D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Bernhard Aichernig and Bernhard Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5–9 September 2005, Koblenz, Germany*, pages 2–11, 2005.

[MN03] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741, pages 121–135. Springer, 2003.

[Moo03] J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757, pages 161–172. Springer, 2003.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NBF+80] P.G. Neumann, R.S Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. *A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116*. Computer Science Laboratory, SRI International, Menlo Park, California, May 1980.

[NF03] P. Neumann and R. Feiertag. PSOS revisited. In *19th Annual Computer Security Applications Conference*, 2003.

[Ngu05] V.G. Nguiekom. Verifikation von doppelt verketteten Listen auf Pointerebene. Master's thesis, Saarland University, 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002.

[NS06] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 320–333, New York, NY, USA, 2006. ACM.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using xcap to certify realistic systems code: Machine context management. In *TPHOLs*, pages 189–206, 2007.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, May 2007.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, volume 3452, pages 398–414. Springer, 2005.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, Institut für Informatik, 2006.

[Sha06] Andrey Shadrin. Design and implementation of the portmapper and rpc primitives in the context of the sos. Master's thesis, Saarland University, 2006.

[Smi78] Alan Jay Smith. Bibliography on paging and related topics. *SIGOPS Oper. Syst. Rev.*, 12(4):39–56, 1978.

[ST08a] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In *3rd International Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[ST08b] Artem Starostin and Alexandra Tsyban. Verified process-context switch for c-programmed kernels. In Natarajan Shankar and Jim Woodcock, editors, *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 240–254. Springer, 2008.

[Sta06] Artem Starostin. Formal verification of a c-library for strings. Master's thesis, Saarland University, 2006.

[SW00] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *IEEE Symposium on Security and Privacy*, pages 166–176, May 2000.

[Tan97] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation (Second Edition)*. Prentice-Hall, 1997.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems (Second Edition)*. Prentice-Hall, 2001.

[Tsy09] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmes*. PhD thesis, Saarland University, Computer Science Department, 2009.

[Tve09] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Computer Science Department, 2009.

[WKP79] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 64–65, New York, NY, USA, 1979. ACM.

Index