Verification of Programs in Virtual Memory Using Separation Logic

Rafal Kolanski

Ph.D.

2011

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation, and linguistic expression is acknowledged.'

Signed .....

Date .....

Copyright Statement

Signed .....

Date .....

Authenticity Statement

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed .....

Date .....

Abstract

Formal reasoning about programs executing in virtual memory is a difficult problem, as it is an environment in which writing to memory can change its layout. At the same time, correctly reasoning about virtual memory is essential to operating system verification, a field we are very much interested in. Current approaches rely on entering special modes or making high-level assertions about the nature of virtual memory which may or may not be correct.

In this thesis, we examine the problems created by virtual memory and develop a unified view of memory, both physical and virtual, based on separation logic. We first develop this model for a simple programming language on a simplified architecture with a one-level page table, taking care to prove it constitutes a separation logic. We then extend the framework to deal with low-level C programs executing in a virtual memory environment of the ARMv6 architecture with a two-level page table. We perform two case studies involving mapping in of a new page into the current address space: first for the simple version of our logic, and finally for our full framework. The case studies demonstrate that separation logic style modular reasoning via the frame rule can be used in a unified model which encompasses virtual memory, even in the presence of page table writes.

To our knowledge, we present the first model offering a unified view of virtual and physical memory, the first separation logic involving an address translation mechanism, as well as the first published model of a functional subset of ARM memory management unit. Our memory models, framework, proofs and all results are formalised in the Isabelle/HOL interactive theorem prover.

Acknowledgements

First and foremost, I wish to thank my supervisor, Gerwin Klein, for his continued advice, encouragement and support over the years. His friendly nature, Zen-like calmness and amazing work ethic have been a source of inspiration to me, and this work definitely could not have happened without him.

I would also like to thank the ERTOS and L4.verified teams at NICTA for the best work atmosphere one could wish for and all the good advice. In particular, I wish to thank June Andronick for her heroic efforts of proofreading this thesis, Thomas Sewell for always being willing to assist me in pushing the Isabelle theorem prover to the cutting edge, Michael Norrish for developing and assisting with the C parser I use in this thesis, and of course Harvey Tuch, from whose initial ideas about separation logic and C semantics sprouted my then far-fetched idea of marrying separation logic and virtual memory.

I also thank my parents for their continued support and patience during all the years of my thesis, as well as the years leading up to it.

Finally, I wish to say a thank you to my first Hapkido teacher, Master Scott Wetherell. While he had no direct involvement in this thesis, his strength and confidence contributed significantly to my being the person I am today. I hope that through me, part of his legacy lives on. May he rest in peace.

Contents

List of Figures

Chapter 1

Introduction

When planes fall out of the sky or ships collide, it would be really nice to know it was not a software problem. Mechanical parts wear out and fail, human pilots and captains make mistakes, while weather is unpredictable and can be often be dangerous and erratic. Software, on the other hand, does not wear out. It follows a set of prescribed instructions precisely, never wavering even in the face of overwhelming odds. As for weather, it can certainly interfere with the hardware that software runs on, but we have not yet heard of any weather phenomenon which interferes with software directly.

Software controls an ever-increasing amount of machines and devices around us. As we become more dependent on it, it simultaneously becomes more important that the software correctly implements the tasks it is meant to do. We can perform tests, but it is intractable to cover all possible cases in this manner. If we desire a higher level of assurance, we can create mathematical models of the software and prove things about them. We thus enter the world of *formal verification*. The question then becomes one of depth: the more in-depth the model, the higher our level of assurance as to the correctness of the software implementation conforming to that model, at the price of time required to verify it. A very simple model can be checked on paper. A simple model can be given to another piece of software and checked automatically, as is the case in model checking. More complex models require both human and machine input into the checking process, and a significant investment of time by both. If we go too deep, the high assurance may arrive after the software is declared obsolete.

The goal of our work is to move the world of operating system verification one step in the direction of realism, by providing a memory model which contains both physical and virtual memory in the same framework, with a single view of memory. We believe our fusion of virtual memory and separation logic constitutes such a step.

1.1 Overview

1.1.1 Theorem Proving and Isabelle/HOL

First, we believe it is important to define what we mean by "verified" and "correctness". A system cannot be verified in a vacuum, and there is no such thing as correctness as an absolute concept. When we say that a system is *verified*, we mean

a proof of that the effects of a system correspond a particular specification. This is known as functional correctness. Properties of systems can also be verified, by which we mean a proof that a particular model of the system exhibits a particular property.

In order to raise the confidence in our proofs beyond the level of paper and pencil, humans have enlisted the use of machines, leading to machine-checked proofs. There exist many frameworks, systems and procedures for providing such proofs. Our weapon of choice is Isabelle/HOL [42].

The Isabelle theorem prover is an interactive theorem proving framework built upon a minimal proof kernel. The kernel is based on LCF, or Logic for Computable Functions [25]. These types of kernels follow the approach of a small core capable of deriving and manipulating theorems, with all other functionality of the theorem prover having to go through this core's interface. The interface in turn is enforced by the type system of a strongly-typed language. Isabelle uses Standard ML [40]. Both the size of the core, since it can be rigorously checked, and the strongly-typed implementation result in a highly reliable proof system. If further reassurance is needed, Isabelle can also export "proof terms" which can be checked by a different, even simpler, theorem prover.

HOL, or Higher-Order Logic, extends first-order logic by introducing types and the ability to quantify over functions. Isabelle/HOL is an instantiation of HOL in Isabelle, along with a large body of structures and proofs about them, creating a rich theorem proving environment.

Every theorem proving environment comes with its own notation and idioms. We will cover those of Isabelle/HOL in Chapter 2.

1.1.2 Operating Systems

The operating system of a device is a software component which abstracts away the direct hardware interface. It provides an alternative high-level interface, upon which other software components can be executed. Operating systems on simple devices may only provide this interface and task scheduling. Modern operating systems running on more capable hardware must handle scheduling and execution of multiple user programs at the same time, while preventing their unauthorised access to hardware, protecting the operating system itself from corruption by user programs, and also protect the user programs from each other. This includes managing access to shared resources and inter-process communication. The operating system *kernel* is the part of the system that runs in privileged mode with direct hardware access. By setting up the hardware, the kernel constructs a restricted environment in which client programs can run. We call this restricted environment *user* mode. The boundary between user and kernel mode is enforced by the hardware.

The responsibilities of an operating system kernel and its direct access to hardware make it a critical component of the system. If it is not reliable, all programs running on the system may be unreliable. If it freezes, the entire system will be unresponsive. No matter how efficient it is, an operating system's usefulness is limited if its kernel doesn't reliably perform what it was designed to do.

It therefore makes sense to aim every weapon in our computer science arsenal at increasing the reliability and security of operating system kernels: beyond testing, beyond checking of simplified models and into the territory of proving something

about what an implementation actually does.

1.1.3 Virtual Memory

Virtual memory is a hardware-enforced abstraction providing running processes with their own "view" of memory (RAM). It is used in most modern operating systems capable of supporting multiple executing processes. All access attempts to virtual memory are intercepted by translation hardware which either resolves to a real memory access or to a violation reported to the operating system. The translation hardware uses a translation table called a page table, stored in real memory, meaning that writing to memory can change the layout of memory.

When used correctly, the virtual memory subsystem allows for process isolation, dynamic allocation, sharing data between processes and controlling access rights to memory regions. When used incorrectly, the system allows security leaks and mysterious crashes in programs whose functioning might otherwise be considered correct.

We will cover virtual memory in more depth in Chapter 3.

1.1.4 Separation Logic

Separation logic is an extension of Hoare Logic [28]. It was first presented by Reynolds [48] as a method for reasoning about imperative programs dealing directly with manipulation of memory. It focuses on local reasoning: separating out and defining rigidly the region of memory involved in access operations. This allows inferring the regions of memory which do not change, eliminating artifacts such as pointer aliasing. We introduce it in more detail in Chapter 4.

1.2 Contributions

We claim the following technical contributions:

- A novel approach to modelling behaviour of programs executing in the presence of virtual memory, based on separation logic. We model virtual and physical memory accesses in a single, unified view, without the need to enter special "modes". To our knowledge, we are the first to do this.
- Development of a separation logic which allows convenient, abstract reasoning about both the virtual and physical layers of memory, supporting the frame rule for arbitrary writes to memory (including the page table). The logic includes a separating conjunction extending to virtually shared memory.
- An abstract framework which makes the core logic independent of the machine architecture and page table implementation.
- An instantiation of our framework developed for demonstration purposes, using a simple machine architecture with a one-level page table, combined with a simple, deeply embedded imperative programming language with arbitrary pointer arithmetic. We prove this instantiation constitutes a separation logic according to the requirements of Calcagno et al. [13], and demonstrate that it can be applied in practice in a simple case study on page allocation.

- A formalisation of the page table subsystem on the ARMv6 architecture in non-legacy mode. We omit some features such as domains, but model a functionality subset sufficient to cover the usage patterns of the seL4 microkernel [31]. To our knowledge, this is the first formalisation of an ARM memory management subsystem in a theorem prover or otherwise.

- Instantiating our framework to our model of the ARMv6 architecture and a shallow embedding of the C programming language, made practically usable by connecting it with the verification framework based on Schirmer's SIMPL [49] used in the L4.verified project [31].

- A case study demonstrating modular reasoning about a C function which modifies the page table, identifying its precise semantics as a function inserting a new mapping into the page table. The function chosen is a small one from the seL4 microkernel [31], but the concept demonstrated applies in general.

All results presented in this work are formalised in the Isabelle/HOL theorem prover [42]. All definitions, datatypes and rules we present are output by the Isabelle/HOL theorem prover, resulting in this document being an accurate representation of our work.

1.3 Related Work

We gave a brief overview of the core topics relevant to our work in Section 1.1. Now we will discuss the relevant work of others.

1.3.1 Separation Logic

Since its inception [48], classic separation logic has been mechanised in Isabelle/HOL by Weber [60], HOL4 by Tuerk [58] and Coq by Marti et al. [39]

Further work by Calcagno et al. [13] defined what exactly constitutes a separation logic. By formulating an *abstract* separation logic which, appropriately, abstracts away the specifics of memory model and state, they formulate a class of *separation algebras* suitable for local reasoning. They demonstrate that the original separation logic is one instance of abstract separation logic. They define the properties a logic must have in order to be called a separation logic; properties which we use to demonstrate that our logic is a separation logic in Chapter 5.

Although we do not use them in our work, the local semantics of separation logic allow for reasoning about concurrency [10] [44]. If two regions of memory are separate, it logically follows that updating one does not affect the other. This means if the regions are given out to two processes, those processes can be executed or interleaved in any order. It also means there can be no sharing between them.

In order to enable read-only sharing of resources between concurrent processes while remaining a separation logic, Bornat et al. [10] introduced *permissions*. Instead of treating cells in memory as atomic units, they allow referring to parts of them, e.g. half of the cell at address 5 contains the value 7. The semantics allows memory reads from partial cells, but memory writes still require an entire cell. Our work uses a similar idea in the concept of *slices* introduced in Chapter 5. We compare these ideas in detail in Chapter 7, Section 7.2.

1.3.2 Operating System Verification

The first attempts to realise the vision of a verified operating system were PSOS [18] and UCLA Secure Unix [47]. The former had a specification, but no proofs. The latter had a specification and about 20% of proofs refining it to its implementation in Pascal. The first system to be completely verified with respect to a specification was Bevier's KIT [7]. By aiming for a smaller, simpler kernel, Bevier managed to push implementation proofs right down to assembler. Unfortunately, this approach resulted a set of kernel features too small to be used by the industry. Thus, after this achievement, the next challenge was to create a more industry-capable verified operating system.

While this was going on, the operating system world started experimenting with scaling back the size of the kernel, or at least breaking it up into pieces, in order to more closely approach real modularity by having the boundaries between operating system modules enforced by hardware [26]. The most common term for such a system is *microkernel*. There also exist *hypervisors* [11,24] which, while similar in their minimalist spirit to microkernels, are targeted at running multiple guest operating systems rather than programs. While microkernels can act as hypervisors, the latter are tuned to re-expose an interface resembling actual hardware, while the former offer a more generic interface.

Unfortunately, microkernels were hard to get right. The first microkernel, Hydra [61] was never widely used due to poor performance. The Mach [2] kernel, which coined the phrase microkernel, was somewhat large and also had poor performance. As a consequence, migration of features into the kernel to improve speed resulted in a kernel that was anything but micro. These early results suggested a different approach was necessary: moving into user mode anything that does not need to be in the kernel, focusing on appropriate abstractions and fast communication [38]. Some of the new wave of microkernels that resulted from this approach did successfully make it into industry: QNX [51], μ-velOSity [50] and OKL4 [35].

The increased viability of microkernel-based operating systems whose kernel size was deserving of the name also contributed to an atmosphere in which attempting verification of operating system kernels made sense [56]. The Verisoft project [4] attempted verification of an entire stack: from user-mode applications down to the logic gates of the processor. Verisoft completed most, but not all of the proofs. Nonetheless, their scope was a narrow slice through all layers, so their kernel was simplified and optimised for verification, not performance. The first kernel with a real claim to the industry-capable verified operating system kernel title turned out to be the seL4 kernel from the L4.verified project [31]. We use some of the source code from this kernel in our case study in Chapter 6.

Due to their direct interface to hardware, as well as the goal of utilising hardware resources as efficiently as possible, operating systems are typically implemented using low-level programming languages such as C [30], even using processor instructions directly where necessary. Since separation logic was developed for reasoning about imperative programs with direct access to memory, that makes it applicable to operating system verification.

1.3.3 Low-level Verification

Assembler

Assembler is the human-readable version of instructions executed by a processor. Given a semantics for those instructions, one can show the exact result of them being executed. At the same time, assembly offers little higher-level information for humans, making it a challenge to reason about. Our work does not include reasoning about programs at the assembly level, even though we do model the bits and bytes in a page table and the lookup procedure used by hardware.

C

The C programming language [30] is one of the most popular languages used for developing low-level applications, as well as operating systems. Although C++ is also used, the subset of its semantics that has been formalised remains small [59]. Projects attempting verification of C++ code, such as VFiasco [29], have not yet moved past their preliminary stages. It is in our opinion far too complex a language to involve in formal verification.

In fact, although the C programming language appears simple, it is sufficiently complex for it to be tricky to reason about. Various attempts exist at modifying C to be simpler for verification purposes. Examples include C0 in the Verisoft project [36], Cminor [37] and C-light [8] (a "representative subset").

Norrish formulated a formal semantics of the C programming language [43]. He also developed a C to Isabelle/HOL parser [57] which parses a substantial subset of C code. The result is represented inside Isabelle/HOL using the SIMPL [49] environment developed Schirmer for verification of sequential imperative programs. When combined with the memory model of Tuch [54, 55], which includes storage of C structures and padding, the combined framework [57] captured a significantly large part of C semantics to be used for the verification of the seL4 microkernel [31].

Tuch's memory model supports both separation logic and typed heaps of Burstall [12] and Bornat [9], and takes into account the presence of padding in C *struct*s. It is, however, a flat memory model and does not deal with virtual memory in any way. Our work is inspired by the results of the L4.verified project, but our focus is on virtual memory. We use Norrish's C parser, but plug in an entirely new memory model. We focus on separation logic exclusively. We also don't currently handle padding, since we observed that no C structure in the seL4 kernel has any.

There exist other tools for formal reasoning about C code. Key-C [41] deals only with the type-safe subset of C. VCC and Caduceus both generate proof obligations from annotated C code. VCC [15] tries to discharge them automatically. It additionally supports concurrency. The Verisoft XT project used VCC to successfully perform automated proofs for a very small hypervisor [5]. However, VCC uses a memory model which axiomatises a weaker version of what Tuch proves. Caduceus [20] translates C code into the functional language of the Why [19] tool, which then generates verification conditions that can be proved by other tools. Like Norrish's C parser, Caduceus covers a substantial subset of C. Caduceus is now part of the Frama-C framework [1]. None of memory models in these frameworks have virtual memory built in.

Gast [23] describes a way of reasoning about some of C's more complicated

features, such as pointers to local variables, which don't typically appear in formalised C semantics such as those used in the L4.verified project or Caduceus. To our knowledge, a larger framework incorporating such features does not yet exist.

Architectures and Virtual Memory

Reasoning about the interaction between software and hardware requires a model of the hardware. Our work involves reasoning about virtual memory on the ARMv6 architecture. While thorough models of the ARMv6 [21] and ARMv7 [22] instruction set semantics exist, they do not include the memory management unit. Tews et al. include some virtual memory characteristics in their model of the IA32 architecture [53]. Our work contains a model of the page table behaviour of the ARMv6 in *non-legacy* mode, as described in Chapter 3 and Chapter 6. To our knowledge, it is the first such model.

When it comes to reasoning about programs executing in virtual memory, especially the operating system itself, we have seen two approaches used.

The first approach is to set up a rigid page table structure, abstract it away and from that point on act like it isn't there. This approach was used in the Verisoft project [3]: a simple one-level page table for user programs, direct physical access for the kernel (no translation), limited functionality for mapping, unmapping and remapping. The simplicity, however, allowed for verifying a paging mechanism. The L4.verified [31] project sets up a semi-rigid structure: a range of addresses at the top of virtual memory in any page table is accessible by the kernel only and maps directly, one-to-one, to physical memory. The lower addresses of virtual memory can be used by user-level tasks, but the kernel never touches it; it only manipulates memory through the top section of virtual memory. The L4.verified project proved invariants which state that even though the kernel page table is in the one-to-one virtual memory region, all writes to this region do not affect its mappings in virtual memory. However, since virtual memory is not part of the core memory model, the invariants do not *follow* from the model, but rather are forced onto the model by human hand.

The other approach we have encountered is a "magic mode" abstraction. In normal mode, we expect virtual memory to behave much like normal memory, and the set of mappings from virtual to physical memory cannot be modified. To modify the contents of page tables and, as a result, the mappings, one has to enter a special mode. In this special mode one can modify the memory however one wishes, and any invariants about the memory state have to be reestablished upon returning to normal mode. This approach is used in the Robin project [52, 53]: there is a subset of addresses which is *blessed* and can be accessed normally as *plain* memory. Writing to outside blessed regions violates the plain memory property and requires that the property be reestablished.

In our work, we acknowledge the fact that writing to a page table entry necessarily invalidates the regions of virtual memory that are resolved through that entry, but we also observe that it isn't the end of the world. Writing to the second level of a two-level page table with 4KB sizes only affects 4KB of virtual memory addresses, and the rest remain unaffected. This means we can apply local reasoning, and hence separation logic, to specify exactly what changes. When we know *exactly* what changes, we can derive what does not change via the frame rule. Additionally,

by using a single memory model, we do not enter or exit any special modes. Our rules are standard separation logic Hoare-style triple reasoning.

1.4 Layout of the Rest of the Document

Chapter 2 introduces the basics of Isabelle/HOL notation as well as the more common structures we use in our formalisation and proofs. Although this is mostly standard and can be skipped by a reader experienced in Isabelle/HOL, we do use some variants of standard notation for presentation purposes.

Chapter 3 contains an in-depth discussion of virtual memory, address translation, issues involved and different page table types. A reader experienced with operating systems and virtual memory may wish to skip directly to Section 3.3 where we discuss the ARMv6 page table which we will use in instantiating our framework in Chapter 6.

Chapter 4 contains an introduction to separation logic, beginning with an informal. A reader well-versed in separation logic may wish to skip this chapter entirely.

In Chapter 5 we present the core idea of our work, mapped separation logic. We instantiate it to a simplified machine architecture supporting virtual memory. We also perform a small case study using a simple programming language to show it is indeed a separation logic. This chapter is based on our paper by the same name [33].

Chapter 6 describes our improved and extended version of mapped separation logic. We instantiate it to an ARMv6 architecture and page table and add pointers to objects in memory. We create an interface to the C programming language and the Norrish's C parser [57], along with a simple demonstration of this functionality. Section 6.8 of Chapter 6 contains our case study, in which we verifying the semantics of a function from the seL4 microkernel [31] which writes to the page table.

Chapter 2

Notation

In this chapter we will give an overview of Isabelle/HOL [42] idioms and notations. As we mentioned in Chapter 1, Isabelle/HOL is a Higher-Order Logic implementation on top of the Isabelle theorem prover.

2.1 Isabelle

Isabelle has its own "meta" logic, in which other logics, such as HOL can be implemented. For theorems in the metalogic, there are zero or more assumptions, as well as one goal. We write these using meta-implication, denoted by $\_ \Longrightarrow \_$. It has multiple syntax variants. The following are all equivalent:

$$A \Longrightarrow B \Longrightarrow A \wedge B \qquad \llbracket P;\ Q \rrbracket \Longrightarrow P \wedge Q \qquad \frac{P \qquad Q}{P \wedge Q}$$

The rightmost variant mimics standard logical notation, thus we use it for presentation of rules. There is also meta-equality, written as $\_ \equiv \_$. These mostly appear in definitions. For the purposes of reading this document, it is safe to assume that meta-equality is the same as normal equality.

Isabelle also supports a type system. We write $x :: 't$ to indicate that x has type $'t$. Type variables have an apostrophe in front, e.g. $'t$, $'zft$. By convention, type names are usually lowercase.

Functions in Isabelle are total. Their types are written using $\Rightarrow$, e.g. $'d \Rightarrow 'r$. Application of a function f to arguments x, y and z is written as $f\ x\ y\ z$, which yields three function applications, resulting in functions at each stage, i.e. $(((f\ x)\ y)\ z)$. One can construct functions on the fly using lambda notation, e.g. $\lambda x.\ x = y$.

Isabelle supports custom syntax for constants. When dealing with the syntax of *infix* operators, we refer to the operator itself using op. For example, $a + b$ is identical to $op + a\ b$.

2.2 HOL

We will now cover the standard components of the HOL instantiation in Isabelle. We will refer to this as "HOL" throughout.

Basic Logic

HOL defines the standard boolean operations of first-order logic, as well as standard quantification operators:

negation	`¬ P`
conjunction	`P ∧ Q`
disjunction	`P ∨ Q`
implication	`P ⟶ Q`
universal quantifier	`∀x. P x`
existential quantifier	`∃x. P x`

HOL also provides the choice operator `SOME`, which returns any member of a type meeting a certain predicate, provided such a member exists. For example, `SOME x. x < 17` returns a number less than `17`, but does not specifically state which number it is.

Functions

Functions in HOL are defined using a set of equations. The simplest of these are definitions, which only have one equation and can neither recurse nor have any constructors on the left-hand side. For example, the definition of the identity function is: `id = (λx. x)`

HOL provides several ways of constructing functions from sets of equations with different tradeoffs between ease of specifying the equations compared to reasoning about them later. Since functions in HOL are total, recursive function definitions must be accompanied by a potentially automatic termination proof. For the purposes of this work, we will simply state the function types and their corresponding equations.

The function composition operator is defined as follows:

```
f ∘ g = (λx. f (g x))
```

The term `f(x := y)` represents a function derived from `f`, but updated such that `f x = y`.

Let

The let construct allows writing a term such as:

```
let a = f x;
    b = g a
in h a b
```

instead of:

```
h (f x) (g (f x))
```

We use it to improve readability of complicated statements.

Datatypes

Datatypes are types which can have different kinds of contents, each tagged by a *constructor*. The constructor is also a function that will construct a particular kind of content. A simple datatype is that of `bool`, which has the constructors True and False:

datatype *bool* = True | False

By convention, datatype constructors start with a capital letter.

We can differentiate values of a datatype based on the constructor used to make them with case, for example:

case *is_magic* of True ⇒ *42* | False ⇒ *0*

Pairs

Pairs are denoted by *(a, b)* where *a* and *b* are the two members. The first can be accessed with fst, the second with snd.

Multi-member tuples are actually a left-associative construct of pairs. For example, *(a, b, c, d)* has the type *'a × 'b × 'c × 'd*, which is internally represented by *'a × ('b × ('c × 'd))*.

Option

The *'a option* datatype is a simple container type with two constructors:

datatype *'a option* = ⌊*'a*⌋ | None

As its name suggests, it has the option of containing the value of that type, or None. By using an option type as the range of a function, we can create partial functions. HOL even has special syntax for such functions, thus the following types are identical:

bool => bool option
bool ⇀ bool

Given ⌊*x*⌋, we can get *x* directly using the:

the :: *'a option ⇒ 'a*

Note that the None returns undefined, which is a HOL constant representing "any value of a given type". Another name for this constant is arbitrary.

Applying a function to the contents of an option type is done using Option.map:

Option.map :: *('a ⇒ 'b) ⇒ 'a option ⇀ 'b*

As one would expect, Option.map *f* None is None and Option.map *f* ⌊*x*⌋ is ⌊*f x*⌋.

Type Declarations and Type Introspection

The **typedecl** keyword declares a new type in Isabelle without any extra information. We use this in Chapter 6 to declare tags for our two types of pointers: virtual and physical.

Isabelle allows limited type introspection. We can create functions which work on types: rather than working on values of *'a*, such functions work on *'a itself*. *TYPE* allows capturing types to pass as values to such functions, i.e. the type of *TYPE('a)* is *'a itself*.

Types in the *finite* type class have a finite number of members (cardinality). This number can be queried using *CARD*, e.g. *CARD('a)*.

Sets

In HOL, sets are represented as functions from the type of element to `bool`. The empty set is denoted by ∅, the universal set by UNIV.

HOL set notation is otherwise standard:

set of elements	`{e1, e2, e3}`
membership	`x ∈ S, x ∉ S`
union	`S ∪ T`
intersection	`S ∩ T`
difference	`S - T`
product	`S × T`
subset	`S ⊂ T, S ⊆ T`
set comprehension	`{x | P x}`

Lists

The `list` datatype features the empty list as well as a list constructor for prepending an element to an existing list:

datatype `'a list = [] | 'a · 'a list`

HOL contains the usual operations on lists: hd gets the first element, tl gets the rest, length calculates the list length, set constructs a set from the lists's elements, map applies a function to all list elements, zip creates a list of pairs from two lists, and foldl reduces a list to one element by applying a folding function from left to right. We list their type signatures here for reference:

```
hd :: 'a list ⇒ 'a
tl :: 'a list ⇒ 'a list
length :: 'a list ⇒ nat
set :: 'a list ⇒ 'a ⇒ bool
map :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list
zip :: 'a list ⇒ 'b list ⇒ ('a × 'b) list
foldl :: ('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b list ⇒ 'a
```

The nth element of a list is obtained using $\_[n]$. Taking and dropping n elements from a list is done using take `n` and drop `n` respectively, while concat concatenates a list of lists into one list.

Numbers

Apart from words, which we will cover shortly, the other type of number which appears in our work is the type of natural numbers. Natural numbers are also a datatype, built around zero and the successor function Suc:

datatype `nat = 0 |` Suc `nat`

The min and max functions respectively find the minimum and maximum of two comparable quantities.

Division is performed by `_ div _`, and when `x` divides `y`, we write `x dvd y`.

N-bit Words

HOL provides a library of n-bit words. These can be considered to be an n-length vector of bits, or a ring modulo 2^n. They can be constructed from a natural number using of_nat, while unat performs an unsigned conversion to a natural number.

The word library provides standard bitwise word operations:

bitwise not	`NOT _`
bitwise and	`_ AND _`
bitwise or	`_ OR _`
shift left	`_ << _`
shift right	`_ >> _`
nth bit as a `bool`	`_ !! n`

The mask function creates a word whose lowest n bits are set. The size function takes an n bit value and returns n as a natural number, e.g.

```
size (w :: 32 word) = 32
```

Maps

In HOL, a map is a partial function, i.e. function with the type:

$$'a \rightharpoonup 'b$$

These are the operations we use on maps:

empty map	`empty = (λx. None)`
domain	dom `m = {a . m a ≠ None}`
merge (right-override)	`f ++ g = (λx.` case `g x` of None ⇒ `f x . ⌊y⌋ ⇒ ⌊y⌋)`
restrict domain	$m\restriction_A$ `= (λx.` if `x ∈ A` then `m x` else None`)`
construct from maplets	`[d1 ↦ r1, d2 ↦ r2]`

2.3 Special Notation in This Work

In this work, we differentiate all constants, such as datatype constructors by using a different font. For example, when stating that `x` is True, one can clearly see which is a constant already defined in the current theory, and which is the variable: `x =` True. Some existing constants define their own syntax, and their writers chose their own notation. In such cases we do not interfere, even if the notation chosen is a normal word. An example of such notation is the integral division operator: `_ div _`.

Lifted conjunction We define a conjunction predicate which works on predicates taking one parameter and returning `bool`:

```
_ ⌊∧⌋ _ :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool
P ⌊∧⌋ Q ≡ λx. P x ∧ Q x
```

K combinator We state the functional K combinator in HOL:

```
K :: 'a ⇒ 'b ⇒ 'a
K ≡ λx y. x
```

Map disjunction We define two maps to be disjoint if their domains are disjoint:

$$\_ \perp \_ :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$$
$$h_0 \perp h_1 \equiv \mathsf{dom}\ h_0 \cap \mathsf{dom}\ h_1 = \emptyset$$

We will explain the constants and their notation as we define them. We may repeat some of the definitions shown in this chapter for reader convenience.

Chapter 3

Virtual Memory

Virtual memory is a hardware-supported abstraction over physical memory. It is used in most modern operating systems. Virtual memory allows isolation of programs, dynamic allocation, permission control over memory access and sharing of memory between programs. Although the term "physical memory" is sometimes extended to encompass all physical media such as disk, we consider only the main system memory to be physical memory. This is nearly always RAM.

In this chapter, we introduce virtual memory, its relevance, fundamental concepts and mechanisms. In Section 3.2 we introduce page tables, and in particular, in Section 3.2.2 we discuss the most common setup for 32-bit machines: the two-level page table with optional superpages. Section 3.3 documents an implementation of this on a real architecture, ARMv6. We conclude with a discussion of some of the less popular page table models, which we do not formalise in our work.

We will present a formalisation of a one-level page table in Chapter 5, and a two-level page table of the ARMv6 architecture in Chapter 6. While our formalisation does not presently cover other page table types, caches or the translation lookaside buffer (TLB), we nonetheless discuss these concepts in this chapter, in order to give an impression of the diversity of the underlying mechanisms our framework may later be applied or extended to.

3.1 Importance of Virtual Memory

An operating system executing highly specialised tasks on limited hardware may depend on a priori knowledge about the nature and requirements of those tasks. However, an operating system claiming to be *general purpose* cannot, by its very nature, know what tasks executing on it will try to do. In particular, it cannot know how much memory a task will require.

A task is supplied by a user in the form of a program. To run the program, an operating system creates a process. Since programs are user-supplied, their memory requirements are initially unknown. Allocating "enough" memory beforehand makes no sense, as it may be too little (task cannot complete) or too much (memory wasted on this process cannot be given to other processes). Trusting the user to supply this information likewise makes no sense, as the user may be wrong, their program may be buggy or even malicious.

The virtual memory mechanism involves hardware-level monitoring of every memory access attempt performed by a process and checking it against a table of

valid memory locations for the process. This allows allocation of physical memory on an as-needed basis, while restricting access to memory the process is not authorised to access. The operating system sets up these tables and is invoked whenever there is a problem.

An operating system running user-supplied programs should minimise the possibility of those programs being able to interfere with each other, whether accidentally or intentionally. This means that programs must not be able to access each other's memory unless *specifically permitted to do so* by first asking the operating system for permission. Permission granted need not be absolute; access to memory may place restrictions on execution, reading or writing. The virtual memory mechanism is enforced by hardware, and the operating system controls the mechanism, which allows memory isolation between programs.

Work performed by a process at any time tends to centre around a set of resources relevant to the task at hand. This set of resources that the process is currently working with and accessing in quick succession is called the process' *working set*. It changes over time. The implication of this is that to perform its task successfully, a process does not need all of the memory allocated to it all of the time. Physical memory not in use by a process can be backed up onto another physical medium, such as disk, and made available to whichever process needs it next. Since the virtual memory mechanism monitors all memory access, the operating system can request that it be invoked when the process tries to access its "missing" memory, in order to load it from disk. This process is known as *demand paging* or *swapping*. Since the focus of swapping is keeping the working set of processes in memory, it allows execution of processes with resident sizes exceeding the size of physical memory. There is a limit to the benefits of swapping: when the sum of working sets of all processes on the system exceeds the size of physical memory, however, the system will spend most of its time juggling memory between processes and disk, resulting in performance degradation.

