Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 255

Systems Group, Department of Computer Science, ETH Zurich

Multiple Address Spaces in a Distributed Capability System

by