We omit any discussion of segmented memory architectures. While the rather popular x86 architecture is segmented, the segments are typically set up to be identical. Actual use of segmentation is rare; e.g. using a segment for thread-local storage on Windows and some versions of Linux.

In the remainder of the chapter, we will describe the mechanism by which virtual memory can allow the behaviours mentioned in this section. We will omit discussion of swapping, as it is a matter of operating system policy rather than the mechanism itself.

3.2 Virtual Memory Overview

The basic idea behind virtual memory is that a process cannot access physical memory directly. Instead, a process accesses *virtual memory addresses*, which are resolved to physical addresses by a hardware-supported translation mechanism controlled by the operating system.

From the point of view of the user process, it has access to a large, contiguous address space. Reading and writing to "allocated memory" works. Execution proceeds normally. It is as if the process was the only one on the system. In reality, this address space *does not exist*. This is where virtual memory gets its name. Only

physical memory exists. Virtual memory accesses are treated as requests. They can be:

- denied - e.g. invalid address, attempt to write to a read-only area; usually results in the process being killed.
- delayed - the value viewable at a virtual address by a given process is not currently available; for instance, it has been placed on disk.
- granted - the virtual address is translated to a physical one and the access is performed on the physical address.

In the first two cases, a hardware-enforced virtual memory abstraction invokes the operating system with information on what went wrong.

3.2.1 Page Tables

So far, we have been talking about virtual memory in terms of individually mapping each virtual address to a physical one. The virtual-to-physical memory map is many-to-one (the same physical address may appear at multiple virtual addresses), partial (some virtual addresses are not mapped), and annotated with permission flags supported by the hardware (e.g. read/write/execute). In order to perform the per-process access control and translation necessary for virtual memory to work, these mappings have to be stored somewhere. Typically, they are stored in physical memory where the hardware can access them to perform lookups. The encoded form of the set of virtual-to-physical mappings, stored in physical memory, is known as a *page table*.

Access to and manipulation of page tables is the domain of the operating system kernel. The kernel needs to set up its address space in order to access the entirety of physical memory and also set up the address spaces of user-level programs. Conversely, a user-level program should never have access to its own page table. If it can access its page table for any reason, it can modify its memory layout to allow access to data in other programs and in the operating system itself. This circumvents the point of virtual memory in the first place.

Let us examine the case of a 32-bit machine, i.e. a machine with 32-bit registers. Since memory pointers need to fit in registers, the machine will have a 32-bit virtual address space. Typically, the physical address space will also be 32-bit, though hardware specifics will dictate how much of it is accessible. This means there are 2^{32} virtual addresses to keep track of per process. Each mapping is 32-bits in size, i.e. 4 bytes. Storing one physical address per virtual address in the mapping would hence require $4 * 2^{32}$ bytes (16GB) of memory for the entire set of mappings, *per process*. Clearly, this is not a viable solution.

One way to approach this problem is by decreasing the *granularity* of the mappings, from bytes to larger-sized *pages*. In each entry, we store the physical address a page is mapped to. A typical page size is 4KB (2^{12} bytes). If we store one physical address per page, we now need 2^{20} mappings per process for a 32-bit machine.

In this setup, the virtual address can be broken down as in Figure 3.1. The highest 20 bits of the virtual address indicate which page it lies in – the *page number*. This can be used as an index into the table of mappings, to obtain the physical address of where the page is mapped to. The lowest 12 bits of the virtual address

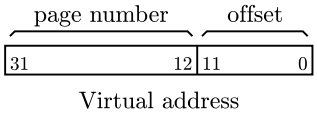


Figure 3.1: *A 32-bit virtual address on an architecture with 4KB page sizes.*

are the *offset* within the page. By adding them to the physical address the page is at, we resolve the virtual address to a physical one.

With 32-bit physical addresses, this means our table needs to be $4 * 2^{20}$ bytes per process, or 4MB, already a significant improvement. The downside to a decrease in granularity is imperfect use of allocated memory. If a process uses only one byte in a page, the operating system needs to allocate it the entire page anyway. This is called *internal fragmentation*. Per-byte granularity has no internal fragmentation, but is effectively impossible. Some hardware supports variable-sized pages, shifting the responsibility for the most efficient choice to the operating system designer, which we discuss starting with Section 3.2.2.

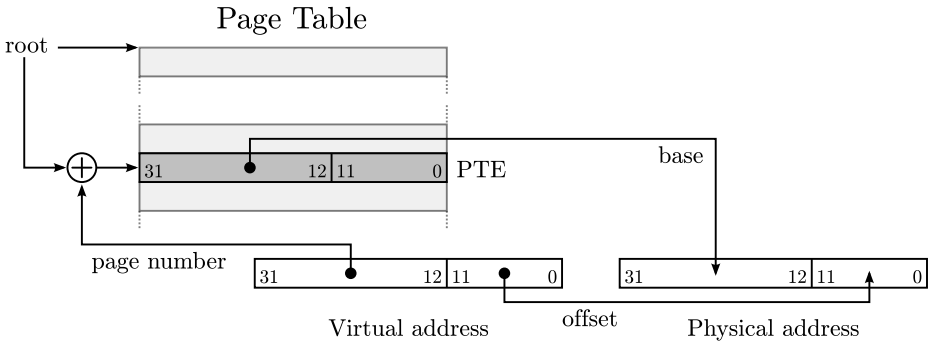


Figure 3.2: *A lookup in a one-level page table on a 32-bit machine with a page size of 4KB.*

One more aspect of our mapping table to consider is the physical equivalent of pages: *frames*. In the previous paragraph, we allowed a page to map to any physical address. Real hardware does not allow this due to efficiency considerations. Instead of performing addition, the hardware takes a 32-bit virtual address, translates the first 20 bits as in our example, and leaves the last 12 unchanged. The first 20 bits before the translation we call the *page number*, while after the translation they become the *frame number*. Taking this into consideration, when we store the frame's address in our table, we know the last 12 bits will always be zero, since the last 12 bits of any page are zero. Therefore rather than wasting 12 bits per entry, one can place any extra flags (e.g. read/write/execute permission flags) there. A special value, typically 0, is used for entries to indicate that a page is not mapped, i.e. does not have a corresponding physical frame.

There exist many ways of encoding the virtual-to-physical mappings. The table of entries we described in this section is among the simplest available: the *one-level page table*. We will discuss other encodings later in the chapter. The procedure we described for using the page table to resolve a virtual address into a physical one we refer to as a *page table lookup*. Figure 3.2 shows this process for our one-level page table. We refer to the location at which the page table can be found as the page table *root*.

In Chapter 5, we will discuss our formalisation of a simple 32-bit machine

with a one-level page table, as a simple test case for our mapped separation logic framework.

3.2.2 Multi-level Page Tables

In Section 3.2.1 we explained the origins of the single-level page table. Though a single-level page table can store the mappings for the entire virtual address space, it cannot store any fewer. It turns out that in practice, a process usually does not use the entire address space. At 4MB, the single-level page table we described would use up 200MB of memory for 50 processes, regardless of how sparse their use of the virtual address space actually is. This is a waste of memory. What we need is a structure that can map sparse address spaces efficiently. The per-process solution to this problem is use of a *multi-level* page table.

In our single-level page table example, we divided a virtual address into a 20-bit page number and a 12-bit offset within that page. We used the page number as an offset into a table of addresses of corresponding frames.

The concept of the *multi-level* page table is to simply divide the page number further, using each part as an index into a table of indices to subsequent tables.

Two-level Page Table

To convert our one-level page table example from Section 3.2.1 into a two-level page table, we can divide the 20-bit page number into two 10-bit indices. Figure 3.3 shows the resulting lookup pattern: we use the first 10 bits as an index into the first-level table, obtaining an entry which may contain the address of the second-level table (or an indication that the corresponding memory area contains no mappings – typically an address of 0). Once we have the address of the second-level table, we use the second 10-bits of the page number as an index into that table to obtain the same kind of page table entry as we had in the single-level page table (frame number + flags).

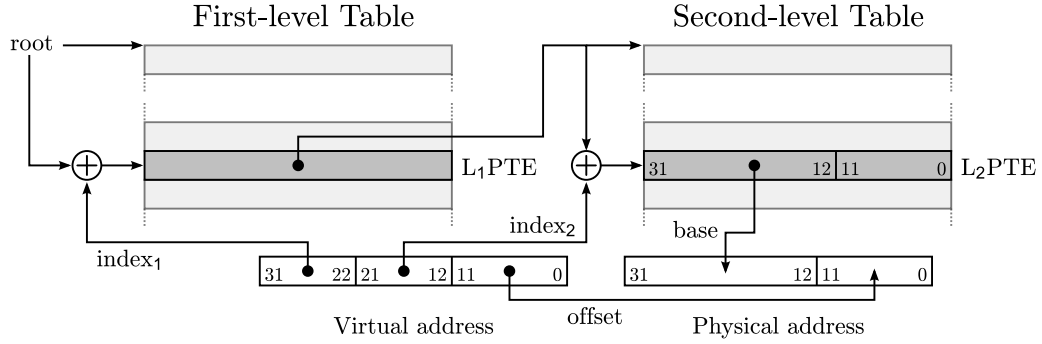


Figure 3.3: *A lookup in a two-level page table on a 32-bit machine with a 4KB page size with a 10-10 split on the page number.*

The minimal memory use of such a page table is the size of first-level page table – 2^{10} 32-bit pointers to second-level tables, 4KB. Should all second-level tables be created, their combined size is identical to the size of a one-level page table. Thus, when fully populated, the overhead of a two-level page table over a one-level page table is the size of the first-level table, i.e. 4KB. As virtual memory usage is typically sparse, the two-level table yields a significant improvement in efficiency.

Many-level Page Table

For 32-bit systems a two-level page table is typical. Due to the efficiency issues we outlined, 64-bit systems require use of more levels.

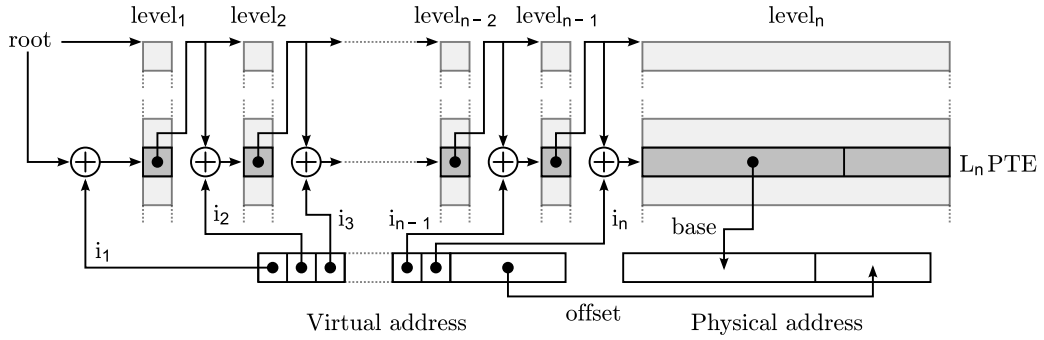


Figure 3.4: *A lookup in a multi-level page table. Intermediate levels not to scale.*

By "many-level" we refer to page tables with more levels than the two-level ARMv6 page table which we formalise in Chapter 6. For 32-bit systems, two levels are sufficient. For 64-bit systems, four or even five-level tables may be used. Figure 3.4 shows how the extra levels are just a variation on a normal two-level page table, but with the page-number bits split into multiple page-table indices rather than just two.

Superpages

The two-level page table in Section 3.2.2 maps one 4KB page at a time. A first-level entry can point to a second-level table, which contains 1024 entries, each corresponding to a 4KB page. Thus, in total, one first-level entry is responsible for a 4MB block of virtual memory. By extracting a physical address and a "don't look further" tag at the first-level entry, we can map a 4MB block of virtual memory to a 4MB block of physical memory without the use of a second-level table. In general, we refer to such areas of memory, mapped using a larger-than-page granularity as *superpages*.

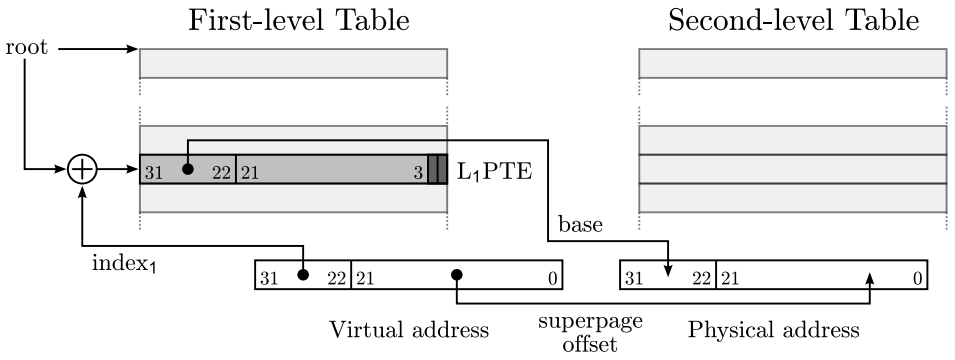


Figure 3.5: *A lookup in a two-level page table completing early due to a "don't look further" flag in the first-level table, resolving to a superpage.*

A "don't look further" bit can be squeezed into the first-level entry of our example if we know that second-level table entries are, for example, 4-byte aligned –

their address is divisible by 4, so its last two bits are zero, allowing two bits of extra information. This alignment can be enforced either by the hardware or the operating system. Figure 3.5 shows the revised lookup pattern.

The same modifications that allow the existence of superpages in a two-level page table allow superpages to exist in a multi-level page table, though naturally there will be more possible superpage sizes.

3.2.3 Translation Lookaside Buffer

Page table lookups are expensive; they potentially involve multiple memory reads: one per page table level. To decrease this cost, these lookups are cached in most architectures in a *translation lookaside buffer* (TLB), a hardware component within the processor.

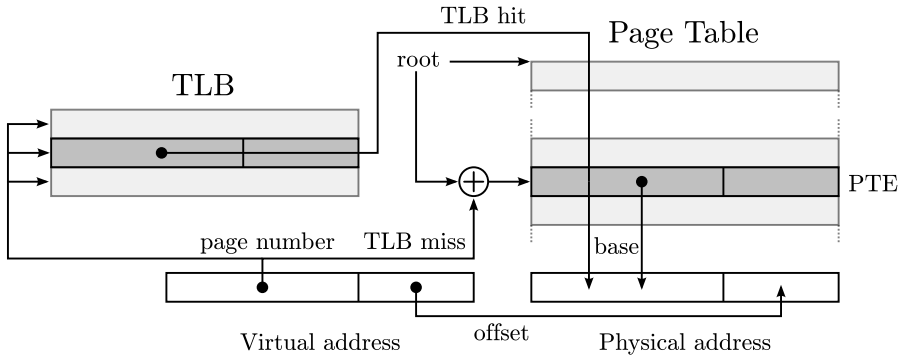


Figure 3.6: *Involvement of the Translation Lookaside Buffer (TLB) in the virtual memory resolution process. If the page number is in the TLB, the base address in the matching TLB entry is used. Otherwise, the page table is consulted.*

Abstractly, the TLB can be seen as a finite, small set of virtual-to-physical mappings. They may include lookups for code instructions as well as data. It is architecture-dependent whether these are handled separately from each other or not, how large the TLBs are, and when a mapping is removed from the TLB and replaced by another. Most architectures provide assembler instructions for explicitly removing all or specific mappings from the TLB, which is called *flushing*.

Although the page table should ultimately define what a mapping is, the hardware will always first consult the TLB and ignore the contents of the page table if a TLB entry is found. When we change the page table and the TLB contains the mapping being changed, we may introduce an inconsistency. This inconsistency can be resolved by flushing the TLB such that the new page table contents will be loaded for future lookups. However, indiscriminate TLB flushes are expensive, as they may flush TLB entries that could be used in the future, requiring extra lookups and hence additional memory reads. Kernel programmers like to optimise by deferring TLB flushes as far as possible and by making them as specific as possible.

3.2.4 Caching

Memory caching is a mechanism which allows processor-local storage of values obtained from memory. It also allows local storage of memory writes, deferring

their transfer to memory. It adds another indirection layer between a user process and memory.

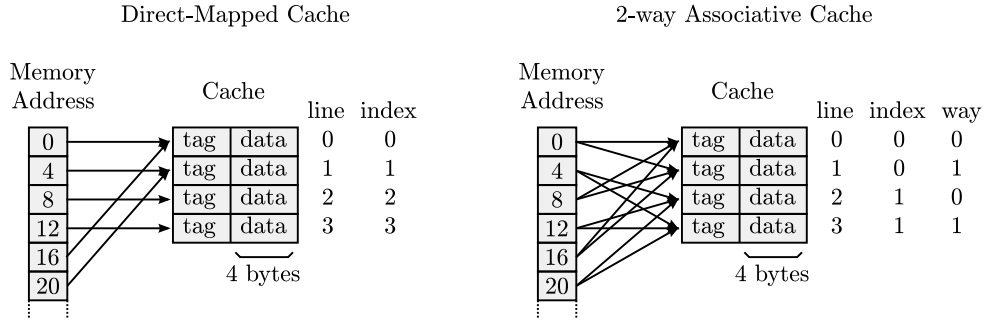


Figure 3.7: *Mapping memory addresses to cache lines for a 4-line cache with 4-byte lines, in the case of 1-way associativity (direct-mapped) and 2-way associativity.*

A cache is arranged into blocks of data called *lines*. Each line is tagged in order to identify which line-sized block of memory it corresponds to.

The simplest way of mapping memory addresses to cache lines is shown in the left side of Figure 3.7. Consider a cache with 4 lines, each line being 4 bytes in size. Since one cache line stores 4 bytes, if we map the start of memory to the first cache line, it will store data for addresses 0, 1, 2 and 3. If we then map the next four addresses to the second cache line, it will store data for addresses 4, 5, 6 and 7. When we get to address 16, we run out of cache lines, so we go back to the first cache line, and so forth. The cache index, hence cache line, for an address a, for a cache with L lines of size s is then $i = \frac{a}{s} \bmod L$. In order to use the line, we also need to check if its tag is $\frac{a}{s}$.

Caches are typically smaller than memory, therefore collisions are a certainty. In our example, an address may be associated with only one cache line. For instance, only one address of 0, 16, 32, ... may be associated with cache line 0. If address 16 is associated with cache line 0 and we wish to access address 32, then cache line 0 must be written to memory first, before being loaded with data from address 32. If a program must repeatedly access addresses 16 and 32, the constant accessing of memory will cause performance degradation, while other cache lines might be unused.

In order to address this issue, we increase the *associativity* of the cache. That is, we group cache lines together, and we associate memory locations with a group. For example, the right side of Figure 3.7 shows a 2-way associative cache. The first group of cache lines, denoted by index 0, can store data blocks from addresses 0, 8, 16, 32, ... while lines with index 1 can store data from addresses 4, 12, 20, 28, ... A program repeatedly accessing addresses 16 and 32 can now run using just the cache.

In a 2-way associative cache, we first find the cache line group with the correct index, then examine the tags on both lines to see if any of them match.

In general, for a n-way associative cache with L lines of size s, the index i for an address a is $i = \frac{a}{n \times s} \bmod \frac{L}{n}$. In order to use the line, we also need to check cache lines $2i + k$ (where $0 \leq k < s$) looking for a tag of $\frac{a}{n \times s}$.

Like the TLB, the contents of caches can be selectively flushed, which discards

the contents. Caches can also be selectively *cleaned*, which writes out the cache contents to memory, ensuring consistency.

Caches may be virtually indexed or physically indexed, depending on whether the cache line group index is calculated from a virtual address or the physical address it represents. Similarly, caches can be virtually or physically tagged, depending on whether the tag is derived from a virtual address or the physical address it represents.

The addition of caching adds complexity. Firstly, it can cause inconsistency with memory-mapped devices, as we'll explain in Section 3.2.5. Secondly, the combination of caching and virtual memory can lead to complex interactions on some architectures, which need to be managed to prevent inconsistency.

An example of the latter is the virtually indexed, physically tagged cache on the ARMv6 architecture. Cache lines have a size of 32 bytes, and the size of the tag is designed to be unique for a 4-way 16KB cache. This means there are $16K/32 = 512$ cache lines in total, 128 cache lines per way. The hardware therefore provides a tag size of 7 bits. ARMv6 allows larger cache sizes than 16KB, however. For a 64KB cache, we need a 9 bit tag to uniquely match an address with a line. Since we only get a 7 bit physical tag, but derive the index from a virtual address, two virtual addresses mapping to the same physical address can be stored in two different cache lines, resulting in an inconsistency.

In our framework, we do not currently model or reason about caches. We consider it a promising direction for future work.

3.2.5 Devices

On most architectures, the physical address space does not only access physical memory. Interfaces to devices appear in it as ordinary physical addresses, but in reality they represent external communication. Unlike ordinary memory, the values at these addresses may change without being written to.

The concept of devices is orthogonal to that of virtual memory. The virtual memory abstraction translates virtual addresses to physical ones, after which the memory controller within the processor decides whether the access represents a memory access request or a device access request.

Caching poses a more significant issue. Since memory locations corresponding to device interfaces may change without the software writing to it, hardware caching is typically turned off for these locations. Some mechanisms such as direct memory access (DMA) allow devices to write to memory directly, bypassing the processor and its cache. In such cases, the cache contents must be carefully managed to prevent inconsistency.

Presence of such devices on the system can be modelled explicitly, for instance by specifically defining how they modify memory, or implicitly, e.g. via non-determinism at the point of a memory read. Our framework does not specify any implicit behaviour related to devices. In order to use devices, their actions must be specified explicitly.

3.3 Overview of ARMv6 Address Translation

In order to demonstrate the applicability of our work to a real-world scenario, we chose to instantiate it to the ARMv6 architecture. The ARMv6 architecture uses a two-level page table with superpages and hardware page table lookup. This is a common setup for 32-bit systems, making it a realistic target. Chapter 6 documents our formalisation and instantiation efforts.

On ARMv6, the page table lookup is performed automatically by the hardware, invoking the operating system only when the lookup fails. Thus, the layout of the page table and format of page table entries is dictated by the hardware.

ARMv6 supports certain legacy features, such as subpages, for compatibility with older processors. These are marked *obsolete* in ARMv6 [6]. We do not consider any of the legacy features in our work.

The architecture is 32-bit, with 32-bit virtual and physical addresses. The page table lookup for a virtual address is performed by using the top 12 bits of the virtual address (bits 20-31) as an index into the first-level table and, if necessary, using the next 8 bits (bits 12-19) as an index into the second-level table, as shown in Figure 3.8.

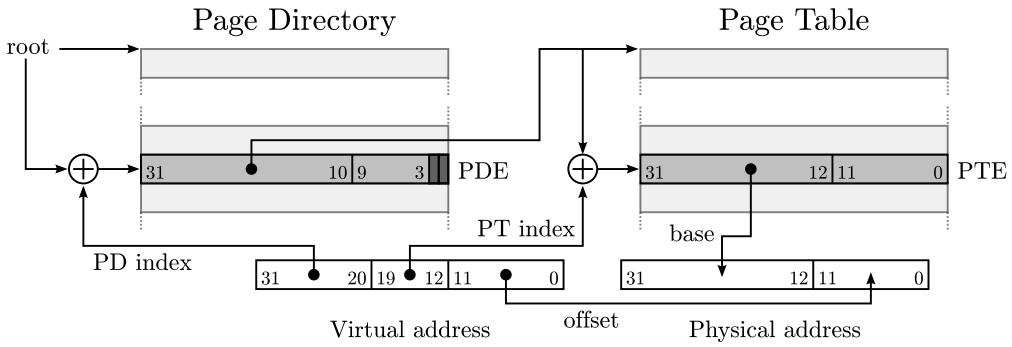


Figure 3.8: *Page table lookup for a 4KB small page on ARMv6*

ARM naming convention refers to the first level table as a *page directory*, whose entries are page directory entries (PDEs), and to the second-level table as a *page table*, whose entries are page table entries (PTEs). We avoid the use of *page table* when referring to the second-level table unless referring to ARMv6 mechanisms specifically, in order to prevent the "do they mean the page table or the *page table*?" ambiguity.

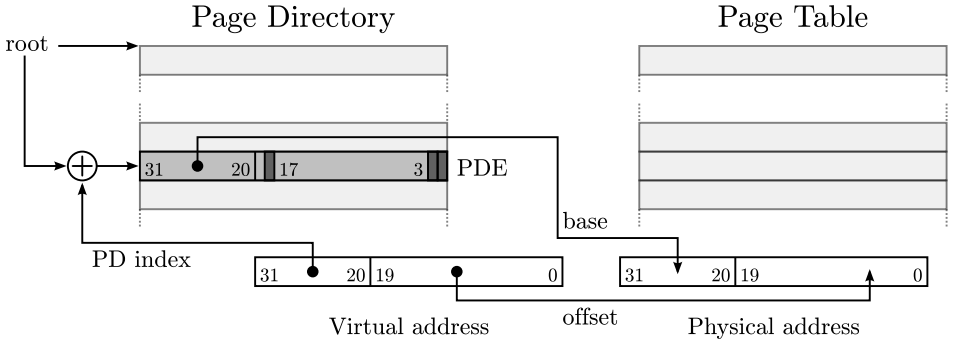


Figure 3.9: *Page table lookup for a 1MB section on ARMv6*

While ARMv6 supports the usual 4KB page size, as well as the 1MB superpages one would expect from the 8-bit index into the second-level table, it also allows

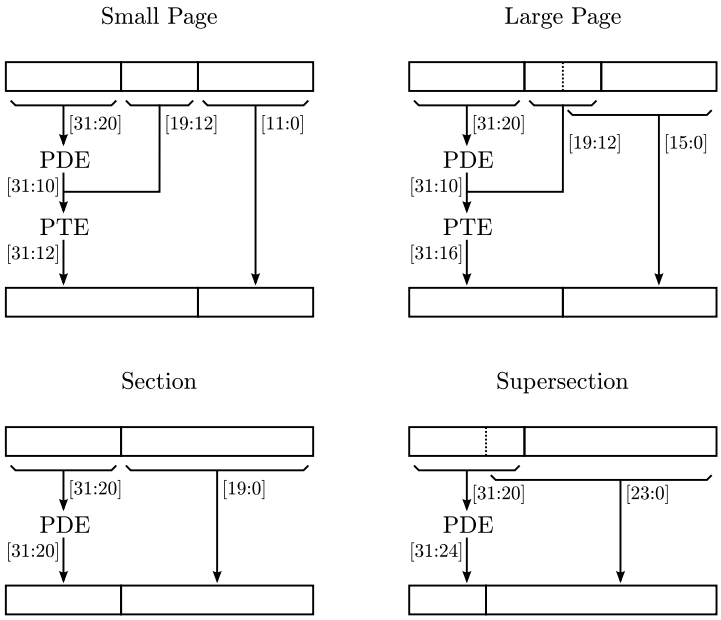


Figure 3.10: *The usage of bits in looking up a virtual address and resolving it to a physical one in ARMv6.*

larger pages (64KB) and larger superpages (16MB). Presumably in avoidance of naming something a "super-superpage", the ARMv6 adopts the following naming convention:

- 4KB page – *small page*
- 64KB page – *large page*
- 1MB superpage – *section*
- 16MB superpage – *supersection*

We will now describe the virtual address translation mechanism on the ARMv6 architecture. We refer to bits from n to m (inclusive) of a value v as v[m:n], e.g. v[31:20] denotes the top 12 bits of the 32-bit value v.

The translation for a virtual address *vaddr* proceeds as follows (see Figure 3.10):

- Use vaddr[31:20] as index into page directory, obtain PDE.
  - If PDE[1:0] = 00 (invalid) or PDE[1:0] = 11 (reserved), the lookup fails.
  - If PDE[1:0] = 10 and PDE[18] = 1, resolve to a supersection.
    Combine PDE[31:24] with vaddr[23:0] into a new address and return it.
  - If PDE[1:0] = 10 and PDE[18] = 0, resolve to a section.
    Combine PDE[31:20] with vaddr[19:0] into a new address and return it.
  - If PDE[1:0] = 01, obtain the address of the second-level table by combining PDE[31:10] with bits [9:0] set to zero.
- Use vaddr[19:12] as index into second-level table, obtain PTE.
  - If PTE[1:0] = 00 (invalid), the lookup fails.

- If PTE[1] = 1, resolve to a small page.
  Combine PTE[31:12] with vaddr[11:0] into new address and return it.
- If PTE[1:0] = 01, resolve to a large page.
  Combine PTE[31:16] with vaddr[15:0] into new address and return it.

The translation for small pages (Figure 3.8) and sections (Figure 3.9) is nearly identical to that of large pages and supersection, except four more bits of vaddr are used for the offset. Figure 3.10 shows the utilisation of bits in each of the four cases.

There is some magic going on here. If vaddr[31:20] are used as an index into the page directory, then how can we use vaddr[23:0] as the offset for superpages? There are 4 bits of overlap. Similarly for large pages, vaddr[19:12] is used as an index into the second-level table, but vaddr[15:0] is used as an offset into the page. The consequence of this fact is that virtual addresses belonging to the same large page or supersection do not necessarily resolve to the same entry in the appropriate table. Instead, they can resolve to one of 2^4 entries. This means that in order to enforce consistency, those entries must be repeated in the table *sixteen* times. Indeed, this is what the ARMv6 standard requires [6].

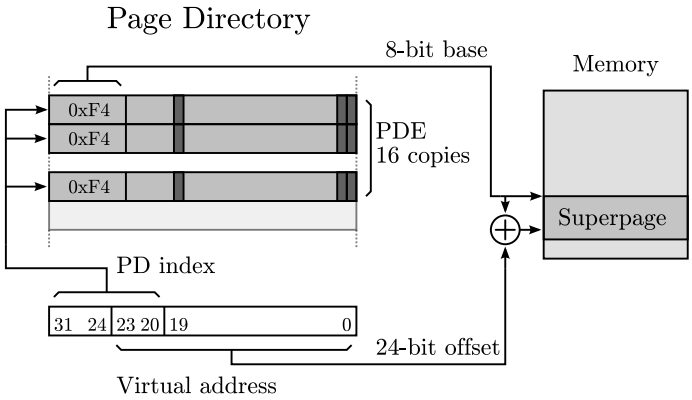


Figure 3.11: *A virtual address lookup in the page directory resolving to a superpage. Note that bits [31:20] are used as an index into the PDE, but only bits [31:24] of the PDE are used as the base address of the superpage. This requires 16 identical PDEs in the page directory. The offset within the superpage comes from virtual address bits [23:0]. A base of 0xF4 indicates the 16MB superpage at physical address 0xF4000000.*

3.4 Overview of Other Page Table Mechanisms

Beyond the multi-level hierarchical page table, there exist other page table schemes. We discuss them here for completeness, along with our thoughts about the difficulty of their formalisation.

3.4.1 Software-loaded TLB

Some processors, rather than implementing specialised page table lookup hardware, leave the implementation of page table lookups to software. If no TLB entry exists to permit/resolve the requested access via a virtual memory address, the processor issues a special interrupt, which invokes an interrupt handler in the operating

system. From that point on, it is the operating system that is responsible for figuring out what to do. The operating system thus can implement an arbitrary page table structure or lookup mechanism. The MIPS processor series is an example of this type of processor. The IA-64 ("Itanium") processor can switch into the software-loaded TLB mode as well.

3.4.2 Virtualised Page Table

A virtualised page table is a variation on another page table type, most commonly a multi-level page table, wherein only part of the page table has to reside in memory whenever the process it belongs to is executing. This core part contains references to virtual, rather than physical, addresses where the remainder of the page table is stored. This means that the operating system can store rarely used parts of the page table on another medium, such as disk, and re-use the memory for another purpose. When a lookup occurs, accessing the missing page table information causes a page fault, which invokes the operating system's page fault handler to resolve the fault and replace the original page. The page fault handler is the operating system routine called upon a page fault.

From a formal perspective, reasoning about virtualised page tables is not very interesting without also formalising the page fault handler itself, and possibly the storage medium. We believe our framework to be sufficient for modelling the part of the virtualised page table that is currently resident in memory, as it is a multi-level page table lookup that either starts or continues in virtual memory.

3.4.3 Guarded Page Table

The guarded page table is a multi-level page table with *path compression* applied. Path compression allows the lookup procedure to skip levels under certain conditions, based on a *guard*. The guard is stored in intermediate page table levels along with a pointer to the next page table level or the final frame address.

Guarded page tables are useful for large, sparse address spaces. For instance, in a five-level page table, to map a single page, one needs to have all five page table levels. This is unnecessary complexity, as we could really encode the frame address directly in the first level of the page table, along with an indication of when that lookup is valid. This indication is what the guard is. Figure 3.12 gives an overview of this process.

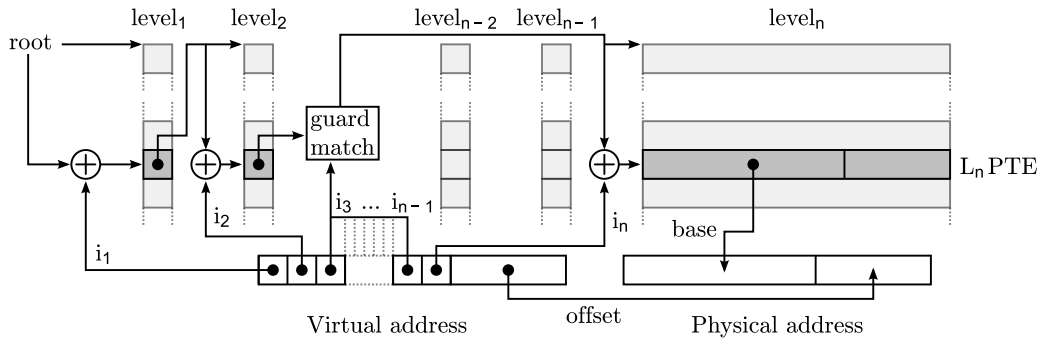


Figure 3.12: *A lookup in a guarded multi-level page table with a guard match at level 2 resulting in a skip to level n.*

To illustrate; consider a simpler example of the two-level page table from Section 3.2.2: 1024 entries per level, each resolving 10 bits, working on 32-bit virtual addresses consisting of a 20-bit page number and 12-bit offset. Let us allocate a page at virtual address 0x123456000 in a previously empty address space. The page number, then, is 0x123456. To perform a lookup, we split it into two 10-bit indices: 0x8D and 0x56. We use 0x8D as an index into the first-level table and obtain our entry, which indicates there are no mappings beyond this point. At this point we have resolved 10 bits of the page number, with 10 bits left to resolve.

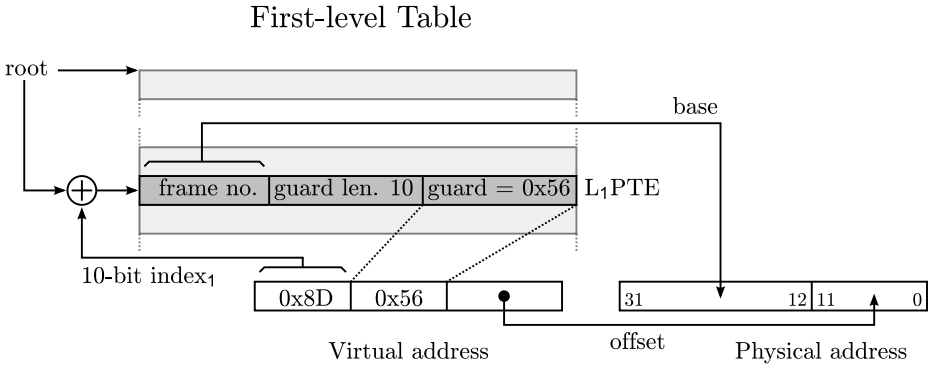


Figure 3.13: *Looking up 0x123456000 in a guarded page table via a PTE requesting 10 guard bits equal to 0x56.*

Normally, we would allocate a second table, point the entry at it, then use 0x56 as an index into the second table and store the physical address of a frame there. Since there would be only one member in the second-level table, this is wasteful. Instead, we can observe that we have 10 bits left to resolve, and in order to be a valid lookup of our page, those bits must be 0x56. We can thus store a new frame address in the first-level entry along with a guard which says "0x56, length = 10 bits". When a lookup for our page is attempted, it will use the first 10 bits to find the first-level entry, then it will check the next 10 bits against the guard. The guard says to check the next 10 unresolved bits against 0x8D. If the guard matches, we get the frame address and skip the second level entirely, as shown in Figure 3.13. If the guard did not match, the lookup would have failed anyway. In our simple example, the guard will always be 10 bits, but in multi-level tables, the length of the guard can vary, and hence there is variation in the number of levels skipped.

From a formalisation perspective, guarded page tables are very similar to superpages in their indication of "don't go any further", except that they indicate a level skip rather than a larger page size.

3.4.4 Inverted Page Table

Unlike the other page table variations, the inverted page table (IPT), is not a variation on the multi-level page table concept. Instead, it uses a hash table at its core, with the page number as the key, possibly along with a process identifier. For the value, it uses the frame number along with any extra flags, such as permissions. Page table lookup proceeds as shown in Figure 3.14.

IPTs are a candidate on systems where the virtual address space is far bigger than the size of physical memory installable in the machine. This is a typical situation on 64-bit systems. The IPT stems from the observation that there is a fixed number

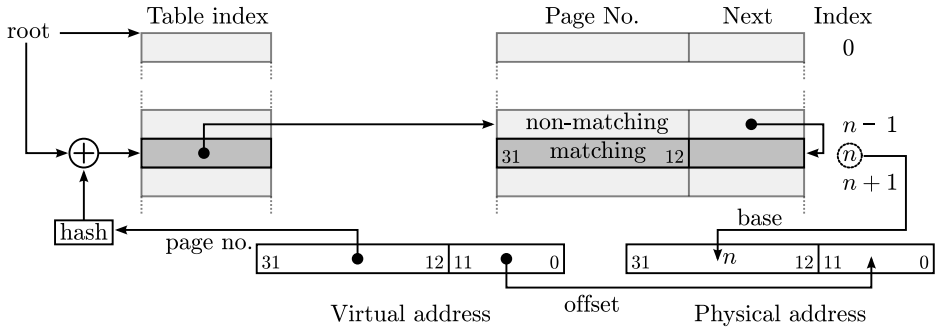


Figure 3.14: *A lookup in an inverted page table. A hashing function is applied to the page number of the virtual address and resolves to an index via an index table. After resolving a collision chain of length one the lookup successfully finds a table entry whose index forms the base physical address. Process identifiers omitted.*

of frames in the system. Therefore, if we store a page number in a table which has one entry per frame, we can hash the page number of the virtual address we are trying to look up, and then follow a collision chain if necessary to get to an entry whose page number and process identifier matches. The index at which we end up is the frame number we resolve to. If the chain ends without a match, a page fault is issued.

The IPT is fast and very memory-efficient. Unfortunately, using a hash table based on virtual page numbers *and* process identifiers makes shared memory difficult to manage.

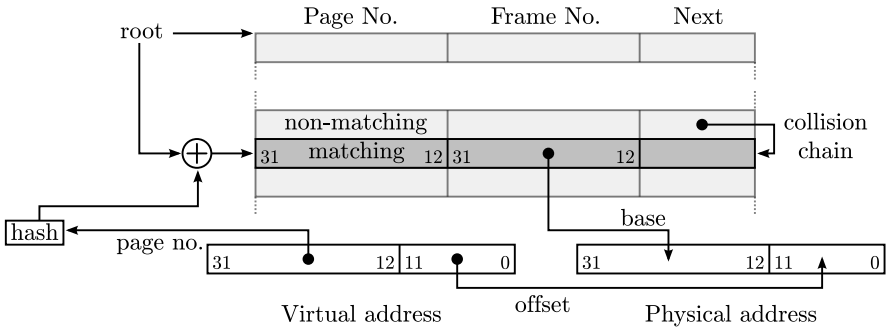


Figure 3.15: *A lookup in an hashed page table (HPT). Functionality is almost identical to the IPT it is based on, though there is no intermediate table of indices.*

In practice, such as on the relatively popular PowerPC, UltraSPARC and IA-64 ("Itanium") architectures, a variant of the IPT is used: the Hashed Page Table (HPT), pictured in Figure 3.15.

Formal reasoning about this page table model in our framework is difficult, as we based it in separation logic. As we will discuss in Chapter 4, the focus is on *separation* of resources. The idea of collision chains on the lookup path implies a dependency between an entry in the table and one that is upstream from it in a collision chain. While in theory it is possible to disentangle the dependencies, our belief is that this particular page table type is best modeled in a framework not based around separation logic. We are not aware of any formalisation or verification efforts based around IPTs.

3.5 Summary

In this chapter, we introduced virtual memory, its core concepts and mechanisms. We described how page table lookups resolve virtual addresses to physical ones, in detail for one and two-level page tables, and in general for other page table types. We also described an instance of a two-level page table on a specific architecture, the ARMv6.

As we mentioned at the start of this chapter, our work contains formalisations of a one-level page table in Chapter 5, and the ARMv6 page table in Chapter 6.

Chapter 4

Separation Logic

Separation logic, first introduced by Reynolds [48], is a logic for reasoning about programs which directly manage resources such as memory. It focuses on local reasoning via precise definitions of which areas of memory are modified during a program's operation and, importantly, which areas are *not*. This allows for effective reasoning about programs involving pointers, memory allocation, low-level data structure implementations, even some forms of concurrency.

In this chapter, we will first informally introduce the concepts behind separation logic, while in Section 4.2 we discuss what makes a logic a separation logic.

4.1 An Informal Introduction to Separation Logic

Our running example for this section will be a very simple factory, whose job it is to take objects and paint them. We will not go into details as to how exactly the factory manages this, focusing instead on the issues surrounding arrangement of machinery on the factory floor.

We will begin by introducing the factory scenario, then formalise it naïvely in Isabelle/HOL. By identifying issues with the naïve formalisation, we will introduce how separation logic addresses them and adjust the formalisation while introducing the core concepts of separation logic. As we progress with the informal introduction, we will establish that the factory floor is in fact an analogy for computer memory. This analogy will allow us to explain the concept of pointer aliasing in an informal setting.

4.1.1 The Factory

As in Figure 4.1, the factory floor consists of 16 squares, addressable by an index between 0 and 15. Each unit may be empty, or may contain one item of machinery. We will refer to the position with index `i` as `[i]`.

A *source* at `[i]` introduces objects to be painted into the factory (for instance, from below the floor) and passes them to the square at `[i+1]`. A *sink* at `[i]` receives objects from the square at `[i-1]` and removes them from the factory. In order to paint objects, they must pass through a *painter*. A painter at `[i]` takes objects from a source at `[i-1]`, paints them, and passes them to a sink at `[i+1]`. We define three painters, respectively named *red*, *green*, and *blue*. Each paints objects the colour it is named after. Figure 4.2 demonstrates a working production unit which paints

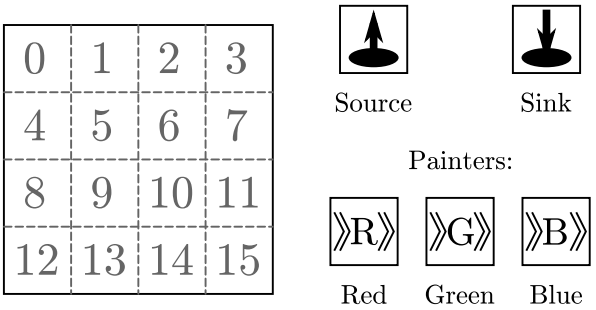


Figure 4.1: *Empty factory floor; factory components*

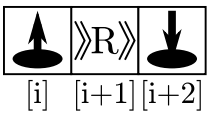


Figure 4.2: *A working production unit for painting objects red.*

objects red.

For manipulating the floor plan, we define two commands:

- *get*:
  `localvar := [address]`
  places the value at `[address]` into local variable `localvar`
- *set*:
  `[address] := val`
  places the value `val` into the square at `[address]`; `val` may be an immediate value or the value of a local variable.

In order to reason about the arrangement of machinery on the factory floor, we formalise its state. In this case, the state is a mapping from an index to the contents of the square it corresponds to. First, we know the types of machinery that can be placed in a square, thus we can define the square's contents as a data type:

```
datatype contents =
    Empty
  | RedPainter
  | BluePainter
  | GreenPainter
  | Source
  | Sink
```

Second, we know the index is between 0 and 15, containing exactly four bits of information. Let us represent it as `4 word`, i.e. a four-bit number. The state of the factory then has type `4 word ⇒ contents`. Since `Empty` is part of the contents description, we use a total function. An empty factory floor can then be defined as `λi. Empty`

Finally, in order to reason about effects of operations on the state, we add a predicate for describing the value at a given index. In the current formalisation, the value at index `p` in state `s` is simply `s p`. For `v` such that `s p = v`, we say that `p` maps to `v` in `s`, denoted by `(p ↪ v) s`.

```
_ ↪ _ :: 4 word ⇒ contents ⇒ (4 word ⇒ contents) ⇒ bool
p ↪ v ≡ λs. s p = v
```

Hoare triples are assertions on sequential programs. They feature a precondition, the program being executed, and a postcondition which holds after the program is executed provided the precondition held before the program was executed. Using our new maps-to predicate, we can define the semantics for our factory floor *get* and *set* operations as the Hoare triples in Figure 4.3. The pre- and postconditions of these triples talk about the factory floor, thus are predicates on the state and hence have the type `(4 word ⇒ contents) ⇒ bool`

- $\{p \hookrightarrow v\}$ `var := [p]` $\{\lambda s.\ var = v\}$
- $\{\lambda s.\ \mathsf{True}\}$ `[p] := var` $\{p \hookrightarrow var\}$

Figure 4.3: *Factory operational semantics for the naïve model.*

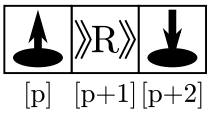
The factory floor is really a metaphor for (computer) memory; in this case, a 4-bit memory space. An index `p` refers to `[p]`, and `[p]` contains some value `v`. This is analogous to a memory pointer `p` pointing to memory location `[p]` containing the value `v`. We will henceforth refer to an index of a square on the factory floor as a *pointer* to that square.

4.1.2 Pointer Aliasing

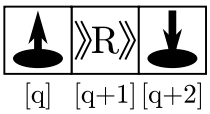
One of the big problems faced when reasoning about pointers and memory is pointer *aliasing*. We will now demonstrate this problem in the context of the factory floor.

The Problem

Suppose we have two pointers. Pointer `p` points to a production unit painting objects red:



Pointer `q` also points to a production unit painting objects red:



Suppose new orders come in, requiring our production units to be converted such that the unit at `p` paints objects green, and the unit at `q` paints objects blue. We now need to change the painters at `[p+1]` and `[q+1]` to `GreenPainter` and `BluePainter`. We issue the following two operations in sequence, with the following semantics:

$\{p + 1 \hookrightarrow RedPainter \lfloor\wedge\rfloor\ q + 1 \hookrightarrow RedPainter\}$
`[p+1] := GreenPainter ;`
`[q+1] := BluePainter`
$\{p + 1 \hookrightarrow GreenPainter \lfloor\wedge\rfloor\ q + 1 \hookrightarrow BluePainter\}$

where $\lfloor\wedge\rfloor$ is lifted conjunction:

$$P \lfloor\wedge\rfloor\ Q \equiv \lambda x.\ P\ x \wedge Q\ x$$

There is a problem here though. The Hoare triple as written is incorrect. We assumed the situation in Figure 4.4. However, if `p` and `q` both point to the *same* assembly unit, the result of the two operations will instead be the undesired post-condition of

$$\{\texttt{p + 1} \hookrightarrow \texttt{BluePainter} \lfloor\wedge\rfloor \texttt{ q + 1} \hookrightarrow \texttt{BluePainter}\}$$

This is known as the *aliasing problem*.

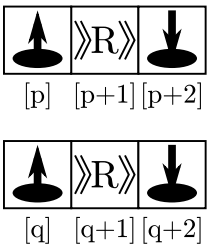


Figure 4.4: *Two production units pointed to by* `p` *and* `q`.

We can solve this particular problem by changing the precondition to require distinct `p` and `q`:

$$\{\lambda\texttt{s. (p + 1} \hookrightarrow \texttt{RedPainter} \lfloor\wedge\rfloor \texttt{ q + 1} \hookrightarrow \texttt{RedPainter) s} \wedge \texttt{ p} \neq \texttt{ q}\}$$

The requirement of stating that pointers point to *distinct* resources is easy when reasoning about only two pointers. Dealing with more becomes cumbersome. Furthermore, if we build up more complex structures than those consisting of only one factory square, we need to worry about *overlap* even if the pointers to their beginning are distinct.

For more complex structures in memory, the proof obligations increase. For instance, in a classical tree structure, one must not only ensure that a parent node does not point to the same child node twice, but that no two parents point to the same child.

The Separation Logic Approach

What if we had the ability to state our assumptions in a manner that directly implied that pointers point to distinct, non-overlapping memory areas? Separation logic provides this by local reasoning on partial *heaps*. In our factory example, this means considering areas of the factory floor separately, applying each predicate to a different area of the floor.

In Figure 4.4 we clearly see two areas of factory floor, one for each production unit. To solve the aliasing problem in our factory, we can simply divide the factory floor in two and "give" each maps-to predicate its own piece.

Since our representation of the state of the factory floor is currently a total function from indices to valid values, we cannot perform this division. To be able to describe parts of the factory floor by themselves, we need the factory floor state to become a partial function.

We create partial functions from total functions by making their range the option type:

datatype `'a option = None | Some 'a`

such that in a partial function `f`, for some `x`, `f x` is either None or `⌊v⌋`. The function's domain is the set of all arguments for which the function results in `⌊v⌋`:

$$\text{dom } f = \{a \mid f\ a \neq \text{None}\}$$

We change the type of the factory floor state to `4 word ⇒ contents option`, which we will write as `4 word ⇀ contents`. Now we can refer to specific areas of the factory floor. In separation logic, this partial function of resources to values is referred to as a *heap*. Our factory floor is therefore a heap.

We can now divide the factory floor into areas, but we still need the ability to say that predicates hold on separate areas. In other words, instead of saying "p maps to _ and q maps to _", we say "p maps to _ and *separately* q maps to _". In separation logic, this alternate form of conjunction is referred to as *separating conjunction*. We will denote it as $\wedge^*$. It is also sometimes referred to as "star" [48].

For `(P ∧* Q) h` to be true for a *heap* `h`, we must be able to break `h` up into two partial heaps h_0 such that:

- Their domains must be disjoint:
  $h_0 \perp h_1$, where $h_0 \perp h_1 \equiv \text{dom } h_0 \cap \text{dom } h_1 = \emptyset$
- When joined together they must re-form `h`:
  h_0 `++` h_1 `= h`
  where
  h_0 `++` h_1 `= (λx. case` h_1 `x of None ⇒` h_0 `x | ⌊y⌋ ⇒ ⌊y⌋)`
- `P` h_0
- `Q` h_1

This results in the formal definition in Figure 4.5. In the case of our factory, we instantiate the heap type `'a ⇀ 'b` with our factory floor type: `4 word ⇀ contents`.

```
_ ∧* _ :: (('a ⇀ 'b) ⇒ bool) ⇒ (('a ⇀ 'b) ⇒ bool) ⇒ ('a ⇀ 'b) ⇒ bool
P ∧* Q ≡ λh. ∃h0 h1. h0 ⊥ h1 ∧ h = h0 ++ h1 ∧ P h0 ∧ Q h1
```

Figure 4.5: *Separating conjunction*

Our old maps-to operator dealt with the total function representing the factory floor. We need to change it to deal with parts of the floor, i.e. partial heaps:

```
_ ↪ _ :: 4 word ⇒ contents ⇒ (4 word ⇀ contents) ⇒ bool
p ↪ v ≡ λh. h p = ⌊v⌋
```

Using separating conjunction, we can rephrase our problematic statement from the beginning of this section (4.1.2) correctly as:

```
{p + 1 ↪ RedPainter ∧* q + 1 ↪ RedPainter}
 [p+1] := GreenPainter ;
 [q+1] := BluePainter
{p + 1 ↪ GreenPainter ∧* q + 1 ↪ BluePainter}
```

Now if `p=q`, we cannot construct two partial heaps such that `[p]` is in one and `[q]` in the other, resulting in a false precondition for the Hoare triple.

The separating conjunction concept scales to an arbitrary number of pointers and removes the possibility of overlap, significantly reducing the work needed to reason about aliasing of locations on the heap.

4.1.3 Local Reasoning and Memory Safety

At the beginning of this chapter, we mentioned local reasoning, i.e. reasoning about precisely what changes, but also what does *not* change. In the previous example, we divide the heap into two sub-heaps: one for `p` and one for `q`. However, the size of these partial heaps is not strictly constrained, and thus says nothing about what our program *has not* changed. For instance, adding superfluous information to the precondition still results in a true statement:

```
{p + 1 ↪ RedPainter ∧* q + 1 ↪ RedPainter ∧* t ↪ Empty}
 [p+1] := GreenPainter ;
 [q+1] := BluePainter
{p + 1 ↪ GreenPainter ∧* q + 1 ↪ BluePainter}
```

The postcondition can talk about a completely different heap than the assumption. We can also omit information, as the following is likewise correct:

```
{p + 1 ↪ RedPainter}
 [p+1] := GreenPainter ;
 [q+1] := BluePainter
{p + 1 ↪ GreenPainter}
```

These truths do not form a basis for local reasoning, as our *weak* maps-to predicate, apart from the mapping it talks about, will happily consume *any* information about other areas of the heap. The resulting rules have imprecise preconditions and weak postconditions. They combine poorly and complicate the creation of tools for automated reasoning, such as weakest-precondition or verification condition generators.

In order to address this shortcoming, we must restrict our maps-to predicate to only the area of the heap that contains the mapping itself. In other words, a mapping of address 42 to value 7 must only be true on a partial heap whose domain is `{42}` and whose value at `[42]` is 7. Making the heap smaller or bigger must make the predicate false. The predicate in Figure 4.6 is the basic maps-to predicate that forms the basis of separation logic.

```
_ ↦ _ :: 4 word ⇒ contents ⇒ (4 word ⇀ contents) ⇒ bool
p ↦ v ≡ λh. h p = ⌊v⌋ ∧ dom h = {p}
```

Figure 4.6: *Basic maps-to predicate of separation logic.*

Using a precise version of the maps-to predicate allows us to reason about programs in a modular fashion, focusing on their *local* properties.

For example, in Figure 4.7 we restate the rule about placing machinery on the factory floor, the assignment rule, using the new maps-to predicate where, for convenience, `p ↦ -` means "maps to *anything*", i.e. `λh. ∃v. (p ↦ v) h`

This rule *clearly* states that, given a heap with only `[p]`, the assignment will set `[p]` to `var` and *do nothing else*. In terms of the factory, this will place object `var` at the

$$\{p \mapsto -\}$$
```
 [p] := var
```
$$\{p \mapsto var\}$$

Figure 4.7: *Updated assignment rule.*

square on the factory floor at index p. We can be certain that the assignment is both unaffected by and does not affect the rest of the factory floor.

Most rules of Hoare Logic apply to separation logic. However, given separation logic predicates P, Q and R, the rule of constancy does not hold:

$$\frac{\{P\}\ program\ \{Q\}}{\{P \lfloor\wedge\rfloor R\}\ program\ \{Q \lfloor\wedge\rfloor R\}}$$

By separation logic predicates we mean any predicate that restricts the size of heaps it is true on. Combining such predicates with normal conjunction does not work, e.g. our precise maps-to predicate is true only for heaps with a domain of cardinality 1. Combining it with another precise maps-to predicate may result in requiring a domain of two addresses, but of cardinality 1, which is impossible. The rule of constancy is only true if R is *pure*, i.e. does not make any assertion about the heap.

$$\frac{\{P\}\ program\ \{Q\}}{\{P \wedge^* R\}\ program\ \{Q \wedge^* R\}}$$

Figure 4.8: *The frame rule.*

What separation logic offers in place of the constancy rule is the *frame rule* (Figure 4.8). The frame rule, in conjunction with separation logic maps-to predicates allows us to effectively combine programs via local reasoning. The R in the frame rule no longer needs to be pure, but it must talk about a separate region of the heap from P and Q.

We will now revisit the problematic statement from the beginning of this section (4.1.2):

```
[p+1] := GreenPainter ;
[q+1] := BluePainter
```

We can instantiate our new assignment rule to the two assignment statements, we get the following Hoare triples:

$\{p + 1 \mapsto RedPainter\}$
```
 [p+1] := GreenPainter
```
$\{p + 1 \mapsto GreenPainter\}$

$\{q + 1 \mapsto RedPainter\}$
```
 [q+1] := BluePainter
```
$\{q + 1 \mapsto BluePainter\}$

We apply the frame rule to both:

$\{p + 1 \mapsto RedPainter \wedge^* R\}$
```
 [p+1] := GreenPainter
```
$\{p + 1 \mapsto GreenPainter \wedge^* R\}$

$\{q + 1 \mapsto RedPainter \wedge^* R'\}$
```
 [q+1] := BluePainter
```
$\{q + 1 \mapsto BluePainter \wedge^* R'\}$

We then match up the postcondition of the first assignment with the precondition of the second:

$\{p + 1 \mapsto \mathit{RedPainter} \wedge^* q + 1 \mapsto \mathit{RedPainter}\}$
[p+1] := GreenPainter
$\{p + 1 \mapsto \mathit{GreenPainter} \wedge^* q + 1 \mapsto \mathit{RedPainter}\}$

$\{q + 1 \mapsto \mathit{RedPainter} \wedge^* p + 1 \mapsto \mathit{GreenPainter}\}$
[q+1] := BluePainter
$\{q + 1 \mapsto \mathit{BluePainter} \wedge^* p + 1 \mapsto \mathit{GreenPainter}\}$

As the postcondition of the first assignment is now the precondition of the second, we can combine the two statements into one statement that only touches the *distinct* addresses `[p+1]` and `[q+1]`:

$\{p + 1 \mapsto \mathit{RedPainter} \wedge^* q + 1 \mapsto \mathit{RedPainter}\}$
[p+1] := GreenPainter
$\{p + 1 \mapsto \mathit{GreenPainter} \wedge^* q + 1 \mapsto \mathit{RedPainter}\}$
[q+1] := BluePainter
$\{p + 1 \mapsto \mathit{GreenPainter} \wedge^* q + 1 \mapsto \mathit{BluePainter}\}$

Note that during this procedure, we did not mention the aliasing problem at all.

4.2 Requirements of Being a Separation Logic

Our informal example gives an intuition for the goals and approaches of separation logic. It does not, however, qualify what *exactly* constitutes a separation logic. Calcagno et al. [13] provide a formal definition and requirements. This section attempts to summarises those requirements and discuss the features relevant to our work.

Firstly, the concept of separation must talk about the management of a set of independent, separable resources. Each of these resources is optionally assigned a value. We call the partial function from the domain of resources onto the range of possible values the *state*. For example, in Section 4.1.3 we used a partial memory-to-value heap as the state. The logic must define the notion of disjointness of states (denoted in our work as $\perp$), as well as state composition ($+\!+$), with conditions as per Definition 4.2.1.

Definition 4.2.1 *Commutativity and associativity of the composition operator when applied to disjoint states.* $s +\!+ s'$ *denotes the composition of states* s *and* s'. $s \perp s'$ *denotes the disjointness of states* s *and* s'.

$$s \perp s' \longrightarrow s +\!+ s' = s' +\!+ s$$
$$s_0 \perp s_1 \wedge s_1 \perp s_2 \longrightarrow (s_0 +\!+ s_1) +\!+ s_2 = s_0 +\!+ (s_1 +\!+ s_2)$$

Secondly, in order to be able to accurately perform local reasoning on states and their sub-states, the basic predicates used should restrict their scope to only be true on states of a specific, minimal, state. For example, the precise mapping predicate we introduced in Section 4.1.3, Figure 4.6 has this property.

Finally, we must satisfy requirements for local reasoning. These requirements apply to the semantics of the programming language used in the logic. Calcagno et al. [13] define them as *safety monotonicity* and the *frame property*. To show these for all possible statements in a programming language, the language must be aware of the allocation of resources in the state. In other words, attempts to access resources

which are not part of the current state must result in *failure* of the program. Let us denote the execution of a program `c` on state `s` as `c s`. If there are insufficient resources to complete the execution, we will indicate failure as `c s = Abort`. If the program succeeds, it returns a new, valid (non-abort) state: `c s = Valid s'`.

Safety monotonicity, defined in Definition 4.2.2, states that if a program in the language succeeds on some state, i.e. has all the resources necessary to execute, then enlarging the state with more resources will have no effect upon the program's outcome. The frame property, defined in Definition 4.2.3, states that given a successful execution of a program on a small state and also on a larger state, we can trace back the execution to local reasoning about execution on the smaller state.

Definition 4.2.2 *Safety monotonicity for a program* `c`.

$$[\![\ c\ s_0 \neq Abort;\ s_0 \perp s_1\ ]\!] \Longrightarrow c\ (s_0\ +\!+\ s_1) \neq Abort$$

Definition 4.2.3 *The frame property for a program* `c`.

$$[\![\ c\ s_0 \neq Abort;\ c\ (s_0\ +\!+\ s_1) = Valid\ s';\ s_0 \perp s_1\ ]\!]$$
$$\Longrightarrow \exists s_0'.\ s' = s_0'\ +\!+\ s_1 \wedge c\ s_0 = s_0'$$

Safety monotonicity and the frame property can be defined and shown to hold for memory-safe language semantics. Memory safety means awareness of which memory areas are allocated/accessible and which are not. This awareness allows abnormal termination upon accessing an unallocated memory area, for example `Abort` as in Definition 4.2.2. In such a semantics we can reason about addresses in a heap as being "allocated" or "valid" and derive local rules for the exact amount of allocated addresses a program needs to execute successfully, hence the size of the heap.

Unfortunately, not all programming language semantics involve memory safety. For example, the C programming language standard defines accesses to unallocated memory as *undefined behaviour*. In practice, nearly all implementations of the C programming language do not keep track of allocated memory. All reads and writes to memory are treated in an identical fashion and performed, regardless of allocation status. In this case, a formalisation of C that results in `Abort` upon accessing unallocated memory would not be a true reflection of C semantics, while a formalisation which allows arbitrary memory reads and writes will neither have safety monotonicity nor the frame property.

Definition 4.2.4 *The frame rule for a program* `c` *and heap predicates* `P`, `Q` *and* `R`.

$$\frac{\{P\}\ c\ \{Q\}}{\{P\ \wedge^*\ R\}\ c\ \{Q\ \wedge^*\ R\}}$$

Fortunately, there is a way out. Safety monotonicity and the frame property can be shown to be equivalent to the frame rule, Definition 4.2.4. This means that even if we cannot prove safety monotonicity and the frame property for the language semantics itself, we can still prove locality by showing the frame rule holds for specific programs. In this manner, we can prove the frame rule on a function-by-function or component-by-component basis, and then compose them in the manner demonstrated in Section 4.1.2.

4.3 Summary

Beginning with an informal setting, we have now introduced the core concepts of what makes a separation logic and hopefully imparted why it can be useful.

To summarise, we can think of a separation logic is a logic where:

- Predicates about the heap specify precisely the size and contents of the heap on which they hold.
- Heap disjunction ($\_ \perp \_$) and heap composition ($\_ +\!+ \_$) operators are defined. The separating conjunction operator is defined using these.
- The safety monotonicity and frame property hold on the language in which programs are written, or at least they can be proven for each program in the language. As a result, the frame rule holds.

The power of separation logic lies in its local reasoning. By defining what changes in a precise way, we can infer what does not change. As a result, the benefits of picking the building blocks well allow for convenient use of modularity in an environment that involves pointers and raw access to memory.

Chapter 5

Mapped Separation Logic

In the previous two chapters, we introduced the virtual memory abstraction, as well as separation logic. In this chapter, we outline their combination, the core idea of our work, into a new logic we call Mapped Separation Logic.

We will analyse the general problem, but instantiate it to a very simple form: a simplified machine architecture supporting virtual memory, with a one-level page table. This simplicity allows us to focus on the core issue: the fusion of virtual memory and separation logic concepts into new predicates; the identification of what exactly *separate* means in virtual memory. At the same time, we will identify the core concepts that will work for other architectures and page table models.

From there we move on to define predicates analogous to those of Separation Logic, which we then use to perform a small case study using a simple programming language. We prove that the combination of the language and predicates forms a separation logic by showing it has safety monotonicity and the frame property, as required by Calcagno et al. [13].

We have already published most of the work in this chapter as *Mapped Separation Logic* [33]. We expand on some points, but our message remains the same.

5.1 Machine Architecture

Our choice of architecture in the case study is a slightly modified 32-bit machine. That is, the basic unit of data manipulation, the machine word, is 32 bits in size. Typically, a machine's registers and pointers to memory are the same size as the machine word, in this case 32 bits. We follow this scheme, but do not formalise registers, in favour of focusing specifically on memory.

The non-standard part of our machine setup is the memory layout. While our memory is still a map from a machine word to a value, we chose a value size of 32-bits, the size of the machine word, rather than the usual 8-bit byte value. This simplifies assembling pointers from memory values and avoids complications such as endianness, allowing us better focus on our goal.

5.1.1 Pointers, Addresses and Values

Our 32-bit machine setup means that all pointers will be 32 bits in size. Since the machine supports virtual memory, it manipulates pointers to both the physical and virtual address spaces. The former allows direct access to physical memory, while

the latter involves a translation via the page table, which may fail. For this reason, although to the machine they are simply 32-bit words, it is important that we never confuse a virtual pointer with a physical one.

We therefore define two distinct pointer types for any type of address:

```
datatype 'a pptr_t = PPtr 'a
datatype 'a vptr_t = VPtr 'a
```

as well as their destructor functions for talking about a physical/virtual address directly:

```
pptr_val :: 'a pptr_t ⇒ 'a
pptr_val (PPtr x) = x

vptr_val :: 'a vptr_t ⇒ 'a
vptr_val (VPtr x) = x
```

The overloaded accessor function ptr_val combines both of the above, working on either virtual or physical pointers.

Much of the framework presented in this chapter is generic over the size of addresses and values. For our 32-bit machine we instantiate them as follows:

```
types  addr  =  32 word
       pptr  =  addr pptr_t
       vptr  =  addr vptr_t
       val   =  32 word
```

In other words: addresses are 32-bit words, pointers are addresses with extra type information. Values stored in memory are also 32-bit words.

5.1.2 Memory

The addition of virtual memory requires we consider three concepts related to memory: the physical address space, the virtual address space, and the translation from virtual to physical addresses.

The physical address space represents the physical memory contained in the machine. Hence it is map from physical addresses to values. Since we will be dealing with separation logic concepts, we make it a partial map as in Chapter 4, in order to talk about specific areas of physical memory in isolation. Physical memory also represents the resource we will be managing. Therefore, in this chapter we use the word *heap* to refer to physical memory. For conciseness we will refer to the virtual-to-physical translation as the *virtual map*, and to the virtual address space as the *address space*. The relevant types are:

```
types  heap        =  pptr ⇀ val
       vmap        =  vptr ⇀ pptr
       addr_space  =  vptr ⇀ val
```

5.1.3 A Page Table

As described in Chapter 3, the page table is an encoding of the translation from virtual to physical addresses. It resides in physical memory, i.e. the *heap*, at a location we call the *root*.

In our development, we abstract the concept of the page table as much as possible, such that our logic remains mostly independent of the particular encoding used. Our abstract page table interface requires two functions:

```
ptable_lift  :: heap ⇒ pptr ⇒ vmap
ptable_trace :: heap ⇒ pptr ⇒ vptr ⇒ pptr set
```

Both functions take the physical heap and page table root as parameters. The result of ptable_lift is the *vmap* encoded by the page table, whereas ptable_trace returns for each virtual address *vp* the set of locations in the page table that are involved in the lookup of *vp*. Naturally, for a one-level page table, this set will only have one element, but other page table models, such as the two-level page table we will use in Chapter 6, have more complicated lookup paths. In contrast to our earlier work [32], we do not require an explicit formulation of the page table area in memory.

We will later require five constraints on these two functions. We defer the presentation of these to Section 5.3 when the associated concepts have been introduced.

As mentioned earlier, we use a simple one-level page table as an example instantiation. As in Chapter 3, Section 3.2.1, it is a contiguous physical memory structure consisting of an array of machine word pointers, where word 0 defines the physical location of page 0 in the address space, word 1 that of page 1 and so forth. While inefficient in terms of storage, it is simple to present and experiment with. The table is based on an arbitrarily chosen page size of 4096, i.e. 20 bits for the page number and 12 for the offset. Page table lookup works as expected: we extract the page number from the virtual address, go to that offset in the page table and obtain a physical frame number which replaces the top 20 bits of the address:

```
get_page :: vptr ⇒ 32 word
get_page vp              ≡ ptr_val vp >> 12
ptr_remap :: vptr ⇒ 32 word ⇒ pptr
ptr_remap (VPtr vp) pg ≡ PPtr (pg AND NOT 0xFFF OR vp AND 0xFFF)

ptable_lift :: (pptr ⇀ 32 word) ⇒ pptr ⇒ vptr ⇀ pptr
ptable_lift h r vp       ≡ case h (r + get_page vp) of None ⇒ None
                             | Some addr ⇒
                                 if addr !! 0 then Some (ptr_remap vp addr)
                                 else None
ptable_trace :: (pptr ⇀ 32 word) ⇒ pptr ⇒ vptr ⇒ pptr ⇒ bool
ptable_trace h r vp      ≡ case h (r + get_page vp) of None ⇒ ∅
                             | Some addr ⇒ {r + get_page vp}
```

AND, OR and NOT are bitwise operations on words. The operator *>>* is bitwise right-shift on words. The term *x !! n* stands for bit *n* in word *x*. Since the bottom 12 bits of the page table entry are unused, we arbitrarily choose to use bit 0 to denote whether a page table entry contains a valid mapping.

5.2 Separation Logic Assertions on Virtual Memory

Thus far, we have defined a 32-bit machine and associated concepts. Apart from our use of the word *heap*, however, we have not raised any issues related to separation logic. In this section, we discuss the extension of classical Separation Logic assertions to hold for a model with virtual memory.

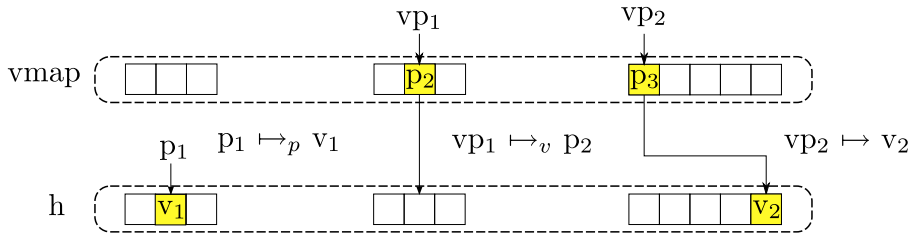


Figure 5.1: *Maps-to assertions on the heap, virtual map and address space.*

5.2.1 The Problem

As we mentioned in Chapter 3, a program accesses memory on an architecture supporting virtual memory via virtual addresses, which involves the translation mechanism, including lookups in the page table. A separation logic used to reason about such a program must hence involve the concept of separation between mappings of virtual addresses to values. This poses a problem, which we will explain below.

In Chapter 4, we discussed two central tools of Separation Logic: the separating conjunction and the frame rule. Separating conjunction $P \wedge^* Q$ is an assertion on heaps, stating that the heap can be split into two separate parts on which the conjuncts P and Q hold respectively. We also say P and Q *consume* disjoint parts of the heap. Separating conjunction conveniently expresses anti-aliasing conditions. For an action f, the frame rule allows us to conclude $\{P \wedge^* R\}\, f\, \{Q \wedge^* R\}$ from $\{P\}\, f\, \{Q\}$ for any R. This expresses that the actions of f are local to the heaps described by P and Q, and can therefore not affect any separate heaps described by R.

Unfortunately, given the additional layer of indirection provided by virtual memory, two different virtual addresses may resolve to the same physical address, which breaks separating conjunction, as on these addresses it does not end up providing separation. Additionally, although a memory update to the page table may only locally change one value in physical memory, it might completely change the view the virtual memory layer provides, affecting a whole number of seemingly unrelated virtual addresses. A local action might therefore have non-local effects. This breaks the frame rule. In the remainder of this chapter, we show how to repair both of these tools while remaining within a virtual memory context.

5.2.2 Slices and the Fractional Heap

As mentioned in the previous section, the common case of reasoning about programs in virtual memory involves mappings from virtual addresses to values in physical memory. We can consider such virtual-to-value mappings as the composition of two other mapping types: the virtual-to-physical mapping, resolving a virtual address to a physical one, and a physical-to-value mapping, declaring a physical memory location to hold a particular value. This requires three corresponding maps-to assertions, shown in Figure 5.1.

As mentioned in Chapter 4, predicates about the heap *consume* parts of it under separating conjunction, i.e. if a predicate states something about a part of the heap, one cannot use separating conjunction to combine it with a predicate involving

that same part of the heap. In our case, it is quite clear that the physical-to-value mapping consumes one heap address which contains some value. However, the question central to the issue is *which part of the heap should a virtual-to-physical assertion consume?*

This question determines the meaning of separation between this and other assertions. Assuming for example a one-level page table where memory location x encodes the lookup of virtual address vp_1 to physical address p_2, there are two cases: (a) the lookup consumes x, or (b) it does not. In case (a) we cannot use separating conjunction to specify separation of vp_1 and another address vp_2 if vp_2 happens to use the same page table entry x. Typically, many virtual addresses share an entry in the encoding, and we clearly do not want to exclude this case. The other extreme, in case (b), would be to say the lookup consumes no resources. In earlier work [32] we came to the conclusion that this model cannot support the frame rule for arbitrary programs: a physical write to the page table would be separate to any virtual-to-physical mapping, but the write might have the non-local effect of changing that apparently separate mapping.

The only way out, and hence our proposal, is to have a page table lookup consume only *parts* of the page table entry involved. Since a page table entry consists of bytes in physical memory, this can be seen as cutting each physical address, i.e. heap location, into multiple slices. If a page table lookup, i.e. a virtual-to-physical mapping, consumes only the parts of the page table entry involved in the lookup, we can make all memory updates local.

This idea is similar to the model of permissions by Bornat et al. [10] for concurrent threads. Their permission model involves mentioning the size of the fraction of a resource consumed each time a mapping predicate is used, as well as placing information on fractional resource usage in the range of the heap. The latter complicates usage of existing infrastructure of maps provided by tools such as the Isabelle theorem prover, which we use in this work. Although this model will work, we make two observations. Firstly, the common case for interaction with virtual memory does not involve manipulation of the page table. Secondly, considering all possible page table models, there is a finite upper bound on the amount of addresses that can be sharing a page table entry: 2^n, where n is the number of bits in a virtual address. This upper bound can only arise in the pathological case of a page table in which one page table entry maps *all* virtual addresses. These observations allow us to create a model more optimised for the common case, which we will explain presently. We maintain the Bornat concept of full permissions being required for writing to the heap, while allowing reading with only partial access to a heap location.

In the worst-case sharing scenario we outlined above, on a 32-bit machine there will be 2^{32} virtual addresses whose lookups resolve through the same page table entry. In terms of resource consumption, we can visualise the situation as follows. Lookups of all 2^{32} virtual addresses need access to that page table entry in order to succeed. Dividing the single page table entry into 2^{32} slices and assigning one slice to each virtual address allows each address' lookup to use one slice of the entry. At this point we can start talking about separation. Two virtual-to-physical mappings are separate if the lookups of the virtual addresses do not use the same slice in the page table entry. Two physical-to-value mappings are separate if the physical addresses are distinct. We will address the concept of virtual-to-value mappings later in this section, as they introduce another issue not directly relevant to discussion of slices.

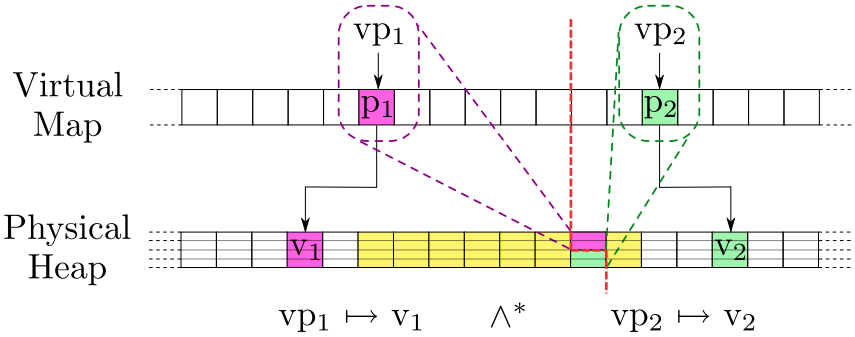


Figure 5.2: *The concept of two virtual-to-value mappings being separate in our framework despite being mapped through the same page table entry by virtue of using separate slices of the page table entry (the contested region in the middle).*

To conclude the pathological example, we observe that if there are 2^{32} virtual addresses and 2^{32} slices of the page table entry, we can trivially assign them one-for-one. Since we do not know where in physical memory this entry is located, *any* physical address in the heap may be a potential page table entry.

for any physical address `p` we need to be able to address the slice of `p` responsible for virtual address `0, 1, ... :: vptr` etc. Thus, the domain of the heap becomes `pptr × vptr` and `(p, vp)` stands for the `vp`-slice of physical address `p`. We call this a *fractional heap*:

types `fheap = (pptr × vptr) ⇀ val`

Applying this idea of a fractional heap to other page table models allows us to conclude that two virtual-to-physical or virtual-to-value mappings are separate even if they are mapped by the same page table entry, provided that they do not consume the same slices of the entry. An intuitive representation of this situation appears in Figure 5.2. The common case of memory access in a virtual memory system does not involve manipulating or accessing the page table directly. In these cases, we do not need to mention how much and which part of a page table entry a virtual-to-physical mapping consumes. It is always the same: lookup of virtual address `vp` consumes slice `vp` of every physical address required for the lookup, i.e. of the page table trace. Thus the memory footprint of page table lookups has a direct formulation in our model.

The consequences of this are three-fold. Firstly, the basic definition of heap merge (`++`), domain, and disjointness (⊥) remain completely standard:

```
_ ++ _ :: ('a ⇀ 'b) ⇒ ('a ⇀ 'b) ⇒ 'a ⇀ 'b
h1 ++ h2 ≡ λx. case h2 x of None ⇒ h1 x | Some y ⇒ Some y
dom :: ('a ⇀ 'b) ⇒ 'a ⇒ bool
dom h ≡ {x | h x ≠ None}
_ ⊥ _ :: ('a ⇀ 'b) ⇒ ('a ⇀ 'b) ⇒ bool
h0 ⊥ h1 ≡ dom h0 ∩ dom h1 = ∅
```

Secondly, in the cases when we do manipulate the page table, we will need to deal with all slices of a page table entry, including those that are not involved in resolving any virtual addresses. In our one-level page table example, each page table entry

resolves only 4096 virtual addresses, much smaller than the worst-case scenario of 2^{32}, meaning we need to keep track of the extra slices. This is similar to making sure that the permissions add up to 1 in the Bornat model.
Finally, the fractional heap allows the possibility of expressing an invalid memory state: two distinct slices of the same physical address mapping to distinct values. As we require full permissions, i.e. presence of all slices of a physical address to perform a write, this does not pose a problem.

5.2.3 Assertions

Now that we have a fractional heap to work with, we can start building up the three maps-to assertions mentioned in Section 5.2.2. Although a fractional heap is sufficient to express the state of physical memory of a system, a maps-to assertion, in particular one related to virtual addresses, needs the physical location of the page table that will be used for lookups:

```
types  map_assert = (fheap × pptr) ⇒ bool
```

The Separation Logic connectives for empty heap, true, and conjunction stay almost unchanged. We additionally supply the page table root `r`.

```
□ :: ('a × 'b ⇀ 'c) × 'd ⇒ bool
□      ≡ λ(h, r). h = empty
⊤ :: ('a × 'b ⇀ 'c) × 'd ⇒ bool
⊤      ≡ λ(h, r). True
_ ∧* _ :: (('a × 'b ⇀ 'c) × 'd ⇒ bool)
          ⇒ (('a × 'b ⇀ 'c) × 'd ⇒ bool) ⇒ ('a × 'b ⇀ 'c) × 'd ⇒ bool
P ∧* Q ≡ λ(h, r). ∃h0 h1. h0 ⊥ h1 ∧ h = h0 ++ h1 ∧ P (h0, r) ∧ Q (h1, r)
```

Since the definitions are almost unchanged it is unsurprising that the usual properties such as commutativity and associativity and distribution over lifted normal conjunction continue to hold.

The interesting, new assertions are the three *maps-to* statements that we alluded to in Figure 5.1. Corresponding to the traditional Separation Logic predicate is the physical-to-value mapping:

```
_ ↦p _ :: pptr ⇒ 32 word ⇒ fheap × pptr ⇒ bool
p ↦p v ≡ λ(h, r). (∀vp. h (p, vp) = Some v) ∧ dom h = {p} × UNIV
```

For this assertion, we require that all slices of the heap at physical address `p` map to the value `v` and that the domain of the heap is exactly the set of all pairs with `p` as the first component (UNIV is the universe set). This predicate would typically be used for direct memory access in devices or for low-level kernel operations.

The next assertion is the virtual-to-physical mapping:

```
_ ↦v _ :: vptr ⇒ pptr ⇒ fheap × pptr ⇒ bool
vp ↦v p ≡ λ(h, r).
            let heap = h_view h vp; vmap = ptable_lift heap r
            in vmap vp = Some p ∧ dom h = ptable_trace heap r vp × {vp}
```

where `h_view fh vp ≡ λp. fh (p, vp)`. Here, we first lift the page table out of the heap to get the abstract `vmap` which provides us with the translation from `vp` to `p`.

Additionally, we assert that the domain of the heap is the vp slice of all page table entries that are involved in the lookup. With h_view we project out the vp slice for all addresses so that the page table lift function can work on a traditional $pptr \rightharpoonup val$ heap.

Putting the two together, we arrive at the virtual-to-value mapping that corresponds to the level that most of the operating system and user code will be reasoning at.

$$\_ \mapsto \_ :: vptr \Rightarrow 32\ word \Rightarrow fheap \times pptr \Rightarrow bool$$
$$vp \mapsto v \equiv \lfloor\exists\rfloor p.\ vp \mapsto_v p \wedge^* p \mapsto_p v$$

where $\lfloor\exists\rfloor x.\ P\ x \equiv \lambda s.\ \exists x.\ P\ x\ s$. The predicate implies that the lookup path is separate from the physical address p the value v is at. This is the case for all situations we have encountered so far and, as the case study shows, works for page table manipulations as well. It is possible to define a weaker predicate without the separation, but this creates a special case for the assignment rule: if a write changes the page table for the address the write goes to, the post condition would have to take the change in the translation layer into account directly.

The usual variations on the maps-to predicate can be defined in the standard manner again for virtual-to-value mappings:

$$\_ \mapsto - :: vptr \Rightarrow fheap \times pptr \Rightarrow bool$$
$$p \mapsto - \equiv \lfloor\exists\rfloor v.\ p \mapsto v$$
$$\_ \hookrightarrow \_ :: vptr \Rightarrow 32\ word \Rightarrow fheap \times pptr \Rightarrow bool$$
$$p \hookrightarrow v \equiv p \mapsto v \wedge^* \top$$
$$\_\ \{\mapsto\}\ - :: (vptr \Rightarrow bool) \Rightarrow 32\ word \Rightarrow fheap \times pptr \Rightarrow bool$$
$$p\ \{\mapsto\}\ - \equiv \text{fold\_image}\ op\ \wedge^*\ (\lambda x.\ x \mapsto -)\ \square\ S$$

The latter definition refers to a finite set S of addresses. It states that all of the elements map separately by folding the $\wedge^*$ operator over S and the maps-to predicate. There are analogous variations for physical-to-value and virtual-to-physical mappings.

We have proved that our basic mappings $\mapsto_p$ and $\mapsto_v$ are domain exact [48]. Note that, although not domain exact due to the existential quantifier on p, the virtual-to-value mapping is still precise [45]:

$$\text{precise}\ P \equiv \forall Q\ R.\ (P \wedge^* Q\ \lfloor\wedge\rfloor\ R) = ((P \wedge^* Q)\ \lfloor\wedge\rfloor\ (P \wedge^* R))$$

That is, it distributes over conjunction in both directions. We write $P\ \lfloor\wedge\rfloor\ Q$ for the lifted conjunction of assertions P and Q: $P\ \lfloor\wedge\rfloor\ Q \equiv \lambda x.\ P\ x \wedge Q\ x$

5.3 The Logic

Having introduced the basic and mapping predicates, we can now build a separation logic. To do so, we require a programming language. This section introduces a simple, heap based programming language with pointer arithmetic to analyse how the assertions presented above can be developed into a full Separation Logic. For the meta-level proofs in this chapter, we provide a deep embedding of the language into Isabelle/HOL. The language is standard, with skip, if, while, and assignment. The WHILE and IF statements potentially read from virtual memory in their guards. Assignment is the most interesting: it accesses memory through the virtual memory layer and can potentially modify this translation by writing to the page table. We

```
datatype aexp =
        HeapLookup aexp
    |   BinOp (val ⇒ val ⇒ val) aexp aexp
    |   UnOp (val ⇒ val) aexp
    |   Const val

datatype com =
        SKIP
    |   aexp := aexp
    |   com; com
    |   IF bexp THEN com ELSE com
    |   WHILE bexp DO com

datatype bexp =
        BConst bool
    |   BComp (val ⇒ val ⇒ bool) aexp aexp
    |   BBinOp (bool ⇒ bool ⇒ bool) bexp bexp
    |   BNot bexp
```

Figure 5.3: *Syntax of the heap based WHILE language.*

leave out the simpler physical write access for the presentation here. It would be easy to add and does not increase the complexity of the language.

Figure 5.3 shows the Isabelle datatypes that make up the syntax of the language. Note that the left hand side of assignments can be an arbitrary arithmetic expression. For simplicity, we identify values and pointers in this language and admit arbitrary HOL functions for comparison and arithmetic expressions. The program states are the same states that the separation assertions work on. To keep the number of statements small, we only provide the more complicated case of virtual access: even low-level page table manipulations are translated.

The semantics of boolean and arithmetic expressions is shown in Figure 5.4. We write $[\![B]\!]_b$ for the meaning of boolean expression `B` as a partial function from program state to `bool`. Analogously $[\![A]\!]$ is a partial function from state to values. All of these are straightforward, only heap lookup deserves more attention.

Heap lookup only succeeds if the address the argument of the lookup evaluates to virtually maps to a value. This means that the appropriate slices that are involved in the page table lookup as well as the full cell of the target in physical memory must be available in the domain of the heap. In any execution, there will always be full memory cells available, but using our assertions from above we are able to make finer-grained distinctions during program proofs. The function

```
as_view :: fheap × pptr ⇒ vptr ⇀ 32 word
as_view (fh, r) vp ≡ (let hp = h_view fh vp in ptable_lift hp r ▷ hp) vp
```

uses h_view to read the value from the heap after the address has been translated, and

```
_ ▷ _ :: ('a ⇀ 'b) ⇒ ('b ⇀ 'c) ⇒ 'a ⇀ 'c
f ▷ g ≡ λx. case f x of None ⇒ None | Some y ⇒ g y
```

to compose the two partial functions.

Finally, Figure 5.5 shows a big-step operational semantics of commands in the language. We write $\langle c, s\rangle \to s'$ if command c, started in s, evaluates to s'. As usual, the semantics is the smallest relation satisfying the rules of Figure 5.5. Non-

```
⟦Const c⟧ s          = Some c

⟦HeapLookup vp⟧ s = case ⟦vp⟧ s of None ⇒ None
                      | Some v ⇒
                          if (VPtr v ↪ -) s then as_view s (VPtr v)
                          else None
⟦BinOp f e1 e2⟧ s = case (⟦e1⟧ s, ⟦e2⟧ s) of
                      (⌊v1⌋, ⌊v2⌋) ⇒ ⌊f v1 v2⌋ | _ ⇒ None
⟦UnOp f e⟧ s      = case ⟦e⟧ s of None ⇒ None | Some v ⇒ Some (f v)

⟦BConst b⟧b s        = Some b

⟦BComp f e1 e2⟧b s = case (⟦e1⟧ s, ⟦e2⟧ s) of
                       (⌊v1⌋, ⌊v2⌋) ⇒ ⌊f v1 v2⌋ | _ ⇒ None

⟦BBinOp f e1 e2⟧b s = case (⟦e1⟧b s, ⟦e2⟧b s) of
                        (⌊v1⌋, ⌊v2⌋) ⇒ ⌊f v1 v2⌋ | _ ⇒ None

⟦BNot b⟧b s          = case ⟦b⟧b s of None ⇒ None | Some v ⇒ Some (¬ v)
```

Figure 5.4: *Semantics of arithmetic and boolean expressions.*

termination is modelled by absence of the transition from the relation. In contrast to this, we model memory access failure explicitly by writing $\langle c, s\rangle \to$ None, and we do not allow accessing unmapped pages. On hardware, this would lead to a page fault and the execution of a page fault handler. This can also be modelled by making the assignment statement a conditional jump to the page fault handler, or with an abstraction layer showing that the page fault handler always establishes a known good state by mapping pages in from disk. We follow the lead of the seL4 microkernel [16, 17] verification project [31] in wanting to show absence of page faults in the kernel itself, hence our stricter model. Excluding the conditional jump is no loss of generality as we already have IF statements in the language.

As with arithmetic expressions, the most interesting rule in Figure 5.5 is assignment which involves heap access and which we will explain below. In the other rules, we abbreviate $⟦b⟧_b\ s = ⌊True⌋$ with $<b>\ s$, and $⟦b⟧_b\ s = ⌊False⌋$ with $¬<b>\ s$. Failure in any part of the execution leads to failure of the whole statement.

The assignment rule in the top right corner of Figure 5.5 requires that both the arithmetic expressions for the left and right hand side evaluate without failure. The left hand side is taken as a virtual pointer, the right hand side as the value being assigned. The assignment succeeds if the virtual address vp is mapped and allocated. We use the notation $s\ [vp, - \mapsto v]$ to update with v all slices in the heap belonging to the physical address that vp resolves to. Since heap writes always update such full cells, all heaps in executions will only ever consist of full cells and are thus consistent with the usual, non-sliced view of memory. Slices are a tool for assertions and proofs only.

Having shown the semantics, we can now proceed to defining Hoare triples. Validity is the usual:

$$\{\!|P|\!\}\ \ c\ \ \{\!|Q|\!\} \equiv \forall s\ s'.\ \langle c, s\rangle \to s' \wedge P\ s \longrightarrow (\exists r.\ s' = \text{Some}\ r \wedge Q\ r)$$

We do not define a separate syntactic Hoare calculus. Instead, we define validity only and then derive the Hoare rules as theorems in Isabelle/HOL directly. Figure 5.6 shows the rules we have proved for this language. Again, the rules for IF, WHILE,

$$\langle \mathsf{SKIP}, s\rangle \to \mathsf{Some}\ s$$

$$\frac{[\![lval]\!]\ s = \mathsf{Some}\ vp \qquad [\![rval]\!]\ s = \mathsf{Some}\ v \qquad (\mathsf{VPtr}\ vp \hookrightarrow -)\ s}{\langle lval\ :=\ rval\ , s\rangle \to \mathsf{Some}\ (s\ [\mathsf{VPtr}\ vp, - \mapsto_v v])}$$

$$\frac{[\![lval]\!]\ s = \mathsf{None} \vee [\![rval]\!]\ s = \mathsf{None}}{\langle lval\ :=\ rval\ , s\rangle \to \mathsf{None}} \qquad \frac{[\![lval]\!]\ s = \mathsf{Some}\ vp \qquad \neg\ (\mathsf{VPtr}\ vp \hookrightarrow -)\ s}{\langle lval\ :=\ rval\ , s\rangle \to \mathsf{None}}$$

$$\frac{\langle c_0, s\rangle \to \mathsf{Some}\ s'' \qquad \langle c_1, s''\rangle \to s'}{\langle c_0\ ;\ c_1, s\rangle \to s'} \qquad \frac{\langle c_0, s\rangle \to \mathsf{None}}{\langle c_0\ ;\ c_1, s\rangle \to \mathsf{None}}$$

$$\frac{<b>\ s \qquad \langle c_0, s\rangle \to s'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1, s\rangle \to s'} \qquad \frac{\neg <b>\ s \qquad \langle c_1, s\rangle \to s'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1, s\rangle \to s'}$$

$$\frac{[\![b]\!]_b\ s = \mathsf{None}}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_0\ \mathsf{ELSE}\ c_1, s\rangle \to \mathsf{None}} \qquad \frac{\neg <b>\ s}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \to \mathsf{Some}\ s}$$

$$\frac{<b>\ s \qquad \langle c, s\rangle \to \mathsf{Some}\ s'' \qquad \langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s''\rangle \to s'}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \to s'}$$

$$\frac{<b>\ s \qquad \langle c, s\rangle \to \mathsf{None}}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \to \mathsf{None}} \qquad \frac{[\![b]\!]_b\ s = \mathsf{None}}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c, s\rangle \to \mathsf{None}}$$

where

$$(h,\ r)\ [vp, - \mapsto_v v] \equiv \mathsf{case}\ \mathsf{vmap\_view}\ (h,\ r)\ vp\ \mathsf{of}\ \mathsf{Some}\ p \Rightarrow (h\ [p, - \mapsto v],\ r)$$
$$\mathsf{vmap\_view}\ (h,\ r)\ vp \equiv \mathsf{ptable\_lift}\ (\mathsf{h\_view}\ h\ vp)\ r\ vp$$
$$h\ [p, - \mapsto v] \equiv \lambda p'.\ \mathsf{if}\ \mathsf{fst}\ p' = p\ \mathsf{then}\ \mathsf{Some}\ v\ \mathsf{else}\ h\ p'$$

Figure 5.5: *Big-step semantics of commands.*

etc., are straightforward and the same as in a standard Hoare calculus. We write $P\ \lfloor\longrightarrow\rfloor\ Q \equiv \forall s.\ P\ s \longrightarrow Q\ s$ for lifted implication, and «b» s to denote that $[\![b]\!]_b\ s \neq None$. The precondition P in the IF and WHILE case must be strong enough to guarantee failure free evaluation of the condition b. The lifting rules for conjunction and disjunction, as well as the weakening rule are easy to prove, requiring only the definition of validity and separating conjunction. The interesting cases are the assignment rule and the frame rule.

The assignment rule is more complex looking than the standard rule of Separation Logic. Here, both the left and right hand side of the assignment are arbitrary expressions, potentially including heap lookups that need to be evaluated first. This is the reason for the additional P conjunct separate from the basic pointer mapping. It is not an artifact of virtual memory, but one of expression evaluation only. In essence, this is a rule schema. P can be picked to be just strong enough for the evaluation of the expressions to succeed. For instance, if the left hand side were to contain one heap lookup for location x only, we would choose P to be of the form $x \mapsto r$. The rule is sound for too strong and too weak P: either the precondition is merely stronger than it needs to be, or P is too weak to support the expression evaluation conjunct and the precondition as a whole becomes false.

The proof of the assignment rule proceeds by unfolding the definitions and observing that the postcondition is established by the memory update, noting the fact that the ptable_trace from the precondition fully accounts for all page table entries

$$\{\!|P|\!\}\quad \mathsf{SKIP}\quad \{\!|P|\!\} \qquad \frac{\{\!|P|\!\}\quad c\quad \{\!|Q|\!\} \qquad P' \lfloor\longrightarrow\rfloor P \qquad Q \lfloor\longrightarrow\rfloor Q'}{\{\!|P'|\!\}\quad c\quad \{\!|Q'|\!\}}$$

$$\frac{\{\!|P \lfloor\wedge\rfloor \langle b\rangle|\!\}\quad c_1\quad \{\!|Q|\!\} \qquad \{\!|P \lfloor\wedge\rfloor \neg\langle b\rangle|\!\}\quad c_2\quad \{\!|Q|\!\}}{\{\!|P \lfloor\wedge\rfloor \langle\!\langle b\rangle\!\rangle|\!\}\quad \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2\quad \{\!|Q|\!\}}$$

$$\frac{\{\!|P \lfloor\wedge\rfloor \langle b\rangle|\!\}\quad c\quad \{\!|P|\!\} \qquad P \lfloor\longrightarrow\rfloor \langle\!\langle b\rangle\!\rangle}{\{\!|P|\!\}\quad \mathsf{WHILE}\ b\ \mathsf{DO}\ c\quad \{\!|P \lfloor\wedge\rfloor \neg\langle b\rangle|\!\}} \qquad \frac{\{\!|P|\!\}\quad c_1\quad \{\!|Q|\!\} \qquad \{\!|Q|\!\}\quad c_2\quad \{\!|R|\!\}}{\{\!|P|\!\}\quad c_1\ ;\ c_2\quad \{\!|R|\!\}}$$

$$\frac{\{\!|P|\!\}\quad c\quad \{\!|Q|\!\} \qquad \{\!|R|\!\}\quad c\quad \{\!|S|\!\}}{\{\!|P \lfloor\wedge\rfloor R|\!\}\quad c\quad \{\!|Q \lfloor\wedge\rfloor S|\!\}} \qquad \frac{\{\!|P|\!\}\quad c\quad \{\!|Q|\!\} \qquad \{\!|R|\!\}\quad c\quad \{\!|S|\!\}}{\{\!|P \lfloor\vee\rfloor R|\!\}\quad c\quad \{\!|Q \lfloor\vee\rfloor S|\!\}}$$

$$\frac{\{\!|(\mathsf{VPtr}\ vp \mapsto -) \wedge^* P) \lfloor\wedge\rfloor [\![l]\!] = \lfloor vp\rfloor \lfloor\wedge\rfloor [\![r]\!] = \lfloor v\rfloor|\!\}}{l := r \qquad \{\!|\mathsf{VPtr}\ vp \mapsto v) \wedge^* P|\!\}}$$

$$\frac{\{\!|P|\!\}\quad c\quad \{\!|Q|\!\}}{\{\!|P \wedge^* R|\!\}\quad c\quad \{\!|Q \wedge^* R|\!\}}$$

Figure 5.6: *The proof rules for Mapped Separation Logic.*

that are relevant in the postcondition and that P is preserved, because it is separate from the heap in which the update occurs. Since the physical address of the write is separate from the page table lookup for this address and from P, the translation layer for their heaps is not affected by the write. To reason about page table updates we need a slightly stronger rule that unfolds the virtual-to-value mapping and lets us talk about the physical address p:

$$\frac{\{\!|(\mathsf{VPtr}\ vp \mapsto_v p \wedge^* p \to_p - \wedge^* P) \lfloor\wedge\rfloor [\![l]\!] = \lfloor vp\rfloor \lfloor\wedge\rfloor [\![r]\!] = \lfloor v\rfloor|\!\}}{l := r \qquad \{\!|\mathsf{VPtr}\ vp \mapsto_v p \wedge^* p \mapsto_p v \wedge^* P|\!\}}$$

Reasoning with the new mapping predicates is similar to abstract-predicate style reasoning [46] if we never unfold their definitions in client proofs. As we will see in the case study, we only need to do this locally if we are reasoning about changes to the page table and we are interested in the page that is being modified. This obviously heavily depends on the page table encoding. Application level reasoning can proceed fully abstractly.

The second interesting proof rule is the frame rule that allows global reasoning based on local proofs. In many ways it can be seen as the core of Separation Logic. Calcagno et al. [13] provide an abstract framework for identifying a logic as Separation Logic and distill out the central property of locality. In their setting, our separation algebra is the common heap monoid where the binary operation is heap merge lifted to the $heap \times pptr$ type. Our definition of separating conjunction then coincides with the one in the framework, and to show that our logic is a Separation Logic, we only need to show that all actions in the programming language are local. Locality is equivalent to the combination of safety monotonicity and the frame property [13]. In our setting, these two are:

Theorem 5.3.1 (Safety Monotonicity) *If a small state provides enough resources to run a command c, then so does a larger state. Formally, the converse is easier to state:*

$$\frac{\text{ptable\_lift}\ (h_0\ +\!+\ h_1)\ r\ vp = \text{Some}\ p \qquad h_0 \perp h_1}{\text{ptable\_lift}\ h_0\ r\ vp = \text{Some}\ p \vee \text{ptable\_lift}\ h_0\ r\ vp = \text{None}}$$

$$\frac{\text{ptable\_lift}\ h_0\ r\ vp = \text{Some}\ p \qquad h_0 \perp h_1}{\text{ptable\_lift}\ (h_0\ +\!+\ h_1)\ r\ vp = \text{Some}\ p}$$

$$\frac{\begin{array}{c}\text{get\_page}\ vp = \text{get\_page}\ vp' \\ \text{ptable\_lift}\ h\ r\ vp = \text{Some}\ val \qquad \text{ptable\_lift}\ h'\ r\ vp' = \text{Some}\ val'\end{array}}{\text{ptable\_trace}\ h\ r\ vp = \text{ptable\_trace}\ h'\ r\ vp'}$$

$$\frac{p \notin \text{ptable\_trace}\ h\ r\ vp \qquad \text{ptable\_lift}\ h\ r\ vp = \text{Some}\ p}{\text{ptable\_lift}\ (h(p \mapsto v))\ r\ vp = \text{Some}\ p}$$

$$\frac{p \notin \text{ptable\_trace}\ h\ r\ vp \qquad \text{ptable\_lift}\ h\ r\ vp = \text{Some}\ p}{\text{ptable\_trace}\ (h(p \mapsto v))\ r\ vp = \text{ptable\_trace}\ h\ r\ vp}$$

Figure 5.7: *The page table interface.*

$$\mathit{If}\ \langle c, (h\ +\!+\ h', r)\rangle \to \text{None}\ \mathit{and}\ h \perp h'\ \mathit{then}\ \langle c, (h, r)\rangle \to \text{None}.$$

Theorem 5.3.2 (Frame Monotonicity) *Execution of a command can be traced back to a smaller part of the state as long as the command executes at all on the smaller state.*

$$\mathit{If}\ \neg\ \langle c, (h_0, r)\rangle \to \text{None}\ \mathit{and}\ \langle c, (h_0\ +\!+\ h_1, r)\rangle \to \text{Some}\ (h', r)\ \mathit{and}\ h_0 \perp h_1\ \mathit{then}\ \exists h_0'.\ h' = h_0'\ +\!+\ h_1 \wedge \langle c, (h_0, r)\rangle \to \text{Some}\ (h_0', r).$$

We have not formalised the full abstract, relational treatment of the framework by Calcagno et al., but shown the above two properties and the implied frame rule directly by induction on the evaluation of commands.

Figure 5.7 gives the page table interface constraints that we promised in Section 5.1.3. These rules need to be proved about ptable_lift and ptable_trace for a new page table instantiation in order to use the abstract logic rules presented in Figure 5.6. The first two rules are the frame and monotonicity property on ptable_lift. The third rule states that if the domain of ptable_lift does not change, neither does ptable_trace. The last two rules state that updates to the heap outside the trace affect neither the lifting nor the trace of the page table. We have proved the rules in Figure 5.7 for the one-level page table instantiation in the examples.

5.4 Case Study

In this section, we present a small page allocation and assignment routine one might see in operating system services, mapping a *frame*, the physical equivalent of a page, to some address in virtual memory. The program appears in Figure 5.8 with simplified syntax. Frame availability information is stored in a frame table, which contains one entry per frame in the system, marking it as used or unused. In our program, line 1 attempts to find a free frame's entry in the free frame list. An empty list causes an erroneous return at line 7. Line 3 removes the head of the list. Line 4 calculates the number of the frame from its entry. Upon successfully allocating a

```
1. fte := ft_free_list;
2. IF fte != NULL THEN
3.      ft_free_list := *ft_free_list;
4.      frame := &fte - &frame_table;
5.      *(ptable + (a2p page_addr)) := f2a frame OR valid_pmask;
6.      ret_val := 0
7. ELSE ret_val := -1
```

Figure 5.8: *A simple page table manipulating program.*

$$\{|\text{vars} \wedge^* \text{in\_pt } page\_addr \wedge^* \text{frame\_list } (f \cdot fs) \wedge^* \text{pt\_alloc } page\_addr \wedge^* ret\_val_v \mapsto -|\}$$
$$program\ page\_addr$$
$$\{|\text{vars} \wedge^* \text{in\_pt } page\_addr \wedge^* \text{frame\_list } fs \wedge^* \text{VPtr } f \mapsto - \wedge^* ret\_val_v \mapsto 0 \wedge^* \text{page\_mapped } page\_addr|\}$$

where:

$$\text{vars} \equiv ptable_v \mapsto pt \wedge^* frame_v \mapsto - \wedge^* frame\_table_v \mapsto frame\_table \wedge^* fte_v \mapsto -$$

$$\text{in\_pt} :: vptr \Rightarrow fheap \times pptr \Rightarrow bool$$
$$\text{in\_pt } page\_addr\ (h, r) \equiv (\text{VPtr } pt + \text{a2p } page\_addr \mapsto_v r + \text{a2p } page\_addr)\ (h, r)$$

$$\text{pt\_alloc} :: vptr \Rightarrow fheap \times pptr \Rightarrow bool$$
$$\text{pt\_alloc } page\_addr \equiv \lambda(h, r).\ (r + \text{get\_page } page\_addr \rightarrow_p -)\ (h, r)$$

$$\text{frame\_list} :: 32\ word\ list \Rightarrow fheap \times pptr \Rightarrow bool$$

$$\text{frame\_list } xs \equiv \text{list } xs\ ft\_free\_list_v\ (\text{fte\_property } frame\_table)$$

$$\text{list} :: 32\ word\ list \Rightarrow vptr \Rightarrow (32\ word \Rightarrow fheap \times pptr \Rightarrow bool) \Rightarrow fheap \times pptr \Rightarrow bool$$
$$\text{list } []\ h\ P \equiv h \mapsto 0$$
$$\text{list } (x \cdot xs)\ h\ P \equiv \lambda s.\ x \neq 0 \wedge (h \mapsto x \wedge^* \text{list } xs\ (\text{VPtr } x)\ P \wedge^* P\ x)\ s$$

$$\text{fte\_property} :: 32\ word \Rightarrow 32\ word \Rightarrow fheap \times pptr \Rightarrow bool$$
$$\text{fte\_property } start\ ptr \equiv \text{entire\_frame } (\text{PPtr } (\text{f2a } (ptr - start)))\ \{\rightarrow_p\} -$$

$$\text{entire\_frame} :: pptr \Rightarrow pptr \Rightarrow bool$$
$$\text{entire\_frame } p \equiv \{p..\text{PPtr } (\text{pptr\_val } p + 0xFFF)\}$$

Figure 5.9: *Specification of the program in Figure 5.8.*

frame, line 5 updates the page table with the appropriate entry mapping *page_addr* to the new frame.

The functions a2p and f2a convert addresses to page numbers and frame numbers to addresses by respectively dividing or multiplying by 4096.

We have proved that the program conforms to specification in Figure 5.9.

The language is slightly atypical for a separation-logic based language in that it is purely heap based. In other words, there are no local variables accessible in the language. If we wish to reason about use of a local function variable, we must assert its location and value using a maps-to predicate. Hence, we use $var_v \mapsto var$ to denote that a variable *var* has a specific value and to represent `var` in Figure 5.8. For our case study we assume that the page table is accessible from the virtual address `ptable`, and the frame table lies at `frame_table`. Further, we have a non-empty free list starting at `first_free`, corresponding to *f* in Figure 5.9, where

each address in the list indicates the presence of a frame. Finally, we require that the page table entry that is used to resolve a lookup of *page_addr* is allocated, and accessible from our address space. We additionally require that *page_addr* is aligned to the page size. As a result of executing the program, the free frame list becomes shorter and the page at *page_addr* is fully accessible. Figure 5.10.

```
page_mapped :: vptr ⇒ fheap × pptr ⇒ bool
page_mapped vp ≡
    entire_page vp {↦} − ∧* consume_slices vp (UNIV − entire_page vp)
consume_slice :: vptr ⇒ vptr ⇒ fheap × pptr ⇒ bool
consume_slice vp sl ≡ λ(h, r). dom h = ptable_trace (h_view h sl) r vp × {sl}
consume_slices :: vptr ⇒ (vptr ⇒ bool) ⇒ fheap × pptr ⇒ bool
consume_slices vp S ≡ fold_image op ∧* (consume_slice vp) □ S
entire_page :: vptr ⇒ vptr ⇒ bool
entire_page vp ≡ {vp..VPtr (vptr_val vp + 0xFFF)}
```

Figure 5.10: *The* page_mapped *predicate and associated definitions;* page_mapped *indicates that a page at a given virtual address is fully accessible.*

All our separation logic statements work on heaps of a precise size. Using up the entire page table entry in our precondition means we must likewise use it all in the postcondition. A page table entry maps 4096 addresses, but is made up of more slices. Stating that those addresses are mapped does not consume all the slices. The difference in heap size is made up by consume_slices.

The actual mapping-in step from line 5 of our program performs a write to the page table at *pt* + a2p *page_addr*. In addition to our assignment rule, we have shown another property that allows us to to conclude from the post-state of line 5 that the page at *page_addr* is now completely mapped:

```
entire_frame (PPtr (f2a frame)) {→p} − ∧* pt_map page_addr frame ⌊⟶⌋
page_mapped page_addr
```

where:

```
pt_map :: vptr ⇒ 32 word ⇒ fheap × pptr ⇒ bool
pt_map page_addr frame ≡
λ(h, r). (PPtr (pptr_val r + a2p page_addr) ↦p f2a frame OR 1) (h, r)
```

This rule is the only place in the case study where we had to unfold page table definitions and reason directly about the encoding. All other reasoning used Separation Logic rules only.

In order to check successful interaction with the newly mapped page, we added an extra segment to our program.

```
*page_addr := 0xFF;
*(page_addr + 3) := *page_addr + 2
```

If executed on a state with offsets 0 and 3 in the page mapped and allocated:

$$\{|page\_addr \mapsto - \wedge^* page\_addr + 3 \mapsto -|\}$$

it results in those offsets set to `0xFF` and `0x101`:

$$\{|page\_addr \mapsto 0xFF \wedge^* page\_addr + 3 \mapsto 0x101|\}$$

These two offsets are part of the page at `page_addr`; the program fragment does not change anything else. As we can rewrite

```
page_mapped p = (p ↦ - ∧* p + 3 ↦ - ∧* page_mapped {p, p + 3} p)
```

we can invoke the frame rule and include the rest of the state.

Our case study shows that our logic allows abstract reasoning, even in the presence of page table manipulation. Examples like this would occur when verifying operating system code directly. The logic is easier to use for application code that might support sharing, but has no direct access to the page table.

Our case study also shows that reasoning about changing page table mappings from the address space they define is a complicated process to reason about. Ironically, the primary benefit of working within a separation logic is not visible in the rules of our case study. We have precisely specified which areas of the heap are involved in mapping in a new page. Therefore, by virtue of the frame rule, we can infer the areas of memory which did not change without any need for further complicated reasoning. Without separation logic, even if the mapping lemma itself were easier to state and prove, further reasoning about virtual memory would be necessary to prove that a virtual address is unaffected by mapping in a page. Some of this modularity is apparent in the case study within our full logic in Chapter 6.

5.5 Conclusion

In this chapter, we have presented an extension of Separation Logic which allows reasoning about virtual memory and processes running within. The logic fully supports the frame rule as well as the other Separation Logic proof rules and allows for a convenient representation of predicates on memory at three levels: the virtual map, physical heap and virtual address space. We have presented a small case study that demonstrates the applicability of the logic to operating system level page table code as well as client code using the page table mechanism.

For our initial analysis of the logic we chose a simplified machine and page table instance. The logic does not depend on the implementation of either.

We have managed to fully hide the complexity of virtual memory reasoning for code that does not directly modify the page table. We have also shown that high-level reasoning is still possible for code that does. The concepts in the logic are close to the mental model operating system kernel programmers have of virtual memory.

In the following chapters, we will enhance our model and instantiate it to the C programming language and apply it to both a more realistic machine and a code base inspired from real life use.

Chapter 6

Typed Mapped Separation Logic

In the previous chapter we described a method for integrating a view of virtual memory into separation logic for a very simple architecture with a one-level page table. In this chapter, we present a more realistic framework, built around the 32-bit ARMv6 [6] architecture in little-endian mode, with a two-level page table modeled after the actual hardware, as well as storing typed objects in memory. We connect this model to the C semantics developed by Norrish for the L4.verified project [57]. Our framework is generic where possible.

For our case study in Section 6.8, we will present our analysis of a function in the seL4 microkernel [31] responsible for writes to the second-level page table, thus responsible for mapping in pages. We will formally precisely identify the context in which the function does this, as well as what it means for a page to be "mapped" in our new framework.

We have already published the core ideas of typed mapped separation in *Types, Maps and Separation Logic* [34]. This chapter covers our work in more detail, along with any changes we have implemented since then. The primary change is abolishing tagging of types in memory in favour of dealing purely in separation logic. This is because such annotations are only necessary to support the Burstall/Bornat style of typed heaps. In separation logic, the information is contained in the maps-to predicates, as will become evident in this chapter.

We will begin this chapter with an overview of architecture-based issues, such as pointers, addresses, their manipulation and sequences thereof, and instantiate the relevant structures to the ARMv6 architecture in little-endian mode. We will then proceed to our updated page table interface, which has been augmented slightly from the one in Chapter 5 in order to facilitate page tables with superpages, as well as decoding permissions from the page table entries. We instantiate the interface to the ARMv6 two-level page table. We explain our semantics for storing words, structures and arrays in memory, before moving on to byte-level, type-level and special-purpose maps-to predicates. We then build upon these maps-to predicates in order to construct an interface to Norrish's C to Isabelle/HOL parser [57]. We conclude with a case study based upon a small function from the seL4 [31] microkernel and precisely verifying the conditions under which it maps in a new page.

6.1 Architecture Setup

6.1.1 Pointers and Addresses

Abstract Pointers and Addresses

As in the previous chapter, we still need to talk about two different types of addresses: virtual and physical. In Chapter 5 pointers were addresses, so we could get away with two datatypes with separate constructors. In this chapter we still have addresses, but we also need to represent typed pointers. such as a pointer to a 32-bit word at virtual address 0xF00F. If we have two different address constructors, we will need separate functions for virtual and physical addresses, and furthermore for pointers to such addresses. In order to avoid this duplication, we have only a single address constructor, Addr which takes a tag (*virtual* or *physical*) as a type argument:

```
typedecl physical
typedecl virtual
datatype ('a, 'p) addr_t = Addr 'a
```

Thus it takes two type arguments: the type of addresses themselves (e.g. 32-bit words), and a phantom type tag designating the sort of address it is. For example, a 32-bit virtual address has the type *(32 word, virtual) addr_t*. Where Addr constructs an address, addr_val gets at the value inside it:

```
addr_val :: ('a, 'p) addr_t ⇒ 'a
addr_val (Addr a) = a
```

We can add to and subtract from addresses:

```
a + x ≡ Addr (addr_val a + x)
a - x ≡ Addr (addr_val a - x)
```

and express address ranges:

```
addr_seq :: ('a, 'p) addr_t ⇒ nat ⇒ ('a, 'p) addr_t list
addr_seq p 0 = []
addr_seq p (Suc n) = p · addr_seq (p + 1) n
```

Similarly to addresses, we define pointers as the constructor Ptr applied to an address. We also add a third type variable, a phantom type representing the type of entity that the pointer points to. We use only a phantom type since languages with direct access to memory, such as C, do not have to respect type safety. At the underlying level, it's just bytes:

```
datatype ('a, 'p, 't) ptr_t = Ptr ('a, 'p) addr_t
```

So a pointer to an *int* in 32-bit physical memory has the type *(32 word, physical, int) ptr_t*. The value of a pointer is its address:

```
ptr_val :: ('a, 'p, 't) ptr_t ⇒ ('a, 'p) addr_t
ptr_val (Ptr a) = a
```

For convenience, we also combine ptr_val and addr_val when we need the value of the address contained in a pointer:

```
ptr_addr :: ('a, 'p, 't) ptr_t ⇒ 'a
ptr_addr x ≡ addr_val (ptr_val x)
```

Since the type of the pointer target is just a phantom type, casting is a no-operation:

```
ptr_coerce :: ('a, 'p, 't) ptr_t ⇒ ('a, 'p, 't2) ptr_t
ptr_coerce (Ptr p) = Ptr p
```

Pointers and Addresses on the ARMv6

Our case study is the ARMv6 [6], a 32-bit architecture. This means that the machine word (corresponding to machine register size), as well as both virtual and physical addresses are 32 bits wide, allowing us to define short-hand specific to this machine:

```
types machine_word = 32 word
      vaddr        = (32 word, virtual) addr_t
      paddr        = (32 word, physical) addr_t
```

and similarly for pointers:

```
types 't vptr = (32 word, virtual, 't) ptr_t
      't pptr = (32 word, physical, 't) ptr_t
```

Knowing the sizes of addresses, we can derive the maximum sizes of physical and virtual memory address spaces in a 32-bit system:

```
memory_size :: nat
memory_size = 2^32
addr_space_size :: nat
addr_space_size = 2^32
```

Finally, we need to consider storage of bytes in memory. On a little-endian system, the bytes in a word are stored in *reverse* order. The word_rsplit and word_rcat functions in the Isabelle/HOL word library can split an n-bit word into a list of k-bit words and recombine it again into an n-bit word. The n and k are specified using the type system. Thus, breaking up a word into a byte list and recombining them, on a little endian system can be represented as follows, with rev being list reversal:

```
machine_w2b :: 'a word ⇒ 8 word list
machine_w2b ≡ rev ∘ word_rsplit
machine_b2w :: 8 word list ⇒ 'a word
machine_b2w ≡ word_rcat ∘ rev
```

6.2 Page Tables

6.2.1 A Page Table Abstraction

The page table is at the core of virtual memory, being the physical memory encoding of the virtual memory layout. In Chapter 5, we had an abstract page table interface instantiated to a very simple one-level page table. When working with more complicated page tables on realistic hardware, such as the two-level page table on the ARMv6, we need to augment the abstract interface slightly. We explained the basics of the functioning of the ARMv6 page table in Chapter 3, Section 3.3.

Our abstract interface requires providing three functions:

$$\frac{p \notin \text{ptable\_trace}\ h\ r\ vp \qquad \text{ptable\_lift}\ h\ r\ vp = \lfloor p \rfloor}{\text{ptable\_trace}\ (h(p \mapsto v))\ r\ vp = \text{ptable\_trace}\ h\ r\ vp}$$

$$\frac{p \notin \text{ptable\_trace}\ h\ r\ vp \qquad \text{ptable\_lift}\ h\ r\ vp = \lfloor p \rfloor}{\text{ptable\_lift}\ (h(p \mapsto v))\ r\ vp = \lfloor p \rfloor}$$

$$\frac{\text{ptable\_lift}\ h_0\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\text{ptable\_lift}\ (h_0\ ++\ h_1)\ r\ vp = \lfloor p \rfloor}$$

$$\frac{\text{ptable\_lift}\ h\ r\ vp = \lfloor p \rfloor \qquad h \perp h'}{\text{ptable\_trace}\ (h\ ++\ h')\ r\ vp = \text{ptable\_trace}\ h\ r\ vp}$$

$$\frac{\text{ptable\_lift}\ (h_0\ ++\ h_1)\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\text{ptable\_lift}\ h_0\ r\ vp = \lfloor p \rfloor \lor \text{ptable\_lift}\ h_0\ r\ vp = \text{None}}$$

Figure 6.1: *Abstract page table interface.*

```
ptable_lift  :: ('paddr ⇀ 'val) ⇒ 'base ⇒ 'vaddr ⇀ 'paddr
ptable_trace :: ('paddr ⇀ 'val) ⇒ 'base ⇒ 'vaddr ⇒ 'paddr set
get_page     :: ('paddr ⇀ 'val) ⇒ 'base ⇒ 'vaddr ⇒ 'a
```

The first of these, ptable_lift, represents a standard walk of the page table. Given a *'vaddr* address and a physical memory representation from *'paddr* addresses to some kind of values, perform a lookup and return a *'paddr* address. The *'base* parameter is meant to contain information on where the page table is to be found in memory. It is typically an address in physical memory. As expected from their names, *'vaddr* and *paddr* are meant to be virtual and physical addresses respectively. Naturally, the function is partial, as not all virtual addresses can be resolved to physical ones. There is an additional difference from what one might expect of an interface for page tables: our physical memory representation is a *partial* map. This is because our entire framework is built around the concept of *partial heaps* that underlies the local reasoning of separation logic. Essentially, we expect a lookup of *any* kind outside of the part of the heap we have access to to fail; this includes page table lookups.

For this reason we require ptable_trace, which is to report on the set of all physical addresses that need to be accessed during a ptable_trace. Given this set of addresses for looking up a virtual address, it intuitively follows that modifying any *other* address will have no effect on the lookup. We enforce this in the assumptions of our interface in Figure 6.1. Finally, get_page is a convenience function used to figure out the granularity of the mapping of the block a given virtual address belongs to. For our one-level page table in Chapter 5 this is completely obvious. However for multi-level page tables with superpages such as those on an ARMv6, extra information is necessary, as we will demonstrate shortly.

The properties we require of any page table instantiation are listed in Figure 6.1. They require that ptable_lift and ptable_trace behave like separation logic assertions, which we described in Chapter 4. The first two rules establish the relationship between the trace and result of a page table lookup: given a valid lookup for a virtual address, modifying any physical address outside of the lookup trace has no effect either on the lookup or the trace in the updated heap. Next we require that the lookup and trace

respect the separation logic safety monotonicity property [13]: if a lookup succeeds for a virtual address in a small heap, then it will also succeed in a larger heap. Our final requirement is that page table lookups have safety monotonicity property of separation logic [13]. Reducing the size of the heap in which a lookup succeeds can cause the lookup to fail, but it can never cause the lookup to successfully resolve to a *different* address. We do not require the safety monotonicity property on the lookup trace, however. This is because it is still useful to talk about a failed lookup in which n levels of an m-level page table were successfully traversed; rather than all-or-nothing reasoning, we can build up a successful lookup from a failed one with a partial trace of how far it got.

6.2.2 ARMv6 Page Table Formalisation

In Chapter 3, Section 3.3, we described the address translation mechanism on the ARMv6 architecture. Now, as we said earlier, we will present our formalisation of it. We will later use this formalisation to instantiate our logic to the ARMv6. We do not model the entirety of the ARMv6 address translation mechanism. We have reduced the scope of what we model in order to focus on the implications of address translation in a two-level page table with superpages. Nonetheless, the subset we model is a real-world example, as this is the setup used in the seL4 microkernel [31] which was verified in the L4.verified project.

We make the following assumptions in our formalisation about the configuration of the translation unit. The ARM reference manual [6] contains the exact meaning of these settings:

- The translation mechanism is set to use the ARMv6 page table format with subpages disabled, i.e. the subpage AP bits are disabled.
- Extended physical addresses, i.e. a 40-bit physical address space, are disabled. The physical address space is 32-bit.
- We do not consider domains outside of domain 0. We assume the domain is always zero.

Page Sizes

Our model supports all non-legacy ARMv6 page table sizes. That is small (4Kb) and large (64Kb) pages, as well as sections (1Mb) and supersections (16Mb):

```
datatype page_type =
      ArmSmallPage
    | ArmLargePage
    | ArmSection
    | ArmSuperSection
```

The amount of bits that can be used to index into them is 12, 16, 20 and 24 respectively:

```
page_bits :: page_type ⇒ nat
page_bits ArmSmallPage    ≡ 12
page_bits ArmLargePage    ≡ 16
page_bits ArmSection      ≡ 20
page_bits ArmSuperSection ≡ 24
```

which is sufficient to define their size in bytes:

```
page_size :: page_type ⇒ nat
page_size page_type ≡ 2^page_bits page_type
```

We also need to be able to split a 32-bit virtual address into the base address of the page it is in and the offset within that page by respectively either masking the low bits or the high bits:

```
vaddr_offset :: page_type ⇒ 32 word ⇒ 32 word
vaddr_offset p w  ≡  w AND mask (page_bits p)
addr_base :: page_type ⇒ 32 word ⇒ 32 word
addr_base sz w  ≡  w AND NOT mask (page_bits sz)
```

Furthermore, if an address' offset inside a page is zero, we consider the address to be *page aligned*:

```
page_aligned :: page_type ⇒ 32 word ⇒ bool
page_aligned page_type p = (vaddr_offset page_type p = 0)
```

Permissions

All mappings of pages also contain architecture-specific flags which indicate permissions and other properties of the particular mapping. Our formalisation extracts them into a `arm_perm_bits` record where each named field is one of the properties:

```
record arm_perm_bits =
    arm_p_APX  :: 1 word
    arm_p_AP   :: 2 word
    arm_p_TEX  :: 3 word
    arm_p_S    :: 1 word
    arm_p_XN   :: 1 word
    arm_p_C    :: 1 word
    arm_p_B    :: 1 word
    arm_p_nG   :: 1 word
```

According to the ARM reference manual [6], on ARMv6 these flags correspond to access permissions (AP), extended access permission bit (APX), type extension (TEX), as well as the shared (S), execute-never (XN), cacheable (C), bufferable (B), not-global (nG) bits. We do not provide a higher-level abstraction here, such as a set of properties, as these do not follow in a straightforward manner from the permission bits. For example, setting APX to 0 and AP to "read-only" results in read-only mode for user-level code, but read/write mode for privileged level code. If we change APX to 1, then the permissions switch to being read-only for privileged level code and inaccessible to user-level code.

The permission bits are arranged differently for section/supersection mappings, large page and small page mappings. For valid entries, we use the following functions to extract them, following the ARMv6 standard exactly:

```
perm_bits_pde_sections :: 32 word ⇒ arm_perm_bits
perm_bits_pde_sections w ≡ (|arm_p_APX = ucast ((w >> 15) AND 1),
                             arm_p_AP = ucast ((w >> 10) AND 3),
                             arm_p_TEX = ucast ((w >> 12) AND 7),
                             arm_p_S = ucast ((w >> 16) AND 1),
                             arm_p_XN = ucast ((w >> 4) AND 1),
                             arm_p_C = ucast ((w >> 3) AND 1),
                             arm_p_B = ucast ((w >> 2) AND 1),
                             arm_p_nG = ucast ((w >> 17) AND 1)|)

perm_bits_pte_small :: 32 word ⇒ arm_perm_bits
perm_bits_pte_small w ≡ (|arm_p_APX = ucast ((w >> 9) AND 1),
                           arm_p_AP = ucast ((w >> 4) AND 3),
                           arm_p_TEX = ucast ((w >> 6) AND 7),
                           arm_p_S = ucast ((w >> 10) AND 1),
                           arm_p_XN = ucast (w AND 1),
                           arm_p_C = ucast ((w >> 3) AND 1),
                           arm_p_B = ucast ((w >> 2) AND 1),
                           arm_p_nG = ucast ((w >> 11) AND 1)|)

perm_bits_pte_large :: 32 word ⇒ arm_perm_bits
perm_bits_pte_large w ≡ (|arm_p_APX = ucast ((w >> 9) AND 1),
                           arm_p_AP = ucast ((w >> 4) AND 3),
                           arm_p_TEX = ucast ((w >> 12) AND 7),
                           arm_p_S = ucast ((w >> 10) AND 1),
                           arm_p_XN = ucast ((w >> 15) AND 1),
                           arm_p_C = ucast ((w >> 3) AND 1),
                           arm_p_B = ucast ((w >> 2) AND 1),
                           arm_p_nG = ucast ((w >> 11) AND 1)|)
```

ucast is an unsigned cast between two words. In this case, it performs truncation from a 32-bit machine word to the target size of the field.

PDEs and PTEs

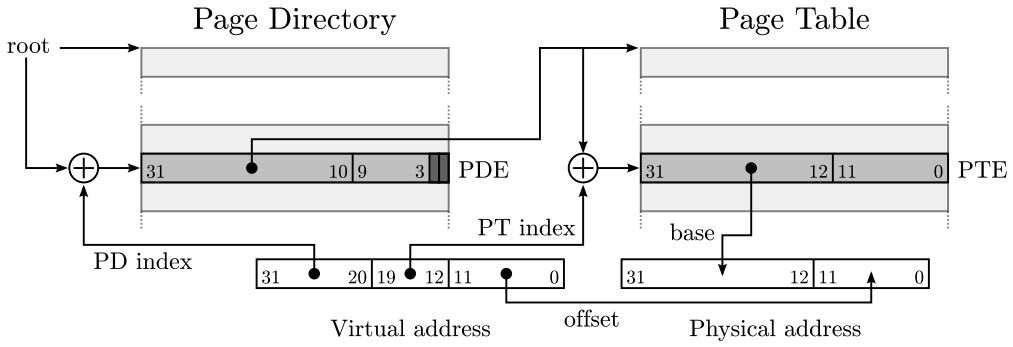


Figure 6.2: *Page table lookup for a 4KB small page on ARMv6*

Now we move on to resolving virtual addresses to physical ones. For convenience, we will repeat the diagrams from Chapter 3 in this one as Figure 6.2 and

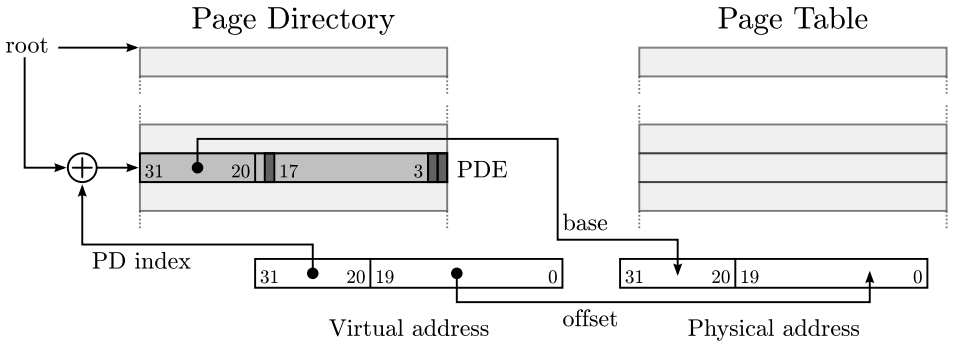


Figure 6.3: *Page table lookup for a 1MB section on ARMv6*

Figure 6.3. As we described there, the ARMv6 uses a two-level page table structure: the first level contains page directory entries (PDEs) and the second level page table entries (PTEs). A PDE may either be invalid, reserved for future processor functionality, resolve an address to a section or supersection, or point to a PTE:

```
datatype pde = InvalidPDE | ReservedPDE
  | PageTablePDE paddr
  | SectionPDE paddr arm_perm_bits
  | SuperSectionPDE paddr arm_perm_bits
```

Similarly, a PTE may be invalid, reserved or resolve to a large or small page:

```
datatype pte = InvalidPTE
  | LargePagePTE paddr arm_perm_bits
  | SmallPagePTE paddr arm_perm_bits
```

To get a datatype we can reason about out of a 32-bit word in memory, we need to perform some decoding. This consists of following the ARMv6 manual [6] and picking the right bits. For PDEs, we do it as follows:

```
decode_pde_section :: 32 word ⇒ pde
decode_pde_section w ≡ SectionPDE (Addr (addr_base ArmSection w))
                         (perm_bits_pde_sections w)
decode_pde_ssection :: 32 word ⇒ pde
decode_pde_ssection w ≡ SuperSectionPDE (Addr (addr_base ArmSuperSection w))
                         (perm_bits_pde_sections w)
decode_pde_pt :: 32 word ⇒ pde
decode_pde_pt w ≡ PageTablePDE (Addr (w AND NOT mask 9))
decode_pde :: 32 word ⇒ pde
decode_pde w ≡
  let pde_type = w AND 3
  in if pde_type = 1 then decode_pde_pt w
     else if pde_type = 2
          then if w !! 18 then decode_pde_ssection w else decode_pde_section w
          else if pde_type = 3 then ReservedPDE else InvalidPDE
decode_heap_pde :: (paddr ⇀ 8 word) ⇒ paddr ⇀ pde
decode_heap_pde h p ≡ Option.map decode_pde (load_machine_word h p)
```

We will introduce load_machine_word later, in Section 6.3. At this point, it is sufficient to know it loads a 32-bit address from the physical address space by invoking

machine_b2w on the four consecutive addresses from the supplied location. If the partial heap does not contain those addresses, it will return None.

We perform a similar decoding for PTEs:

```
decode_pte_small :: 32 word ⇒ pte
decode_pte_small w     ≡   SmallPagePTE (Addr (addr_base ArmSmallPage w))
                             (perm_bits_pte_small w)
decode_pte_large :: 32 word ⇒ pte
decode_pte_large w     ≡   LargePagePTE (Addr (addr_base ArmLargePage w))
                             (perm_bits_pte_large w)
decode_pte :: 32 word ⇒ pte
decode_pte w ≡
    if w AND 3 = 0 then InvalidPTE
    else if w !! 1 then decode_pte_small w else decode_pte_large w
decode_heap_pte :: (paddr ⇀ 8 word) ⇒ paddr ⇀ pte
decode_heap_pte h p   ≡   Option.map decode_pte (load_machine_word h p)
```

Performing a Page Table Lookup

Given the decoding functions we introduced, we create lookup functions which, given the address of a PTE or PDE can give us back the size of the page a virtual address is in, as well as the physical base address that page is mapped to. On the ARMv6, bits [31:20] of a virtual address represent an index into the page directory, while bits [19:12] are used as an index into the page table. Each PDE and PTE entry is 32-bits, i.e. 4 bytes, thus we multiply those indices by 4, look up in the PDE/PTE and decode:

```
vaddr_pd_index :: 32 word ⇒ 32 word
vaddr_pd_index w   ≡   (w >> 20) AND mask 12
vaddr_pt_index :: 32 word ⇒ 32 word
vaddr_pt_index w   ≡   (w >> 12) AND mask 8
lookup_pde :: (paddr ⇀ 8 word)
           ⇒ paddr ⇒ vaddr ⇀ paddr × page_type × arm_perm_bits
lookup_pde h root vp ≡
    case decode_heap_pde h (root + (vaddr_pd_index (addr_val vp) << 2)) of
    None ⇒ None | ⌊PageTablePDE pt_base⌋ ⇒ lookup_pte h pt_base vp
    | ⌊SectionPDE base perms⌋ ⇒ ⌊(base, ArmSection, perms)⌋
    | ⌊SuperSectionPDE base perms⌋ ⇒ ⌊(base, ArmSuperSection, perms)⌋
    | ⌊_⌋ ⇒ None
lookup_pte :: (paddr ⇀ 8 word)
           ⇒ paddr ⇒ vaddr ⇀ paddr × page_type × arm_perm_bits
lookup_pte h pt_base vp ≡
    case decode_heap_pte h
         (pt_base + (vaddr_pt_index (addr_val vp) << 2)) of
    None ⇒ None | ⌊InvalidPTE⌋ ⇒ None
    | ⌊LargePagePTE base perms⌋ ⇒ ⌊(base, ArmLargePage, perms)⌋
    | ⌊SmallPagePTE base perms⌋ ⇒ ⌊(base, ArmSmallPage, perms)⌋
```

With lookup_pde, we can figure out which page a virtual address belongs to by masking out the offset bits for the page size it resolves to:

```
get_page :: (paddr ⇀ 8 word)
          ⇒ paddr ⇒ vaddr ⇀ vaddr × page_type × arm_perm_bits
get_page h root vp ≡
    Option.map
     (λ(base, pg_size, perms).
          (Addr (addr_base pg_size (addr_val vp)), pg_size, perms))
     (lookup_pde h root vp)
```

We can use the page lookup to transform a virtual pointer by masking the offset while replacing the virtual page address with a physical frame address, resulting in Definition 6.2.1.

Definition 6.2.1 *The ARMv6 version of* ptable_lift.

Given a physical memory heap and a physical address denoting where the page directory of a page table resides, try to resolve a virtual address to a physical one.

```
ptable_lift :: (paddr ⇀ 8 word) ⇒ paddr ⇒ vmap
ptable_lift h pt_root vp ≡
    let vp_val = addr_val vp
    in Option.map (λ(base, pg_size, perms). base + vaddr_offset pg_size
    vp_val)
        (lookup_pde h pt_root vp)
```

Finally, we implement the page table lookup trace in Definition 6.2.2 by following the same procedure as ptable_lift, but keeping track of memory areas touched rather than the lookup result.

Definition 6.2.2 *The ARMv6 version of* ptable_trace.

Given a physical memory heap and a physical address denoting where the page directory of a page table resides, return all physical addresses that influence what a page table lookup for a virtual address resolves to.

```
ptable_trace :: (paddr ⇀ 8 word) ⇒ paddr ⇒ vaddr ⇒ paddr ⇒ bool
ptable_trace h root vp ≡
    let vp_val = addr_val vp; pd_idx_offset = vaddr_pd_index vp_val << 2;
       pt_idx_offset = vaddr_pt_index vp_val << 2;
       pd_touched = set (addr_seq (root + pd_idx_offset) 4);
       pt_touched = λpt_base. set (addr_seq (pt_base + pt_idx_offset) 4)
    in case decode_heap_pde h (root + pd_idx_offset) of None ⇒ ∅
       | ⌊PageTablePDE pt_base⌋ ⇒ pd_touched ∪ pt_touched pt_base
       | ⌊_⌋ ⇒ pd_touched
```

We have also proven that these definitions fulfill the requirements for our abstract page table interface described in Section 6.2.1, Figure 6.1. As we showed in Chapter 5, these allow us to include the page table lookup mechanisms in separation logic.

6.3 Storage of Values in Memory

In order to talk about *typed* mapped separation logic, we need a model for storing values of different types (such as a C *int* or *struct*) in memory. In other words, when we say "physical pointer p points to a *struct s*", we must have an understanding of what it means for something to be stored in memory which is, after all, a sequence of byte-sized cells.

This section describes a type class of memory-storable types, the instantiation of this class to 8, 16, and 32-bit words, as well as how we consider multi-field structures without padding as storable.

$$0 < \text{size\_of } TYPE('a) \qquad \text{size\_of } TYPE('a) < \min \text{ memory\_size addr\_space\_size}$$

$$\text{align\_of } TYPE('a)\ dvd \text{ memory\_size} \wedge \text{align\_of } TYPE('a)\ dvd \text{ addr\_space\_size}$$

$$\text{from\_bytes } (\text{to\_bytes } v) = v \qquad \text{length } (\text{to\_bytes } v) = \text{size\_of } TYPE('a)$$

$$\frac{\text{length } bs = \text{size\_of } TYPE('a)}{\text{to\_bytes } (\text{from\_bytes } bs) = bs}$$

Figure 6.4: *Memory-storable type class* `mem_type` *axioms for a type* `'a`*.*

6.3.1 The Class of Memory-Storable Types

In our work, we consider *non-padded* objects in memory. That is, objects that are serialisable to and from a stream of bytes which happens to be their representation in memory. We do not support gaps in this stream. The work of Tuch [54] demonstrates a method for encoding padding. In the interest of clarity, we omit consideration of padding. This decision is hardly crippling: in our chosen arena of operating system kernels, structures are usually hand-packed anyway to avoid padding, since presence of padding means wasted memory. Our memory-storable class is inspired by the one in the work of Tuch et al. [57].

Our name for the memory-storable class is `mem_type`, thus when we will refer to a memory-storable type `'t`, we will implicitly mean `'t::mem_type`.

We require the following interface from a memory-storable type:

```
size_of     :: 'a itself ⇒ nat
align_of    :: 'a itself ⇒ nat
to_bytes    :: 'a ⇒ 8 word list
from_bytes  :: 8 word list ⇒ 'a
```

As expected, size_of is the size in bytes of any object of this type in memory, while align_of is the required alignment in bytes. The serialisation to a stream of bytes in memory is performed by to_bytes, while from_bytes does the inverse.

Figure 6.4 shows the axioms of the memory-storable class of types. The object's serial representation's length must be the same as its size_of, a size which must fit both in physical memory and virtual memory address space. Similarly, the object can only request an alignment which divides both physical and virtual memory address space sizes. In the binary-based system world those sizes will always be powers of two. The consequence of our requirement is that alignments must also be powers of two. Finally, from_bytes must be an inverse of to_bytes, and vice-versa. The last rule in Figure 6.4 allows us to define a generic decoder to pop off a sequence of memory-storable objects from a byte stream:

```
mem_type_decode :: 8 word list ⇒ 'a × 8 word list
mem_type_decode bs  ≡  let sz = size_of TYPE('a)
                       in (from_bytes (take sz bs), drop sz bs)
```

Pointers to Memory-Storable Types

In the C programming language, adding n to a pointer does not add n to its address, but instead increments the address by the size of the pointed to type n times. Since we will be interfacing to the C programming language, and we know the in-memory sizes of memory-storable types, we implement a similar functionality, shown in Figure 6.5. We use of_nat to create a word out of a natural number.

```
ptr_raw_add :: ('a, 'p, 't) ptr_t ⇒ 'a ⇒ ('a, 'p, 't) ptr_t
ptr_raw_add p x ≡ Ptr (ptr_val p + x)

_ + _ :: ('a, 'p, 't) ptr_t ⇒ 'a ⇒ ('a, 'p, 't) ptr_t
p + x ≡ ptr_raw_add p (x * of_nat (size_of TYPE('t)))

ptr_raw_sub :: ('a, 'p, 't) ptr_t ⇒ 'a ⇒ ('a, 'p, 't) ptr_t
ptr_raw_sub p x ≡ Ptr (ptr_val p - x)

_ - _ :: ('a, 'p, 't) ptr_t ⇒ 'a ⇒ ('a, 'p, 't) ptr_t
p - x ≡ ptr_raw_sub p (x * of_nat (size_of TYPE('t)))

ptr_seq :: ('a, 'p, 't) ptr_t ⇒ nat ⇒ ('a, 'p, 't) ptr_t list
ptr_seq p 0 = []
ptr_seq p (Suc n) = p · ptr_seq (p + 1) n
```

Figure 6.5: *Addition and subtraction of offsets to pointers. "Raw" manipulation has byte granularity. Manipulation of pointers to type 't, where 't is memory-storable has* size_of `TYPE('t)`. *Sequences of pointers.*

A pointer is considered aligned if the required alignment of the type it points to divides its address. We use unat to perform an unsigned conversion from a word to a natural number:

```
ptr_aligned :: ('a word, 'p, 't) ptr_t ⇒ bool
ptr_aligned p ≡ align_of TYPE('t) dvd unat (ptr_addr p)
```

6.3.2 Loading Memory-Storable Types

Given a class of serialisable types, we can concoct a system for storing them in memory. In our work, we have two different types of memory: virtual and physical. The top-level abstractions of them, however, are still those of a partial heap: a partial map from addresses to values, given some kind of addresses.

If we try load a number of values sequentially beginning from some address, we will get a list of load results, each of whose success depends on whether the address is in the partial heap or not:

```
load_list_basic :: (('a, 'p) addr_t ⇀ 'v) ⇒ nat ⇒ ('a, 'p) addr_t
                ⇒ 'v option list
load_list_basic h 0 p ≡ []
load_list_basic h (Suc n) p ≡ h p · load_list_basic h n (p + 1)
```

We consolidate this result into `Some` list of actual values, or None if any of them failed:

```
deoption_list :: 'a option list ⇀ 'a list
deoption_list xs ≡ if None ∈ set xs then None else ⌊map the xs⌋
load_list :: (('a, 'p) addr_t ⇀ 'v) ⇒ nat ⇒ ('a, 'p) addr_t ⇀ 'v list
load_list h n p ≡ deoption_list (load_list_basic h n p)
```

All that remains is to plug our list of bytes into from_bytes, which we explained in Section 6.3.1, and tell the type system which memory-storable type `'t` we want to get out:

```
load_value :: (('a, 'p) addr_t ⇀ 8 word) ⇒ ('a, 'p) addr_t ⇀ 't
load_value h p ≡ Option.map from_bytes (load_list h (size_of TYPE('t)) p)
```

For example, the load_machine_word we promised to explain in Section 6.2.2 is just load_value constrained to return a 32-bit word, which is the machine word size on ARMv6:

```
load_machine_word :: (('a, 'p) addr_t ⇀ 8 word) ⇒ ('a, 'p) addr_t
                     ⇀ 32 word
load_machine_word ≡ load_value
```

Next we will explain how it is that words and other types become memory-storable.

6.3.3 Memory-Storable Words

The ARMv6 can natively address 8, 16, and 32-bit words in memory. In Section 6.1.1 we already introduces machine_w2b and machine_b2w which will correctly serialise such values into and from a list of bytes, taking endianness into account. What remains is the instantiation to the memory-storable interface, shown in Figure 6.6. The undefined constant in size_of is an arbitrary value of the type given by `x`. The size and alignment for words of these size is identical.

```
to_bytes    ≡ machine_w2b
from_bytes  ≡ machine_b2w
size_of x   ≡ size undefined div 8
align_of    ≡ size_of
```

Figure 6.6: *The memory-storable interface instantiated to words.*

There is also a small technical obstacle in the current Isabelle/HOL type system: one cannot simply make three statements of "an n-bit word is a member of the memory-storable class". Isabelle will only accept *one* such statement. In order to work around this, we define a type class of words whose number of bits are exactly 8, 16 or 32 and instantiate this class only:

```
len_of TYPE('a) = 8 ∨ len_of TYPE('a) = 16 ∨ len_of TYPE('a) = 32
```

We can derive such a class using a chain of classes. We can obtain a 8/16/32 word by adding a bit to a 4/8/16 word, which we can get from 2/4/8 and 1/2/4 words. We can get a 1/2/4 word from a 1/2 word, and that from a 1 bit word. The initial condition is that the one-bit numeral can be shown to be in the class of 1-bit words, 1 or 2 bit words, and 1/2/4 bit words.

Thus the statement we prove in Isabelle/HOL is:

```
instance word :: (word_len_8_16_32) mem_type
```

6.3.4 Memory-Storable Structures without Padding

Norrish's C parser [57] provides Isabelle representations of C structures in the form of records. In order to use them within our framework, we need to show those are memory-storable. Unfortunately, there is currently no way to show that all records are memory-storable, so the proof must be performed for each record type separately. We have automated the instantiation process, but will describe its steps manually for demonstration purposes on a sample C *struct*.

The C parser uses its own record package rather than the standard one that comes with Isabelle/HOL. The reason for this is that Isabelle/HOL records do not permit recursion, and it is valid C code to have a *struct* containing a pointer to a *struct* of the same type. The second difference is that Isabelle/HOL records can be extended, whereas the C parser ones cannot. This functionality is not necessary, as C only allows non-extensible structures.

Instantiation to Memory-Storable Class

```
struct example {
    short fa;
    short fb;
    int fc;
};
```

Figure 6.7: *An example of a structure in C.*

Let us instantiate *struct example* in Figure 6.7. It contains three fields, the first two of which are *short* and the last one is an *int*. On an ARMv6, a *short* is a 16-bit word and an *int* is a 32-bit word. We already know those word sizes are memory-storable, hence it becomes an issue of putting those building blocks together.

The C parser generates a record type named `example_C` with the following constructor of the same name, as well as three field accessors:

```
example_C :: 16 word ⇒ 16 word ⇒ 32 word ⇒ example_C
fa_C :: example_C ⇒ 16 word
fb_C :: example_C ⇒ 16 word
fc_C :: example_C ⇒ 32 word
```

Given that this structure has no padding, its size is simply the sum of its fields' sizes:

```
size_of x = size_of TYPE(16 word) + size_of TYPE(16 word) + size_of TYPE(32 word)
```

To fulfil the requirements of the C programming language [43], we make the structure alignment the same as the largest alignment required by its fields:

```
align_of x  ≡  max (align_of TYPE(16 word))
                   (max (align_of TYPE(16 word)) (align_of TYPE(32 word)))
```

Although these are the definitions we use for instantiating, we perform a sanity check against the size and alignment of the structure as reported by the C parser and automatically prove the following rules, which we then include in the set of terms automatically simplified by Isabelle:

```
size_of TYPE(example_C)  =  8
align_of TYPE(example_C)  =  4
```

For serialising the structure, it is sufficient to concatenate its serialised fields:

```
to_bytes x = to_bytes (fa_C x) @ to_bytes (fb_C x) @ to_bytes (fc_C x)
```

Note that the C programming language standard does not permit C compilers to rearrange the order of fields of a structure in memory. If we do not permit padding, then our memory representation of `example_C` will be identical to that generated by a standard C compiler.

We restore a structure from a list of bytes by "popping" off its fields in sequence and then continuing with the rest of the stream. This results in a rather monadic computation:

```
from_bytes bs  ≡  let (f1, bs) = mem_type_decode bs;
                      (f2, bs) = mem_type_decode bs;
                      (f3, bs) = mem_type_decode bs
                  in example_C f1 f2 f3
```

Since the structure is smaller than size of virtual or physical memory, its alignment is a power of two, and the serialisation functions are built on those of other memory-storable types, it is also memory-storable, which we proved automatically in order to have `example_C` appear in this document:

```
instance example_C :: mem_type
```

Accessing the Structure Fields

The primary requirement for accessing structure fields is knowing where in the structure they are. We generate a partial function in Isabelle from a list of maplets – given a field name, give us an offset:

```
example_C_offs :: char list ⇀ nat
example_C_offs = [''fa'' ↦ 0, ''fb'' ↦ 2, ''fc'' ↦ 4]
```

Getting a pointer with the correct address is then just adding the offset within the structure to the address of the structure in memory.

We define an address-of-field function for every structure, shown in Figure 6.8. Notice that the source pointer must be a pointer to that structure type. We also define the special syntax `s&→f`, which is shorthand for "&(s->f)" in C. This is due to C notation masking the semantics. In C, "s->f" represents a load from the address of the field when on the right side of an equation (an r-value), and a store to the address of the field when on the left of an equation (an l-value). In reality, the C

```
_&→_ :: ('a, 'p, example_C) ptr_t ⇒ char list ⇒ ('a, 'p, 'ft) ptr_t
ptr&→fld = Ptr (ptr_val ptr + of_nat (the (example_C_offs fld)))
```

Figure 6.8: *Address of structure field operator for the* `example_C` *structure (*`_&→_`*).*

compiler always keeps track of the address of a variable in memory. It is this reality which is exposed to us via the C parser, resulting in our slightly different notation being convenient. This notation also allows chaining, for example the C equivalent of "&(p->fs.fa)" becomes `p&→''fs''&→''fa''`.

There is a trick we need to perform here, however. Notice that our field accessor's function returns a `('a, 'p, 'ft) ptr_t`, where `'ft` is not bound by any other type. The function returns a pointer to a field, and fields have arbitrary types. We can *cast* that pointer to the correct type manually, but this is inconvenient. Instead, our `s&→f` notation uses a hook in Isabelle's parsing mechanism, which proceeds as follows:

- examine the left-hand side, check it is a pointer to a structure;
- find the address-of-field function for that structure;
- get the type of the field named on the right-hand side within the structure;
- translate the notation to invoking the address-of-field function on the field name and constrain the free type variable in the resulting pointer to be the field's type.

6.3.5 Memory-Storable Arrays

Our array representation is based upon the work of Harrison [27]. The representation itself is not our own work, and appears in the work of Tuch et al. [57]. We will explain the basic concept, but will focus on their memory-storable instantiation in our logic.

The two ingredients of an array are the element type, which we will denote `'a`, and an index type `'b`. We write the resulting array type as `'a['b]`. The index type `'b` must be finite, and its cardinality defines the size of the array. The C parser automatically defines a new type of the required cardinality for each array size.

Constructing and destructing arrays is achieved by the two functions index and FCP:

```
index :: ('a['b]) ⇒ nat ⇒ 'a
FCP   :: (nat ⇒ 'a) ⇒ 'a['b]
```

where index returns the nth element of an array, and FCP converts a function with type `nat ⇒ 'a` into an array such that:

```
i < CARD('b) ⟹ index (FCP g) i = g i
```

In order to show arrays are memory-storable, we will need to reconstitute arrays from a stream of bytes. For this purpose, we define utility functions for cutting up a list into lists of size n, and converting a list of element type members to an array:

```
cut_list :: 'a list ⇒ nat ⇒ 'a list list
cut_list [] n                ≡  []
cut_list (v · va) 0          ≡  [v · va]
cut_list (v · va) (Suc vb)   ≡  take (Suc vb) (v · va) ·
                                 cut_list (drop (Suc vb) (v · va)) (Suc vb)

list_to_array :: 'a list ⇒ 'a['b]
list_to_array bs             ≡  FCP (op ! bs)
```

To actually perform the instantiation of arrays to the memory-storable class, we need to pick type classes for 'a and 'b such that we can prove that we can store the array in memory. Unfortunately, in Isabelle/HOL, we can do this only *once*, meaning we need to place a limit on the size of both the size of the array, as well as the in-memory size of its element type.

We thus define the `fourthousand_count` class with the restriction that `CARD('b)` be limited to 2^{12} elements, as well as the `oneMB_size` class to limit its memory-storable members to 2^{20} bytes. These are picked arbitrarily, but when multiplied, they result in 2^{32} bytes, which happens to be the maximum memory size on a 32-bit machine.

```
to_bytes a      ≡  concat (map (λi. to_bytes (index a i)) [0..<CARD('b)])
from_bytes bs   ≡  list_to_array (map from_bytes (cut_list bs (size_of TYPE('a))))
size_of x       ≡  CARD('b) * size_of TYPE('a)
align_of x      ≡  align_of TYPE('a)
```

Figure 6.9: *Instantiation of arrays to the memory-storable type for an element type 'a and an index type 'b.*

Along with the equations in Figure 6.9, our instantiation is then:

```
instance  array :: (oneMB_size, fourthousand_count) mem_type
```

6.4 Separation Logic Constructs at the Byte Level

In Chapter 5, we introduced a separation logic working at the level of memory granularity. On the machine we presented, this granularity was 32 bits. On the ARMv6, it is 8 bits, i.e. a byte. We will now adapt the separation logic constructs from that chapter to the ARMv6 and our updated infrastructure before. We will also go over the basic byte-level concepts again before we move on to the pointer level ones, as they are crucial to the understanding of our approach to virtual memory.

6.4.1 The Fractional Heap and Memory Views

Our separation logic heap remains a *fractional heap* where each physical address is split up into a number of *slices*:

```
types  fheap = (paddr × vaddr) ⇀ 8 word
```

Like last time, the number of slices is sufficient to allocate one to each virtual address, which corresponds to the worst-case entry sharing of a page table: one entry

```
□       ≡ λ(h, r). h = empty
⊤       ≡ λ(h, r). True
P ∧* Q ≡ λ(h, r). ∃h0 h1. h0 ⊥ h1 ∧ h = h0 ++ h1 ∧ P (h0, r) ∧ Q (h1, r)
```

Figure 6.10: *Basic constructs of mapped separation logic: separating conjunction, the empty heap and universal* True*.*

mapping all virtual addresses. Likewise, our separation logic assertions, in addition to the fractional heap, take an extra parameter `r`, which we will use to indicate the physical location of the current page table, i.e. the page table *root*. Figure 6.10 shows the basic constructs. They are identical to those of the previous chapter.
Recall that: `f ▷ g ≡ λx. case f x of None ⇒ None | ⌊y⌋ ⇒ g y`

```
h_view :: fheap ⇒ vaddr ⇒ paddr ⇀ 8 word
h_view fh vp ≡ λp. fh (p, vp)
vmap_view :: fheap × paddr ⇒ vmap
vmap_view (fh, r) vp = ptable_lift (h_view fh vp) r vp
as_view :: fheap × paddr ⇒ vaddr ⇀ 8 word
as_view (fh, r) vp = (vmap_view (fh, r) ▷ h_view fh vp) vp
```

Figure 6.11: *Non-fractional views of the fractional heap for physical memory as well as a program's address space (virtual memory).*

In order to reason about objects in either physical memory or a task's address space (virtual memory) we cannot use our fractional heap as is. For the purposes of such reasoning we therefore generate non-fractional views from a fractional heap. Figure 6.11 shows these views; they are functionally identical to those in the previous chapter.

The three views represent the concepts we introduced in the last chapter. The *physical heap*, derived from the fractional heap by h_view is our physical-to-value layer. The *virtual map*, derived by vmap_view is the virtual-to-physical mapping. We combine these into the *address space*, a virtual-to-value mapping, using as_view.

Each physical address has as many slices associated with it as virtual addresses. If all slices of a physical address are present in the fractional heap, then it does not matter which slice we look at. However, when we use one slice of a byte of a PTE for resolving one virtual address and a different slice for resolving another, then those two virtual addresses end up with different views of the fractional heap: each one "sees" only the slices that are required for its page table lookup and a slice of the physical address it resolves to. Thus our flat view of physical memory, h_view, consists of all the slices attributed to a single virtual address. We can perform page table lookups in such a view of physical memory, generating a virtual-to-physical map using vmap_view. The `r` parameter represents the page table root. We then combine the two into as_view to generate a view of a virtual address space: for every virtual address, we look it up in the page table within the h_view for *that* virtual address.

6.4.2 Maps-to Predicates

As in Chapter 5 Section 5.2.3, three views of memory require three levels of maps-to predicates. All our predicates work upon `fheap × paddr`: the fractional heap and the page table root.

The first of these is our physical-to-value predicate, shown in Definition 6.4.1. A physical address can be considered to map to some value in the fractional heap if *all slices* of that physical address are in the fractional heap, containing the same value. From a separation logic perspective, we expect this mapping to behave very much like the classical separation logic maps-to predicate [48]: it requires that its domain be restricted to precisely that physical address. Indeed, we get this behaviour if we require all slices of the physical address p to be present in the fractional heap. Under separating conjunction, this excludes other predicates mentioning any slice of p, which includes having this address in the pagetable trace of a virtual address.

Definition 6.4.1 *Byte-level maps-to predicate corresponding to the physical heap: a physical address to value mapping.*

```
_ :↦p _ :: paddr ⇒ 8 word ⇒ fheap × paddr ⇒ bool
p :↦p v  ≡  λ(h, r). (∀vp. h (p, vp) = ⌊v⌋) ∧ dom h = {p} × UNIV
```

Definition 6.4.2 shows our virtual-to-physical predicate. Here, we derive the physical heap from the fractional one for the supplied virtual address and then perform a page table lookup to find the physical address. As for the consumption of resources under separating conjunction, we specify the minimum that needs to be present in the fractional heap for the page table lookup to succeed: the slice assigned to the virtual address supplied in all the physical addresses required for its pagetable lookup, i.e. in its pagetable trace. This excludes the physical-to-value mapping from using any of those addresses, as it requires *all* slices, while permitting other virtual addresses to share an identical (or partially identical) pagetable trace, as they will look at different slices of those addresses.

Definition 6.4.2 *Byte-level maps-to predicate corresponding to the virtual map: a virtual address to physical address mapping.*

```
_ :↦v _ :: vaddr ⇒ paddr ⇒ fheap × paddr ⇒ bool
vp :↦v p  ≡  λ(h, r).
                let heap = h_view h vp; vmap = ptable_lift heap r
                in vmap vp = ⌊p⌋ ∧ dom h = ptable_trace heap r vp × {vp}
```

Now that we have the virtual-to-physical and physical-to-value mappings, we can combine them. Unfortunately, due to the nature of virtual memory, there are two ways to combine them, based on whether we hit the uncommon case of the physical address a virtual address resolves to appearing in the virtual addresses pagetable trace. If it does not, we can use a separating conjunction to combine the two as in Definition 6.4.3. We believe that this is the common case, and our case study will be based on using these mappings. Nonetheless, when dealing with operating systems there is usually an exception to any rule of thumb, and so it is here. Since the operating system accesses physical memory via virtual memory addresses, its address space must be stored in a page table. Since the operating system must be

in control of its own address space, then the page table must be mapped into the very address space whose mappings it stores. Thus there will exist addresses which map to bytes of page table entries which map them. For this rather circular situation we instead proceed as in Definition 6.4.4 by combining the virtual-to-physical and physical-to-value mappings using a union of their domains. Note that by "unsafe" we do not mean it violates the logic in any way. We simply mean that after writing to such a virtual address, we cannot guarantee that the value we just wrote, or in fact any value, can be read from that virtual address.

Definition 6.4.3 *Safe version of the byte-level maps-to predicate corresponding to the address space: a virtual address to value mapping.*

```
_ :↦ _ :: vaddr ⇒ 8 word ⇒ fheap × paddr ⇒ bool
vp :↦ v  ≡  λs. ∃p. (vp :↦v p ∧* p :↦p v) s
```

Definition 6.4.4 *Unsafe version of the byte-level maps-to predicate corresponding to the address space: a virtual address to value mapping.*

```
_ :↦u _ :: vaddr ⇒ 8 word ⇒ fheap × paddr ⇒ bool
vp :↦u v  ≡  λ(fh, r).
                let heap = h_view fh vp; vmap = ptable_lift heap r
                in ∃p. vmap vp = ⌊p⌋ ∧
                       (∀vp. fh (p, vp) = ⌊v⌋) ∧
                       dom fh = ptable_trace heap r vp × {vp} ∪ {p} × UNIV
```

For the four types of maps-to predicates, we automatically generate variants such as maps-to-any. Figure 6.12 shows all generated variants for one type of maps-to predicate.

```
vp :↪ v ≡ vp :↦ v ∧* ⊤
vp :↦ - ≡ λs. ∃v. (vp :↦ v) s
vps [:↦] vs ≡ foldl op ∧* □ (map (λ(x, y). x :↦ y) (zip vps vs))
vp :↪ - ≡ λs. ∃v. (vp :↪ v) s
vps [:↦] - ≡ foldl op ∧* □ (map (λvp. vp :↦ -) vps)
vps [:↪] - ≡ foldl op ∧* □ (map (λvp. vp :↪ -) vps)
```

Figure 6.12: *Automatically generated variants of maps-to predicates, shown here for the virtual-to-value maps-to predicate.*

6.5 Type-level Separation Logic Maps-to Predicates

Having defined what happens at the byte level for single addresses, we will now use them as building blocks for type-level predicates working on pointers to memory-storable objects in memory.

Figure 6.13 shows the type-level predicates. All the predicates work using the same principle. In the style of Tuch et al. [57] we employ a *guard* on the supplied pointer. The guard is entirely independent from the fractional heap and page table root. It is there to enforce what constitutes a valid pointer: for example, there exist

```
_ ⊢ _ ↦p _ :: ('t pptr ⇒ bool) ⇒ 't pptr ⇒ 't ⇒
              fheap × paddr ⇒ bool
g ⊢ p ↦p v   ≡  addr_seq (ptr_val p) (size_of TYPE('t)) [:↦p] to_bytes v
                ⌊∧⌋ (λs. g p)

_ ⊢ _ ↦v _ :: ('t vptr ⇒ bool) ⇒ 't vptr ⇒ 't pptr ⇒
              fheap × paddr ⇒ bool
g ⊢ vp ↦v p  ≡  addr_seq (ptr_val vp)
                  (size_of TYPE('t)) [:↦v] addr_seq (ptr_val p) (size_of TYPE('t))
                ⌊∧⌋
                (λs. g vp)
_ ⊢ _ ↦ _ :: ('t vptr ⇒ bool) ⇒ 't vptr ⇒ 't ⇒ fheap × paddr ⇒ bool
g ⊢ vp ↦ v   ≡  addr_seq (ptr_val vp) (size_of TYPE('t)) [:↦] to_bytes v
                ⌊∧⌋ (λs. g vp)

_ ⊢ _ ↦u _ :: ('t vptr ⇒ bool) ⇒ 't vptr ⇒ 't ⇒ fheap × paddr ⇒ bool
g ⊢ vp ↦u v  ≡  addr_seq (ptr_val vp) (size_of TYPE('t)) [:↦u] to_bytes
                v ⌊∧⌋ (λs. g vp)
```

Figure 6.13: *Type-level separation logic maps-to predicates.*

alignment constraints both on specific architectures and in the C programming language, or we do not want the pointer to be NULL. We then use the maps-to-list variant shown in Figure 6.12 to combine a list of addresses and values (in the case of physical-to-value and virtual-to-value maps) or addresses and addresses (in the case of virtual-to-physical maps). The number of addresses that an object of type $'t$ occupies in memory is size_of $TYPE('t)$, which we described in Section 6.3.1. This means $'t$ must be memory-storable.

As we did for our byte-level maps-to predicates, we also automatically generate maps-to variants for our type-level predicates. They are shown in Figure 6.14

```
g ⊢ vp ↪ v ≡ g ⊢ vp ↦ v ∧* ⊤
g ⊢ vp ↦ - ≡ λs. ∃v. (g ⊢ vp ↦ v) s
g ⊢ vps [↦] vs ≡ foldl op ∧* □ (map (λ(x, y). g ⊢ x ↦ y) (zip vps vs))
g ⊢ vp ↪ - ≡ λs. ∃v. (g ⊢ vp ↪ v) s
g ⊢ vps [↦] - ≡ foldl op ∧* □ (map (λvp. g ⊢ vp ↦ -) vps)
g ⊢ vps [↪] - ≡ foldl op ∧* □ (map (λvp. g ⊢ vp ↪ -) vps)
```

Figure 6.14: *Automatically generated variants of typed maps-to predicates, shown here for the typed virtual-to-value maps-to predicate.*

6.6 Slice Accounting and Read-only Maps-to Predicates

In most situations, the maps-to predicates we introduced in Section 6.4.2 and Section 6.5 are sufficient. When dealing with page table modification, two consequences arise where we need predicates with a higher degree of control regarding the slices used.

The first of these consequences concerns correspondence between a page table entry as a machine word in memory and as part of a lookup path for addresses

on the page which the entry maps. We will explore this correspondence in more detail in our case study in Section 6.8. For the moment, consider the following two predicates:

```
g ⊢ pde_ptr ↦p pde ∧* g ⊢ pte_ptr ↦p pte
vp :↦v v
```

The first predicate consumes *all* slices of four physical address bytes starting at `pte_ptr`. Let us assume that the lookup path for `vp` involves both the PDE at `pde_ptr` and the PTE at `pte_ptr`. In this case the second consumes only the `vp` slice of the entire lookup path. Recall that there are 2^{32} slices per physical address. Since the granularity of mapping on the ARMv6 is smaller than 2^{32} bytes, consumption of all slices on the lookup path by a virtual-to-physical mapping is impossible by design. For a small page, only 4096 slices of the PTE and PDE will be used. Since this is separation logic, we cannot simply forget about the domain of our heap. Therefore, we need a way of keeping track of these extra slices. This can be accomplished by using the predicates in figure Figure 6.15.

```
_ of _ :↦p _ :: (vaddr ⇒ bool)
                 ⇒ paddr ⇒ 8 word ⇒ fheap × paddr ⇒ bool
slices of p :↦p v ≡
      λ(h, r). (∀vp∈slices. h (p, vp) = ⌊v⌋) ∧ dom h = {p} × slices

_ ⊢ _ of _ ↦p _ :: ('t pptr ⇒ bool)
                   ⇒ (vaddr ⇒ bool)
                     ⇒ 't pptr ⇒ 't ⇒ fheap × paddr ⇒ bool
g ⊢ slices of p ↦p v ≡
      slices of addr_seq (ptr_val p)
                  (size_of TYPE('t)) [:↦p] to_bytes v ⌊∧⌋
      (λs. g p)
```

Figure 6.15: *Physical byte-level and type-level mappings with precise slice control.*

The sliced-controlled physical mappings correspond to the ordinary one in the expected manner, i.e. by consuming the entire set of slices for the physical addresses involved:

```
p :↦p v = UNIV of p :↦p v
```

Using the slice-controlled physical mappings, we can then also create a minimal virtual-to-value mapping predicate. This predicate identifies the smallest heap necessary to be able to make a statement in our logic about a successful read from that virtual location. Since the virtual-to-physical mappings are already minimal, all we need to do is match up the slices. We present the byte and type versions of this predicate in Figure 6.16.

6.7 Interface to the C Programming Language

In the previous chapter, we used a programming language deeply embedded into our framework. While it allowed us to prove that the combination of predicates and language resulted in a separation logic, it was a very simple language with exactly one user.

```
_ :↦¹ _ :: vaddr ⇒ 8 word ⇒ fheap × paddr ⇒ bool
vp :↦¹ v ≡
      λs. ∃p. (vp :↦_v p ∧* {vp} of p :↦_p v) s

_ ⊢ _ ↦¹ _ :: ('t vptr ⇒ bool)
              ⇒ 't vptr ⇒ 't ⇒ fheap × paddr ⇒ bool
g ⊢ vp ↦¹ v ≡
      addr_seq (ptr_val vp) (size_of TYPE('t)) [:↦¹] to_bytes v
      ⌊∧⌋
      (λs. g vp)
```

Figure 6.16: *Read-only byte-level and type-level address space maps-to predicates.*

In this section, we will build up our framework to the point where it exposes two main functions which can bind to the C programming language: first loading, then storing values in memory. We will assume that the program being executed runs in virtual memory, and so must go through the page table in order to access any address in physical memory.

Please note that we will omit page table permissions for the purposes of this exercise. Since our maps-to predicates guarantee that page table lookups will succeed for the given addresses, assertions on page table permissions can be added as an extra guard on our maps-to assertions. In the context of the seL4 microkernel, this is acceptable, as valid mappings are always read/write to the kernel and the execute-never bit is not used. In order to adapt our work for verification of virtual memory-aware user programs or memory-mapped devices, addition of such a guard would be necessary. We discuss this further in Section 7.3 of Chapter 7.

6.7.1 Loading Values from the Fractional Heap

In Section 6.3.2 we showed loading of memory-storable types from a partial address-to-value heap. In order to interface with another programming language, we need to express the concept of loading an object from memory. Recall that our "state" comprises the fractional heap and a physical address pointing to the page table root. Our loading functions must thus be adapted to load objects given a pointer and a state. Figure 6.17 shows the adapted functions.

```
load_value_v :: fheap × paddr ⇒ 't vptr ⇀ 't
load_value_v s vp  ≡  load_value (as_view s) (ptr_val vp)

load_value_p :: fheap × 'b ⇒ 't pptr ⇀ 't
load_value_p s p   ≡  load_value (h_view (fst s) undefined) (ptr_val p)
```

Figure 6.17: *Loading a memory-storable object from a fractional heap state given a virtual or physical pointer.*

We can combine the loading functions with our maps to predicates. For example, these rules show the relationship between a maps-to at the address space level and the results of trying to load a value from the address space:

$$\frac{(g \vdash vp \mapsto v)\ s}{\text{load\_value\_v}\ s\ vp = \lfloor v \rfloor} \qquad \frac{(g \vdash p \mapsto v \wedge^* P)\ s}{\text{load\_value\_v}\ s\ p = \lfloor v \rfloor}$$

6.7.2 Updating the Fractional Heap

If we can load from the fractional heap, we should also be able to store to it. Conceptually, this is a matter of writing to physical addresses. However, our addresses are divided up into slices, some of which may or may not be present in a given fractional heap.

Definition 6.7.1 shows our core update function: it updates one byte in the fractional heap at the given physical address. Our version updates the slices of the physical address that are present in the fractional heap, completely ignoring those that are not present. Note that if one is not careful, this is a potentially dangerous situation. Invoking this function if not all the slices are present will cause a state in which only *part* of a physical address changes value. Combining such a state which has other slices of this physical address but with a different value will result in a vacuous state. The advantage is that our update functions always preserve the domain of the fractional heap, never adding slices or addresses to it.

In our framework the vacuous state is not a problem, however. We are interested in separation-logic-based memory safety. Arbitrary growth of the heap holds no interest for us and makes the frame rule harder to establish. Furthermore, the C parser generates guards on all accesses. As we will demonstrate soon, the guard we supply it with guarantees a non-vacuous state.

Definition 6.7.1 *Updating a single byte of the fractional heap at the level of physical addresses.*

```
fheap_update :: fheap ⇒ paddr ⇒ 8 word ⇒ fheap
fheap_update fh p v ≡ λppv.
                        if fst ppv = p
                        then Option.map (K v) (fh ppv)
                        else fh ppv
```

Scaling up the fractional heap update to a full memory-storable object update is a fairly simple process. For physical memory updates we simply discard the page table root. For virtual address space updates we first perform a page table lookup to determine the physical address to write to:

```
state_update_p :: fheap × paddr ⇒ paddr ⇒ 8 word ⇒ fheap × paddr
state_update_p s p v ≡ (fheap_update (fst s) p v, snd s)

state_update_v :: fheap × paddr ⇒ vaddr ⇒ 8 word ⇒ fheap × paddr
state_update_v s vp v ≡ case vmap_view s vp of None ⇒ s
                        | ⌊p⌋ ⇒ (fheap_update (fst s) p v, snd s)
```

Once we can write to a virtual or physical address, updating an object becomes updating the range of addresses occupied by an object of that type to the sequence of bytes it represents in memory. We only show the virtual address space updates, as the process is identical for physical pointer updates:

$$\frac{(p :\mapsto_p b \wedge^* Q)\ (fh,\ r)}{(p :\mapsto_p b' \wedge^* Q)\ (\text{fheap\_update}\ fh\ p\ b',\ r)}$$

$$\frac{(vp :\mapsto b \wedge^* P)\ s}{(vp :\mapsto b' \wedge^* P)\ (\text{state\_update\_v}\ s\ vp\ b')}$$

$$\frac{(g \vdash vp \mapsto old \wedge^* P)\ s}{(g \vdash vp \mapsto v \wedge^* P)\ (\text{store\_value\_v}\ vp\ v\ s)}$$

Figure 6.18: *Rules of byte-level updates to physical memory and virtual memory addresses, as well as typed updates to pointers in virtual memory.*

```
state_update_v_list :: fheap × paddr ⇒ (vaddr × 8 word) list ⇒ fheap × paddr
state_update_v_list s []   =   s
state_update_v_list s ((vp, v) · upds) =
        state_update_v_list (state_update_v s vp v) upds

store_value_v :: 't vptr ⇒ 't ⇒ fheap × paddr ⇒ fheap × paddr
store_value_v vp val s  ≡   state_update_v_list s
                              (zip (addr_seq (ptr_val vp) (size_of TYPE('t)))
                                (to_bytes val))
```

We have proved that the rules in Figure 6.18 hold. That is, the frame rule holds for updates to memory when considered by themselves. While this does not *guarantee* that a programming language semantics which uses our updates will obey the frame rule, they do form the most interesting part of the proof, as seen in the previous chapter.

6.7.3 Reading and Updating Memory from C

Now that we have a way of loading and storing typed values in virtual memory, we can finally create functions which hook our framework up to programs in the C programming language. We do this by providing the C parser with functions it will use to model the memory semantics.

Let us first consider what it means to perform a memory *safe* access in C. The C standard requires that the pointer has to be aligned to the size of the object pointed to. Additionally, in all systems with a virtual memory interface examined during the production of this work, it is customary in nearly all situations to make the first page of virtual memory invalid. The purpose of this is to make address zero invalid, thus giving programmers a known invalid address to initialise pointers with. Figure 6.19 shows these constraints in the form of a *c guard* in the style of Tuch et al. [57].

It is not sufficient that the pointer be aligned and not cross the start of the address space for an access to be safe. Recall from Section 6.7.2 that our access rules as well as method of updating the fractional heap require that what we are trying to access actually be present in the heap. For a single access, that would mean:

$$\text{c\_guard} \vdash p \mapsto -$$

However, a program is unlikely to work on just the minimal fractional heap required for this predicate. Therefore, we weaken it, resulting in the predicate in Definition 6.7.2. If the weaker predicate holds for a fractional heap, it is safe to both

```
c_null_guard :: ('a, 'p, 't) ptr_t ⇒ bool
c_null_guard p ≡ Addr (0 :: 'a) ∉ set (addr_seq (ptr_val p) (size_of TYPE('t)))

c_guard :: ('a word, 'p, 't) ptr_t ⇒ bool
c_guard p ≡ ptr_aligned p ∧ c_null_guard p
```

Figure 6.19: *The C pointer guard requiring alignment to the size of the pointed to object, as well as the zero address not being present in the range of addresses occupied by the object.*

read from and write to that address.

Definition 6.7.2 *Safety predicate for reading and writing by C programs.*

```
c_ptr_safe :: 't vptr ⇒ fheap × paddr ⇒ bool
c_ptr_safe p ≡ c_guard ⊢ p ↪ -
```

Now that we have defined memory safety in a C program, we can define the loading and storing interface functions under the assumptions that the C parser will insert the safety predicates on all memory accesses. As shown in Figure 6.20 storing is no different and loading assumes the load can be performed given the current heap.

```
c_load_value :: fheap × paddr ⇒ 't vptr ⇒ 't
c_load_value s ≡ the ∘ load_value_v s

c_state_update :: 't vptr ⇒ 't ⇒ fheap × paddr ⇒ fheap × paddr
c_state_update ≡ store_value_v
```

Figure 6.20: *Exported C semantics of loading and storing a value.*

We can then link our address space maps-to predicate to the C loading and storing interface, again showing that the frame rule holds on the update when considered by itself. Figure 6.21 shows the proved rules.

$$\frac{(g \vdash p \mapsto v)\ s}{\text{c\_load\_value}\ s\ p = v}$$

$$\frac{(g \vdash vp \mapsto old \wedge^* P)\ s}{(g \vdash vp \mapsto v \wedge^* P)\ (\text{c\_state\_update}\ vp\ v\ s)}$$

Figure 6.21: *The rules linking loading and storing of values in virtual memory with the address spaces maps-to predicate.*

Unlike the situation in the previous chapter, where we could prove the frame rule for *all* programs, we cannot induct over the structure of all programs expressible in SIMPL [49]. SIMPL is deeply embedded for statements, but shallowly embedded for expressions. A proof for all programs would thus need to contain a proof for all expressions in Isabelle/HOL. Hence, like Tuch et al. [57] we can only prove the frame rule for individual code components such as functions. Given that the output

```
void int_ptr_swap(int* x, int* y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

Figure 6.22: *A simple C function swapping the contents of two int pointers.*

```
int_ptr_swap_body ≡
TRY
´t :== arbitrary;;
  Guard C_Guard {|c_guard ´x|} (´t :== c_load_value ℋ ´x);;
  Guard C_Guard {|c_guard ´x|}
    (Guard C_Guard {|c_guard ´y|}
      (Guard MemorySafety {|c_ptr_safe ´x ℋ|}
        (ℋ :== c_state_update ´x (c_load_value ℋ ´y) ℋ)));;
  Guard C_Guard {|c_guard ´y|}
    (Guard MemorySafety {|c_ptr_safe ´y ℋ|} (ℋ :== c_state_update ´y ´t ℋ))
CATCH SKIP
END
```

Figure 6.23: *Definition produced by the C parser for the function in Figure 6.22*

is generated by the C parser using *only* the update and store functions we supply ourselves, the process is not complicated and could be completely automated as done by Tuch [54].

6.7.4 A Simple Example: Swapping the Contents of Two Pointers

To give a better understanding of the interaction with the C parser, we will give a simple example of proving the specification and frame rule for the simple pointer content-swapping program in Figure 6.22.

When we invoke the C parser on our file, it produces the definition for the function body shown in Figure 6.23. The overall notation is that of SIMPL [49]: we perform a sequence of operations in a TRY… CATCH block. In this case, we do not return anything in the function at any point, so there is nothing to catch. In SIMPL, statements encased in {| … |} have access to SIMPL states. In these statements, variables annotated with with ´ refer to variables in that state, e.g. ´x.

The C parser generates pointer guards around all C statements which use pointers for memory access. The Guard C_Guard {| … |} identify and label the guard itself in {| … |}. The guard we use is c_guard, which we explained in Section 6.7.3 and provided to the parser as part of the instantiation of our logic.

The other guard is *MemorySafety*. When given the "memsafe" parameter, the parser generates this guard for all writes to memory. As we mentioned in Section 6.7.3, our choice of memory-safety predicate is c_ptr_safe.

The C parser does not allow referring to addresses of local variables such as ´t; they exist only as SIMPL variables, and hence have no guards.

Guards aside, the statements themselves modify memory using c_load_value and c_state_update, introduced in Section 6.7.3. There is some extra C parser infrastructure present, in the form of the heap state variable $\mathcal{H}$. This represents the state of the

```
∀σ xv yv P.
   Γ⊢ {|σ. (c_guard ⊢ ´x ↦ xv ∧* c_guard ⊢ ´y ↦ yv ∧* P) H|}
      Call int_ptr_swap
      {| (c_guard ⊢ σx ↦ yv ∧* c_guard ⊢ σy ↦ xv ∧* P) H|}
```

Figure 6.24: *Specification and frame rule for program in Figure 6.22.*

underlying logic within the C parser's state. In the case of the logic in our work, this corresponds to the fractional heap and page table root.

Figure 6.24 shows the specification of our pointer content swap function. The specification is also the frame rule (see Figure 4.8 in Chapter 4) for this function. The rule is in the form of a Hoare triple, with the invocation of our function being the executed command.

In the precondition, we require that both the pointer arguments $´x$ and $´y$ are valid, disjoint in the fractional heap, as well as initially pointing to the values xv and yv respectively. We also admit any other separation logic predicate that is separate from these two pointers. The σ is there to give a name to the initial state, so we can refer to it in the postcondition.

In the postcondition, we refer to the pointers in the original state by superscripting them with the name we gave the state: ${}^{\sigma}x$ and ${}^{\sigma}y$. We require that they now point to yv and xv, in other words that their contents have been swapped. Our arbitrary separation logic predicate in the original state, whatever it is, must remain unaffected.

To prove the rule, we first want to reduce all the SIMPL annotations to something we can reason about using the rules in our logic. The SIMPL verification condition generator tactic (vcg) transforms Figure 6.24 into a rule containing only components of our logic:

```
(c_guard ⊢ x ↦ xv ∧* c_guard ⊢ y ↦ yv ∧* P) s ⟹
c_guard x ∧
c_guard y ∧
c_ptr_safe x s ∧
c_ptr_safe y (c_state_update x (c_load_value s y) s) ∧
(c_guard ⊢ x ↦ yv ∧* c_guard ⊢ y ↦ xv ∧* P)
  (c_state_update y (c_load_value s x) (c_state_update x (c_load_value s y) s))
```

Given our earlier rules for C loads and stores in Section 6.7.3, as well as the associativity and commutativity of separating conjunction, it is easy to prove this statement is true, and thereby not only that the frame rule holds for simple pointer contents swapping function, but that it also actually swaps the contents of the pointers.

We would further like to note that for such a simple example, it certainly looks rather complicated when parsed into Isabelle. This is deceptive. All pointers in C, as per the standard, must satisfy c_guard. Further, any logic with partial heap semantics must prove that the write is safe, hence the c_ptr_safe guard. We are after all not writing code in SIMPL, we are writing it in C and emulating its semantics in SIMPL. This extra visual complexity is also evident in the work of Tuch [54, 57]. Going from Figure 6.23 to Figure 6.24 requires one line of proof script only. The proof of the statement in Figure 6.24 is also relatively short at around 20 lines, despite the fact we presently use little automation in our framework.

6.8 Case Study

At the end of Chapter 5 we described a short case study which mapped a single page into the page table, for a simplified machine, a one-level page table and a very simple language. Now that we have introduced our entire framework, including the interface to the C language, we will demonstrate a similar study, but at a far more realistic level.

For this case study, we have chosen to examine a function of the seL4 microkernel [31] named `performPageInvocationMapPTE`. Its code is listed in Figure 6.26 on page 87. It is a deceptively simple function. Through being a simple write to memory, it places entries into the second-level page table, potentially resulting in a re-mapping of the current kernel virtual memory layout. We will discuss the code in more detail shortly.

This function is a good candidate for specifying the behaviour of within our framework, as it exposes one of the most complex and intricate situations when dealing with virtual memory. We demonstrated in Section 6.7.4 that in the simple case our logic is just another separation logic. While this is indeed our goal, it is no more interesting in our framework than in any other. Let us focus instead on a more complicated situation: given that a first-level page directory entry is already present and pointing to the second-level page table, what are the semantics of writing to the page table? If the page table defines the current view of memory, how is this view affected? Which heap resources are consumed? In particular, updating the second-level table must consume a fraction of the first-level PDE. Let us apply our logic then and demonstrate what happens.

6.8.1 The Code

Type Definitions

Before we get to the main function definition, we will give an overview of the types used. Their definitions are listed in Figure 6.25. Some of these are not important for the case study, other than to make sure the code compiles. Note that these type definitions are the final result of conditional preprocessing of the original code base for the 32-bit ARMv6 architecture.

`uint32_t` is a 32-bit unsigned integer, identical in size to the 32-bit machine word on ARMv6 when compiled for this platform.

`enum exception_t` signifies the type of return value from seL4 functions. In our case study we only see `EXCEPTION_NONE`, which signifies a normal function return.

`cap_t`, `mdb_node_t` and `cte_t` are components of the seL4 capability system. They are represented internally as arrays of words, which allows reasoning about the bit-fields inside them for verification purposes [14]. For our purposes, we are only interested in the page table system, thus we do not model the capability system. We will model any writes to these structures, but not their system-wide meaning.

`pte_t` represents a page table entry (PTE), such as the one we will be writing to the second-level page table.

```
typedef unsigned long uint32_t;

enum exception {
    EXCEPTION_NONE,
    EXCEPTION_FAULT,
    EXCEPTION_LOOKUP_FAULT,
    EXCEPTION_SYSCALL_ERROR,
    EXCEPTION_PREEMPTED
};
typedef enum exception exception_t;

struct cap {
    uint32_t words[2];
};
typedef struct cap cap_t;

struct mdb_node {
    uint32_t words[2];
};
typedef struct mdb_node mdb_node_t;

struct cte {
    cap_t cap;
    mdb_node_t cteMDBNode;
};
typedef struct cte cte_t;

struct pte {
    uint32_t words[1];
};
typedef struct pte pte_t;

struct pte_range {
    pte_t *base;
    unsigned int length;
};
typedef struct pte_range pte_range_t;
```

Figure 6.25: *The seL4 type definitions used in our case study.*

`pte_range_t` indicates a range of PTEs to write to. Please note that the `base` field does *not* indicate the address of the second-level page table nor the base address of any page or frame. The `base` field is a virtual pointer, pointing to the virtual address at which the first PTE we want to write to is accessible in the kernel's address space. The `length` field indicates how many PTEs should be written to in sequence. As we explained in Chapter 3, Section 3.3, this number is either 1 for a small page or 16 for a large page.

The Code

Figure 6.26 shows the code of `performPageInvocationMapPTE`. We shall now discuss what exactly it does, although at the C code level it is rather simple.

The `cap` and `ctSlot` arguments refer to the seL4 capability system. As part of adding the mapping, the capability `cap` to this mapping is recorded by the system in the specified `ctSlot`. This has no effect on the actual mapping.

```
exception_t
performPageInvocationMapPTE(cap_t cap, cte_t *ctSlot, pte_t pte,
                            pte_range_t pte_entries) {
    unsigned int i;

    ctSlot->cap = cap;

    for(i=0; i<pte_entries.length; i++)
    {
        pte_entries.base[i] = pte;
    }
    /* Our model does not support caches.
    cleanCacheRange((word_t)pte_entries.base,
        (word_t)&pte_entries.base[pte_entries.length-1]);
    */

    return EXCEPTION_NONE;
}
```

Figure 6.26: *The seL4 page mapping function code used in our case study.*

Next, we iterate over the number of PTEs specified by `pte_entries.length`, writing the supplied PTE `pte` that many times in the table, starting at address `pte_entries.base`.

Finally, the seL4 code causes the processor to flush the appropriate caches. As we do not model caches in our framework, we have commented out this code.

6.8.2 The Code as Seen by Isabelle

```
performPageInvocationMapPTE_body ≡
TRY
´i :== arbitrary;;
  Guard C_Guard {|c_guard ´ctSlot&→''cap''|}
    (Guard MemorySafety {|c_ptr_safe ´ctSlot&→''cap'' ℋ|}
      (ℋ :== c_state_update ´ctSlot&→''cap'' ´cap ℋ));;
  ´i :== scast 0;;
  While {|´i < length_C ´pte_entries|}
    (Guard C_Guard {|c_guard (base_C ´pte_entries + ´i)|}
      (Guard MemorySafety {|c_ptr_safe (base_C ´pte_entries + ´i) ℋ|}
        (ℋ :== c_state_update (base_C ´pte_entries + ´i) ´pte ℋ));;
     ´i :== ´i + scast 1);;
  RETURN (scast EXCEPTION_NONE)
CATCH SKIP
END
```

Figure 6.27: *Definition of* `performPageInvocationMapPTE` *from Figure 6.26 produced by the C parser.*

As with the pointer swapping example in Section 6.7.4, the C parser creates generates a SIMPL representation of the code, along with additional guards. Figure 6.27 shows this representation.

The C parser initialises `´i` to any value, then proceeds to our first statement: writing `´cap` to the `cap` field of the `ctSlot`, i.e. `´ctSlot&→''cap''`. In order to do this, we must prove that we can write to the address represented by `´ctSlot&→''cap''`,

both in terms of the heap domain containing the addresses we need to write to, as well as the pointer being correctly aligned. If these conditions are met, $\mathcal{H}$, i.e. the heap, is updated.

As per the for loop initialisation condition, `´i` is set to zero. The scast performs a cast from the signed integer literal to the unsigned `32 word` type of `´i`.

The for loop is expressed as a while loop. The loop condition is as expected: proceed while `´i` is less than the `length` field of `´pte_entries`.

The loop body proceeds as one might expect. We use `´i` as an index into the array starting at the `base` field of `´pte_entries`. If the write-safety guards apply to this address, we write `´pte` there. Then we increment `´i`.

Once the loop completes, we return `EXCEPTION_NONE`.

6.8.3 Proof Stage 1: The Function as a Heap Update

We will now proceed to verifying the semantics of `performPageInvocationMapPTE` as a page table mapping insertion function. We will do the proof in three stages. The first is to establish what the function does purely in terms of memory writes, without considering its address-space modifying potential. Next we will demonstrate what exactly is required in order to establish a small page mapping in our logic. Finally, we will use our specification from the first stage to demonstrate the context in which the function inserts a new mapping.

We begin with the first stage of our proof: defining and proving the semantics of `performPageInvocationMapPTE` purely as a heap update function. What we mean by this is looking at the function as a series of memory writes to virtual addresses which represent access to corresponding physical addresses. Note that we cannot use only address-space mappings as we did for the pointer swap example. In this case, when we write to a virtual address we also need to know specifically which physical address it maps to. While we can infer an address space mapping from a virtual and physical one as shown in Figure 6.28, we cannot easily derive a specific physical address if we try to reverse the process.

$$\frac{(g \vdash vp \mapsto_v p \wedge^* g' \vdash p \mapsto_p v)\ s}{(g \vdash vp \mapsto v)\ s}$$

Figure 6.28: *Deriving an address space mapping from virtual and physical mappings.*

Figure 6.29 shows the specification of our case study function. We will now cover its pre- and postconditions on an issue-by-issue basis. Recall that $´x$ means x in the *current* state, while ${}^{\sigma}x$ means x in the state bound to the name σ. By convention, we bind the initial state to be σ, such that ${}^{\sigma}x$ means "the original value of x".

Restrictions on the number of PTEs written. We require that the number of PTEs written be non-zero, but less than 16 (`0x10` in hexadecimal):

```
length_C ´pte_entries ≤ 0x10
0 < length_C ´pte_entries
```

Heap state root is unchanged. Recall that the heap state $\mathcal{H}$ consists of the fractional heap and the page table root. The function may modify the heap state $\mathcal{H}$, but

```
Γ⊢ {|σ. (c_guard ⊢ ´ctSlot&→''cap'' ↦ - ∧*
         c_guard ⊢ ptr_seq (base_C ´pte_entries)
                           (unat (length_C ´pte_entries)) [↦v] pte_ptrs ∧*
         c_guard ⊢ pte_ptrs [↦p] - ∧* P)
         H ∧
        0 < length_C ´pte_entries ∧
        length_C ´pte_entries ≤ 0x10 ∧
        length pte_ptrs = unat (length_C ´pte_entries) ∧ snd H = r|}
   Call performPageInvocationMapPTE
   {| (c_guard ⊢ σ ctSlot&→''cap'' ↦ σ cap ∧*
     c_guard ⊢ ptr_seq (base_C σ pte_entries)
                       (unat (length_C σ pte_entries)) [↦v] pte_ptrs ∧*
     c_guard ⊢ pte_ptrs [↦p] replicate (unat (length_C σ pte_entries)) σ pte ∧* P)
     H ∧
    snd H = r|}
```

Figure 6.29: *The specification of* `performPageInvocationMapPTE` *in terms of heap updates.*

leaves the page table root unchanged. Thus we constrain the root in the precondition to the free variable `r`, and demonstrate this is still true in the postcondition:

$$\text{snd } \mathcal{H} = r$$

We will now focus on the contents of the heap.

Setting the capability in the capability table entry. At the start, we require a valid address-space mapping for a `cte_C` virtual pointer, but we do not care about the value it points to. Given this precondition, we show that the function updates the `cap` field of the structure pointed to by the original `ctSlot` to the original argument `cap`:

$$\begin{aligned}\text{PRE:}\quad & \text{c\_guard} \vdash \text{´ctSlot\&→''cap''} \mapsto -\\ \text{POST:}\quad & \text{c\_guard} \vdash \sigma\,\text{ctSlot\&→''cap''} \mapsto \sigma\,\text{cap}\end{aligned}$$

The virtual addresses of the PTEs. We assume that the sequence virtual pointers we will be writing to maps to a sequence of distinct physical addresses. This is expected, as second-level tables are required to be contiguous in memory. We prove this mapping is not disturbed by the function:

$$\begin{aligned}\text{PRE:}\quad & \text{c\_guard} \vdash \text{ptr\_seq (base\_C ´pte\_entries)}\\ & \qquad \text{(unat (length\_C ´pte\_entries))} [\mapsto_v] \text{ pte\_ptrs}\\ \text{POST:}\quad & \text{c\_guard} \vdash \text{ptr\_seq (base\_C } \sigma\,\text{pte\_entries)}\\ & \qquad \text{(unat (length\_C } \sigma\,\text{pte\_entries))} [\mapsto_v] \text{ pte\_ptrs}\end{aligned}$$

The PTEs in physical memory are updated. We assume that the sequence of physical pointers representing the PTEs, `pte_ptrs`, is present in the heap. We are not concerned about the initial values. From this we show that after executing the function, all these physical pointers will point to the same value, the `pte` argument given originally:

$$\begin{aligned}\text{PRE:}\quad & \text{c\_guard} \vdash \text{pte\_ptrs} [\mapsto_p] -\\ \text{POST:}\quad & \text{c\_guard} \vdash \text{pte\_ptrs} [\mapsto_p] \text{ replicate}\\ & \qquad \text{(unat (length\_C } \sigma\,\text{pte\_entries))} \ \sigma\,\text{pte}\end{aligned}$$

The framing condition. This is, after all, separation logic. When we add extra predicates to describe our heap state, then if they describe separate areas of the heap to the other predicates (as indicated by separating conjunction), then these extra predicates should be unaffected. This is the frame rule, and we do prove that it holds for this function:

PRE: P
POST: P

```
∃len. len = length_C ´pte_entries ∧
      len ≤ 0x10 ∧
      ´i ≤ len ∧
      0 < len ∧
      length pte_ptrs = unat len ∧
      ´cap = σ cap ∧
      ´ctSlot = σ ctSlot ∧
      ´pte = σ pte ∧
      ´pte_entries = σ pte_entries ∧
      snd H = r ∧
      (c_guard ⊢ drop (unat ´i)
                  (ptr_seq (base_C ´pte_entries)
                     (unat (length_C
                              ´pte_entries))) [↦v] drop (unat ´i) pte_ptrs ∧*
       c_guard ⊢ take (unat ´i)
                  (ptr_seq (base_C ´pte_entries)
                     (unat (length_C
                              ´pte_entries))) [↦v] take (unat ´i) pte_ptrs ∧*
       c_guard ⊢ drop (unat ´i) pte_ptrs [↦p] - ∧*
       c_guard ⊢ take (unat ´i) pte_ptrs [↦p] replicate (unat ´i) ´pte ∧*
       c_guard ⊢ ´ctSlot&→''cap'' ↦ ´cap ∧* P)
       H
```

Figure 6.30: *Loop invariant used in the proof of the specification in Figure 6.29.*

Proving the specification in Figure 6.8.3 is purely an exercise in matching up the Hoare logic reasoning of SIMPL with standard reasoning about separation logic assertions. At this stage, we do not consider the function as performing any kind of page table manipulation at all, just as writes to memory. In a way, it is just a more complicated version of our pointer swap example from Section 6.7.4. Using the loop invariant in Figure 6.30, our proof of the function specification comes to 75 lines of proof script. The proof is straightforward, although at this stage we have little automation, thus much of it is simple reordering of maps-to predicates under separating conjunction to get rules to apply. We did not experience any added complexity from reasoning about writing to virtual memory.

6.8.4 Proof Stage 2: A Heap that Contains a Mapping

Given the specification in Section 6.8.3, we now know exactly what the function does. In order to confirm it can indeed add a virtual-to-physical page table mapping for the current state, we need to place it in the correct context. In order to describe this context, we need to first identify what sort of physical heap state is

actually equivalent to a virtual-to-physical mapping of a small page on the ARMv6 architecture.

Figure 6.31 shows the preconditions required in order to assert that using a page table root r, there exist mappings for all virtual addresses within the page table. Furthermore, in terms of accounting for slices, these are the minimal requirements of a heap which maps a small page.

```
⟦decode_pde pde = PageTablePDE pt; decode_pte pte = SmallPagePTE base perms;
 sz = page_size ArmSmallPage; page_aligned ArmSmallPage vbase;
 (g ⊢ set (addr_seq (Addr vbase) sz) of Ptr r + vaddr_pd_index vbase ↦p pde ∧*
  g ⊢ set (addr_seq (Addr vbase) sz) of Ptr pt + vaddr_pt_index vbase ↦p pte ∧*
  R r)
  (h, r)⟧
⟹ (addr_seq (Addr vbase) sz [:↦v] addr_seq base sz ∧* R r) (h, r)
```

Figure 6.31: *The correspondence theorem between a page table entry (PTE) and the page of virtual addresses it maps to physical ones.*

Assuming we want a small page with a virtual base address of $vbase$ to be mapped to a contiguous sequence of bytes in physical memory at physical address $base$, let us examine the demands on the heap. We will abbreviate the page size in bytes, page_size ArmSmallPage, as sz. This size is $2^{12} = 4096$ bytes.

Firstly, a page in memory must start at a multiple of its page size. Hence $vbase$ must be aligned to the size of a small page:

```
page_aligned ArmSmallPage vbase
```

Next, let us consider the slices required. Recall from Section 6.4.1 that for the purposes of page table lookup, each virtual address gets its own view of physical memory by using a sliced view derived from the address itself. This means that for a lookup of virtual address vp to succeed, the vp slice of any PDE and PTE in its lookup path must be present in the heap. It then follows that for a whole *page* of virtual addresses starting at $vbase$, the slices available must be:

```
set (addr_seq (Addr vbase) sz)
```

If we construct a $32\ word$ pointer from the page table root r, we can add the page directory index obtained from $vbase$ to it to obtain the PDE. This is identical to the left-shift-by-two in the page table lookup procedure in Section 6.2.2, except here we exploit pointer addition to perform the multiplication for us. Recall that size_of $TYPE(32\ word) = 4$. The resulting physical pointer must point to the PDE whose slices required for the page must be in the heap:

```
g ⊢ set (addr_seq (Addr vbase) sz) of Ptr r + vaddr_pd_index vbase ↦p pde
```

If we decode the PDE, it needs to lead us to a second-level page table in physical memory:

```
decode_pde pde = PageTablePDE pt
```

Similarly to the way we extracted the PDE, we extract the PDE starting at the base address of the page table and adding the page table index obtained from $vbase$:

```
g ⊢ set (addr_seq (Addr vbase) sz) of Ptr pt + vaddr_pt_index vbase ↦p pte
```

Finally, if we decode that pte, the result needs to be a small page, with a physical base address of $base$:

```
decode_pte pte = SmallPagePTE base perms
```

If all these conditions are fulfilled, we can conclude that the entire small page is mapped. The slice consumption outlined above corresponds to that of the virtual-to-physical mapping:

```
addr_seq (Addr vbase) sz [:↦v] addr_seq (Addr base) sz
```

Finally, we add the frame condition `R r` which is preserved across this state rearrangement. This is not strictly necessary, but in our experience it makes the rule more useful in the context of proof.

6.8.5 Final Proof: A Function that Maps

Given that we have the specification of `performPageInvocationMapPTE` as well as the physical requirements of which heap conditions constitute a small page mapping, we can now expand the precondition of the specification, have them pass through to the postcondition via the frame rule, and finally rewrite the postcondition using the mapping rule in Figure 6.31. We can do this via the consequence rule in SIMPL, which we present for completeness in Figure 6.32. Interesting symbols aside, one can see that it allows precondition strengthening and postcondition weakening of any existing SIMPL program specification.

$$\frac{\begin{array}{c}\forall z.\ \Gamma,\Theta\vdash_{/F} (P'\ z)\ c\ (Q'\ z),(A'\ z) \\ \forall s.\ s \in P \longrightarrow (\exists z.\ s \in P'\ z \wedge Q'\ z \subseteq Q \wedge A'\ z \subseteq A)\end{array}}{\Gamma,\Theta\vdash_{/F} P\ c\ Q,A}$$

Figure 6.32: *The consequence rule of SIMPL. It allows strengthening the precondition and weakening the postcondition of an existing specification.*

In order to rewrite the postcondition, we need one more rule to make our slice accounting break even. Our function writes to a full physical address, i.e. *all* slices, while our mapping rule only uses some of them to create a mapping. In order to specify "the *other* slices", we use the splitting rule in Figure 6.33. Note that `- S` is the set inverse of `S`, i.e. `UNIV - S`.

$$(g \vdash p \mapsto_p v)\ s = (g \vdash S\ of\ p \mapsto_p v \wedge^* g \vdash (-S)\ of\ p \mapsto_p v)\ s$$

Figure 6.33: *Splitting a physical maps-to predicate into two read-only maps-to predicates with complementary slice sets.*

Using this rule, it is finally possible to express `performPageInvocationMapPTE` in terms of the semantics of a function that adds a mapping. Figure 6.34 shows the completed rule for using the function for mapping in a small page with a physical base address of `base` as contiguous sequence of bytes in virtual memory at virtual address `vbase`.

The large size of this rule is due to the seL4 function not mentioning the page directory or second-level page table directly, but rather only writing to individual PTEs. Nonetheless, the proof is easier than it would seem, as we are strengthening the precondition and weakening the postcondition, with disjoint predicates passed through via the frame rule. The proof takes 24 lines of proof script.

```
Γ⊢ {|σ. (c_guard ⊢ ´ctSlot&→''cap'' ↦ - ∧*
         c_guard ⊢ base_C ´pte_entries ↦v pte_ptr ∧*
         c_guard ⊢ pte_ptr ↦p - ∧*
         c_guard ⊢ set (addr_seq (Addr vbase)
                         sz) of Ptr r + vaddr_pd_index vbase ↦p pde ∧*
         P)
         H ∧
        length_C ´pte_entries = 1 ∧
        index (pte_C.words_C ´pte) 0 = pte ∧
        pte_ptr = Ptr pt + vaddr_pt_index vbase ∧
        decode_pde pde = PageTablePDE pt ∧
        decode_pte pte = SmallPagePTE base perms ∧
        sz = page_size ArmSmallPage ∧
        page_aligned ArmSmallPage vbase ∧ snd H = r|}
    Call performPageInvocationMapPTE
    {| (c_guard ⊢ σ ctSlot&→''cap'' ↦ σ cap ∧*
      c_guard ⊢ base_C σ pte_entries ↦v pte_ptr ∧*
      addr_seq (Addr vbase) sz [:↦v] addr_seq base sz ∧*
      c_guard ⊢ (- set (addr_seq (Addr vbase) sz)) of pte_ptr ↦p σ pte ∧* P)
      H ∧
     snd H = r ∧ index (pte_C.words_C σ pte) 0 = pte|}
```

Figure 6.34: *The specification of* `performPageInvocationMapPTE` *as a page table mapping function.*

Once again, here is an aspect-by-aspect breakdown of the rule.

Framing condition, capability setting and root preservation. These remain identical to Section 6.8.3.

Page size As in Section 6.8.4, we abbreviate the page size in bytes as:

```
sz = page_size ArmSmallPage
```

Restrictions on the number of PTEs written. This time we are specifically mapping a single small page, we restrict the number of modified PTEs to one:

```
length_C ´pte_entries = 1
```

Address of the PTE. We extract the contents of `´pte` into the free variable `pte`. We also require that the page table index in `vbase` and the address of the PTE agree:

```
index (pte_C.words_C ´pte) 0 = pte
pte_ptr = Ptr pt + vaddr_pt_index vbase
```

Furthermore, the address of the PTE must be accessible via the virtual address specified in the `base` field of `´pte_entries`. This is very similar to Section 6.8.3, but with only one PTE. It carries through to the postcondition:

```
PRE:  c_guard ⊢ base_C ´pte_entries ↦v pte_ptr
POST: c_guard ⊢ base_C σ pte_entries ↦v pte_ptr
```

PDE and PTE contents: as in the mapping rule in Figure 6.31, we require that the PDE points to a page table when decoded, and that the PDE contains the base physical address of the page in physical memory:

```
decode_pde pde = PageTablePDE pt
decode_pte pte = SmallPagePTE base perms
```

Alignment of `vbase` . As before, `vbase` must be page aligned:

```
page_aligned ArmSmallPage vbase
```

The mapping. We start with the necessary slices of the PDE being available to map the page, as well as `pte_ptr` pointing to some existing PTE in memory:

```
PRE: c_guard ⊢ pte_ptr ↦p -
PRE: g ⊢ set (addr_seq (Addr vbase)
                sz) of Ptr r + vaddr_pd_index vbase ↦p pde
```

After the function is done, we have consumed the supplied PDE slices within the virtual-to-physical mapping for the entire page, but we are left over with the anticipated surplus of slices from the PDE, corresponding to the complement of the ones we used in the PDE:

```
POST: addr_seq (Addr vbase) sz [:↦v] addr_seq base sz
POST: c_guard ⊢ (- set (addr_seq (Addr vbase) sz)) of pte_ptr ↦p σpte
```

This concludes our case study. As we have demonstrated, it is possible to reason about page table manipulation in our framework. Doing so remains somewhat complicated, but this is entirely expected when dealing with virtual memory reasoning involving page table modification. As we demonstrated, it is convenient that we can develop the specifications for these complicated cases in a modular fashion, by first specifying what a function does in terms of memory writes, and performing the reasoning about virtual memory as a separate step.

The novelty in our framework is visible in perhaps the smallest part of the proof: the framing condition. By specifying exactly what changed in the heap, we have also specified what didn't. Although we still consider pagetable manipulation the most complex case within our framework, it certainly is not the end of the world. This can easily be seen when supplying a different page mapping as `P` in our rule. If this mapping uses a different PTE, even if the PDE is the same, then we can guarantee it is unaffected by the insertion of a new mapping by `performPageInvocationMapPTE`. This includes any other predicate that is separate in the heap, as enforced by separating conjunction.

As promised, we did not enter any special state, but remained in our framework throughout.

Chapter 7

Discussion and Conclusion

7.1 Summary

In this work, we have extended the reach of separation logic to the domain of reasoning about programs executing in a virtual memory environment. We have demonstrated that one can talk about virtual memory, physical memory, and their combinations simultaneously, without the need for entering special "modes". Our basis in separation logic allowed us to express the effects of accessing memory in terms of not only what changed, but also infer what did *not*. This included accesses to the page table which modify the layout of virtual memory itself. We showed that the basic idea works for a simple machine with a trivial one-level page table, and does in fact form a separation logic. We also demonstrated that our idea extends to reasoning about a real machine, a two-level page table with superpages, and that our logic can be plugged into an existing framework for reasoning about C programs, in the form of the L4.verified project's C parser [57].

We demonstrated, over the course of two case studies, that it is possible to reason about writes to the page table in our framework while preserving the frame rule. The case study using our initial, plain untyped framework in Chapter 5 did so for a one-level page table on a simplified architecture. Our case study in Chapter 6 demonstrated this in our full framework, with a realistic page table model based on the ARMv6 architecture, by verifying the exact conditions necessary for making a C function insert a page table mapping for a small page. These studies demonstrate it is possible to manipulate the page table while maintaining use of the frame rule to infer not just what changed, but also what didn't change, thus bringing separation logic-style modularity to the virtual world.

In short, our approach allows local reasoning about programs running in virtual memory, despite the complications that working with virtual memory entails.

Figure 7.1 shows the lines of code and proof script as used in the various components of our work. At least a third of these consist of comments and internal documentation.

7.2 Discussion

While our work does demonstrate our core idea as advertised, and while we do outline an initial framework for working with C programs in virtual memory, it is important to point out that our work is *one* way of doing so. To the best of our

Component	File type	Lines
Simple Logic	Proof Script	6322
Full Logic	Proof Script	7041
	ML code	823
C Parser Integration	Proof Script	1326
	ML code	1126
Full Logic Case Study	Proof Script	567

Figure 7.1: *Lines of code and proof script in the different components of our work.*

knowledge, we appear to be the first person[1] to create such a unified memory model and connect it to separation logic. As we have made decisions that could possibly have been done differently, we believe these are worthy of discussion.

No Page Table Permissions Visible in Maps-to Predicates

There is an obvious omission in our maps-to arrows. While our model of the ARMv6 page table does include the decoding of page table properties, we do not use them in the maps-to assertions themselves. Initially, we omitted them because they were unnecessary to prove that our core idea works. When we investigated the seL4 microkernel, we found that the kernel *only* creates mappings that it can read and write to and does not use the no-execute flag nor domains. Additionally, its single-processor nature means that the memory-ordering bits are largely unimportant. Since our work does not examine caches or interfacing to devices (see Section 7.3), the cacheable and bufferable bits were also irrelevant in the model used in our work.

Thus, while adding an extra "permission guard" on all our virtual memory maps-to predicates is not hard, in our model as it stands in this work, such a guard does not actually provide any extra information to the reader.

Upon exploration of some of the topics we mention in Section 7.3, the concept of permission guards is worth revisiting.

Slices vs. Bornat Model of Permissions

Bornat's model of permissions [10] involves adding a rational number to all maps-to assertions in separation logic, as well as all values in the heap. If that number is 1, we can read and write to an address. If it is less, but greater than 0, we can only read. This allows splitting a value at an address into a number of pieces, allowing specification of read-only sharing of memory between concurrent program components.

To anyone familiar with Bornat's model, our concept of slices would seem familiar. In fact, it is possible to express our slices in terms of Bornat's permission model, if we pick the right rational numbers and are careful when reasoning about them. Why then do we not use the permissions model?

The foremost reason is a conceptual one. We know ahead of time the worst possible situation that can happen on the system in terms of sharing one page table

[1] Throughout this work, we have used the royal "we" to indicate the author.

entry: all virtual addresses sharing the same entry. This gives us an upper bound, meaning we have no need of rational numbers. By cutting up *every* physical address into the upper-bound number of slices, we also gain the ability to give each virtual address a "view" of physical memory. This means that in the common case of accessing virtual memory addresses, when we do not modify the page table and hence change the memory layout, we do not need to work with fractional arithmetic.

The other reason is convenience. Since our number of slices per physical address is constant, we do not need to put a fraction in the range of our heap function. Instead, we simply make its domain be the product of virtual and physical addresses. By doing so, we can use the normal Isabelle/HOL map operations, as well as everything that has been proven about them over the years.

As always, there is a trade off to our choice of slices vs. permissions. With permissions, every time one splits an address, one gets two fractions to carry around with the resulting predicates until they are merged again. One also has to deal with adding up fractions and making sure the result is not greater than 1. With slices, the information on which slices of a physical address are used by a virtual address can be inferred from the virtual address itself, and several tricky cases go away. The price we pay is that we derive everything from the worst-case situation, thus we have a constant excess of slices. For instance, when we take a PTE on ARMv6, the four physical addresses there each have the full complement of slices. If we then use that PTE to map a small page, 2^{12} addresses will be mapped. That leaves $2^{32} - 2^{12}$ slices that are "virtual": they do nothing except serve as a token to keep on the side to be able to reconstruct the full physical addresses of the PTE.

Correctness

As we mentioned in Chapter 1, nothing can be proven completely correct. The only concept of correctness is with regard to a specification.

In our work, we use Norrish's C parser [57], which formalises a subset of the C99 programming language standard into Norbert Schirmer's SIMPL [49] encoding in Isabelle/HOL. We choose to trust that this mapping is correct. It has been heavily tested, but the translation has not yet been verified to conform to the specification of the C programming language subset we use. As the C programming language has features which are implementation-dependent, architecture-dependent and compiler dependent [43], there is unlikely to ever be a specification of *the* C programming language, and certainly there is no way to logically prove that is is authoritative.

Our ARMv6 page table model is likewise developed after the fact from the ARM reference manual [6], as well as examination of the seL4 code base to confirm that domains are not used. We are confident that our model represents the behaviour specified in the ARM reference manual for the ARMv6 architecture in non-legacy mode, but to trust that our model is correct, one needs to trust that the manual is correct.

Finally, we use the Isabelle/HOL theorem prover for every part of our model, and even for displaying all formulas in this document. As we mentioned in Chapter 1, Isabelle is an LCF-style prover, an approach which is designed to maximise reliability. It is written in Standard ML, a functional language with a strict type system, and one of the few languages with an actual formal specification of its semantics. Nonetheless, even here there is room for error, although given the years of usage

and testing performed on Isabelle's proof kernel, we have very high confidence that what Isabelle admits as proof is indeed true.

7.3 Future Work

In this section we will outline next steps and possible future directions one can take our work.

7.3.1 Translation Lookaside Buffer

As we discussed in Section 3.2.3 of Chapter 3, the translation lookaside buffer (TLB) is a hardware cache of mappings contained in the page table. When TLB entries constitute a subset of the mappings in the page table, everything is fine. The difficulty occurs when there is a possible disrepancy. For example, if a mapping is currently cached by the TLB and we change this mapping in the page table, the hardware will ignore our change when looking up a virtual address that corresponds to this mapping. The hardware provides TLB *flush* instructions for this reason. They clear a subset, or potentially all, entries from the TLB. Thus modelling only the page table does not encompass all the issues involved in virtual memory.

There are several ways to get out of this. The most obvious way is to treat the TLB the same way as the L4.verified project treats virtual memory: write down well-reasoned assertions about properties memory accesses should have such that TLB consistency is maintained. The most painful way is to model the contents of the TLB itself and reason about what mappings are cached in it.

We claim that there is a better way to model TLB behaviour in our framework. Our initial concept proceeds as follows. Firstly, the specific situation we are trying to avoid is a memory read or write to a virtual address whose mapping has been modified but that might also be in the TLB. This means we do not care what set of mappings is currently in the TLB. We only care that we do not make any claim about where a virtual address maps to unless we are certain that its lookup path does not take it through a page table entry out of sync with the TLB. Following this line of thought, a write to a page table entry "taints" it until a TLB flush makes it safe to use again.

Our proposal therefore involves adding a "dirty" flag to every byte in the physical heap. When a physical address is written to, we set its dirty flag in the heap. We then make any page table lookup involving that physical address fail. In this situation, we would be unable to assert a virtual-to-physical mapping such as:

$$vp :\mapsto_v p$$

Once a TLB flush occurs which clears any conflicting mapping from the TLB, the dirty flags will be removed from the corresponding page table entries, at which point we can use them for virtual-to-physical mappings again.

Although we believe our approach has merit, there are a number of concerns which need to be explored in terms of its convenience:

- A correspondence needs to be established between the TLB flush instructions on the given architecture and the page table entries involved. This is the only way to remove a dirty flag from a physical address.

- After we write the page table entry that establishes a page table mapping, we still need to be able to state that mapping in our logic. If we simply make page table lookups unsuccessful, such a predicate may become very inconvenient.
- We do not yet know how the addition of a dirty flag to all physical addresses would help or hinder reasoning about more complicated concepts such as devices or caches.

If these concerns were addressed, this approach to TLB management would allow reasoning about TLB consistency without having to model the entire memory management unit on a bit-by-bit level of detail.

7.3.2 Caching

In our framework, we have made no attempts at reasoning about cache mechanisms, although we covered the basics of caching in Section 3.2.4 of Chapter 3. The cache we are referring to is the onboard memory of the processor, which the processor uses to prevent having to access the comparatively much slower RAM.

The presence of caches can create multiple problems when they are not handled properly. The type of problems depend largely on the type of the cache. For example, we previously outlined the implementation-dependent issues with virtually indexed, physically tagged caches on the ARMv6, where two virtual addresses mapping to the same physical address could be stored in two different cache lines.

As with the TLB, when modifying page table mappings, there are consistency problems between the main memory and the cache. Unless instructed to flush caches, the processor will load from the cache. An example mistake is forgetting to issue a cache flush when mapping a physical memory into a process' address space which was once used by a different process. This can result in addresses with stale entries sitting in the processor's cache which will be written out to RAM the moment the new process tries to access them, resulting in erronous behaviour even in correct programs.

Once again, our interest lies not in what is and is not in the cache, but we remain very interested in how to specify cache behaviour similarly to the idea we outlined for the TLB, i.e. annotations to the system model that implicitly result in correct behaviour with respect to cache coherency. We believe that development of such a model, especially if it allows preservation of the modularity of separation logic, would be an exciting research area.

7.3.3 Devices and IOMMUs

Our current framework does not address issues resulting from dealing with hardware devices whose inputs and outputs are mapped into physical memory. In our current model, we can either treat such areas as ordinary physical memory, or we can leave such areas out of the partial heap state to begin with. Beyond this, any actions performed by devices would have to be modelled explicitly. From this angle, the problem of devices is orthogonal to the issue of virtual memory and as such, outside the realm we are exploring.

An input/output memory management unit (IOMMU) is to devices what the normal MMU is to the processor. That is, it translates addresses requested by the

device into addresses in physical memory. Although we do not explore this in detail as we are not at this point interested in device verification, the IOMMU translation mechanism obeys the same principles encoded in our logic and as such can also be reasoned about using separation logic.

7.3.4 Reasoning About Multiple Page Tables at Once

All the case studies in this work involve a process modifying its own address space by writing to its own page table. We focus on this as it is the most complicated situation that can arise which we can think of. Unfortunately, we have not explored in detail how our separation logic assertions can be disassembled and reassembled upon switching processes and thus switching the active page table root.

Naturally, as we have shown that our logic follows the principles of separation logic, it will continue to work in a modular fashion for any predicates we can show to be separate. The slicing model does not permit modelling of all possibilities, however. For example, consider two processes, each of which has a separate page directory. If we wish to share a second-level page table between these processes, then we must make sure that none of the pages mapped by those page table entries have the same virtual address. While the hardware will allow it, in our slicing model these entries cannot be considered separate.

Furthermore, while we explored the management of "virtual" slices within the same address space deriving from the same page table root, we have not examined in detail how well the concept works in practice when reasoning about multiple page table roots simultaneously or when switching page table roots.

In other words, our focus has been on verifying the operating system alone. In order to branch out into verification of user programs in the context of the operating system, the multiple page table root issue will need to be explored.

7.3.5 Larger Case Study

In Section 7.3.4 we mentioned multiple page table roots, switching page table roots, and user programs. The reason we talk about these in the future work section is that while this work presents our logic and thus a way to pull separation logic into the domain of virtual memory, we can only make predictions at how it can be used in the context of a larger project. How it will actually be used depends on the project itself.

We believe that a suitable larger case study for our framework would be the verification of a system pager. A pager is a subsystem or process whose job it is to swap the data of processes between the main memory and the disk, mapping and unmapping pages in and out of page tables as necessary. This type of memory management is difficult to get right, as a pager must make sure that a process only has access to its own data rather than another process', while constantly managing reuse of the same physical memory.

Involving a new framework into a larger project such as a pager component would likely facilitate construction of more effective memory management predicates and ironing out any kinks. We believe the situation would be analogous to comparing the case study of Chapter 5 with the corresponding case study in our full framework in Chapter 6. Despite a more complicated architecture, page table model, as well as the presence of C semantics, the reasoning in the full framework is, in our

opinion, far easier to understand. We hope that our framework will see such further improvement in the future.

7.4 Concluding Remarks

We stated our goal as moving the world of operating system verification one step closer towards realism. By introducing a way of looking at virtual and physical memory simulaneously, while being able to prove properties of how one affects the other and maintaining the ability to reason locally, we believe that we have accomplished that goal.

Bibliography

[1] Frama-C. `http://frama-c.com`.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[3] Elyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In C.R. Ramakrishnan and Jakob Rehof, editors, *Proc14th Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2008.

[4] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224, Toronto, Canada, October 2008. Springer.

[5] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. Automated verification of a small hypervisor. In Peter O'Hearn, Gary T. Leavens, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *Lecture Notes in Computer Science*, pages 40–54, Edinburgh, UK, August 2010. Springer.

[6] ARM Limited. *ARM Architecture Reference Manual*, June 2000.

[7] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[8] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[9] Richard Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.

[10] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proc. 32nd POPL*, pages 259–270. ACM, 2005.

[11] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15:412–447, 1997.

[12] Rod 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.

[13] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proc. 22nd LICS*, pages 366–378. IEEE Computer Society, 2007.

[14] David Cock. Bitfields and tagged unions in C: Verification through automatic generation. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, Aug 2008.

[15] Ernie Cohen, Michal Moskał, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. `http://research.microsoft.com/apps/pubs/default.aspx?id=77174`, 2008.

[16] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, September 2006.

[17] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, 2007.

[18] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conference Proceedings, 1979 National Computer Conference*, pages 329–334, New York, NY, USA, June 1979.

[19] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13:709–745, July 2003.

[20] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Proc. 6th ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[21] Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *TPHOLs '03*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 2003.

[22] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.

[23] Holger Gast. Reasoning about memory layouts. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 628–643. Springer, 2009.

[24] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, June 1974.

[25] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation (Lecture Notes in Computer Science)*. Springer, 1 edition, January 1980.

[26] Per Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13:238–241, April 1970.

[27] John Harrison. A HOL theory of Euclidean space. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, Oxford, UK, 2005. Springer-Verlag.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[29] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005. `http://www.cs.ru.nl/H.Tews/Plos-2005/ecoop-plos-05-letter.pdf`.

[30] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[31] 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[32] Rafal Kolanski. A logic for virtual memory. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proc. 3rd Int'l Workshop on Systems Software Verification (SSV'08)*, volume 217 of *ENTCS*, pages 61–77. Elsevier, February 2008.

[33] Rafal Kolanski and Gerwin Klein. Mapped separation logic. In Jim Woodcock and Natarajan Shankar, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 15–29, Toronto, Canada, Oct 2008. Springer-Verlag.

[34] Rafal Kolanski and Gerwin Klein. Types, maps and separation logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 276–292, Munich, Germany, Aug 2009. Springer-Verlag.

[35] OK Labs. OKL4 microvisor. `http://www.ok-labs.com/`.

[36] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. *Software Engineering and Formal Methods, IEEE International Conference on*, 0:2–12, 2005.

[37] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[38] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39:70–77, September 1996.

[39] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Towards formal verification of memory properties using separation logic. In *22nd Workshop of the Japan Society for Software Science and Technology, 2005. INRIA Logic for Small-step Cminor 29*, 2005.

[40] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[41] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. Key-c: A tool for verification of c programs. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, CADE-21, pages 385–390, Berlin, Heidelberg, 2007. Springer-Verlag.

[42] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[43] Michael Norrish. C formalised in HOL. Technical report, University of Cambridge, 1998.

[44] Peter W. O'Hearn and Peter W. O'hearn. Resources, concurrency and local reasoning. In *Theoretical Computer Science*, pages 49–67. Springer, 2004.

[45] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL '04: Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280. ACM, 2004.

[46] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL '05: Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–258. ACM, 2005.

[47] Gerald J. Popek, Mark Kampe, Charles S. Kline, Allen Stoughton, Michael Urban, and Evelyn J. Walton. Ucla secure unix. *Managing Requirements Knowledge, International Workshop on*, 0:355, 1979.

[48] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[49] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[50] Green Hills Software. μ-velOSity™real-time microkernel. `http://www.ghs.com/products/micro_velosity.html`.

[51] QNX Software Systems. QNX realtime operating system. `http://www.qnx.com/`.

[52] Hendrik Tews, Marcus Völp, and Tjark Weber. Formal memory models for the verification of low-level operating-system code. *J. Autom. Reasoning*, 42(2-4):189–227, 2009.

[53] Hendrik Tews, Tjark Weber, and Marcus Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. In *Proc. 3rd Int. WS on Systems Software Verification (SSV'08), volume 217 of ENTCS*, pages 79–96. Elsevier, 2008.

[54] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Aug 2008.

[55] Harvey Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning: Special Issue on Operating System Verification*, 42(2–4):125–187, Apr 2009.

[56] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, Santa Fe, NM, USA, June 2005. USENIX.

[57] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *POPL '07*, pages 97–108. ACM, 2007.

[58] T. Tuerk. A separation logic framework in HOL. In César Muñoz Otmane Ait Mohamed and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, pages 116–122, 08 2008.

[59] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in c++. In *OOPSLA '06: Object oriented programming, systems, languages, and applications*. ACM Press, 2006.

[60] Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.

[61] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17:337–345, 1974.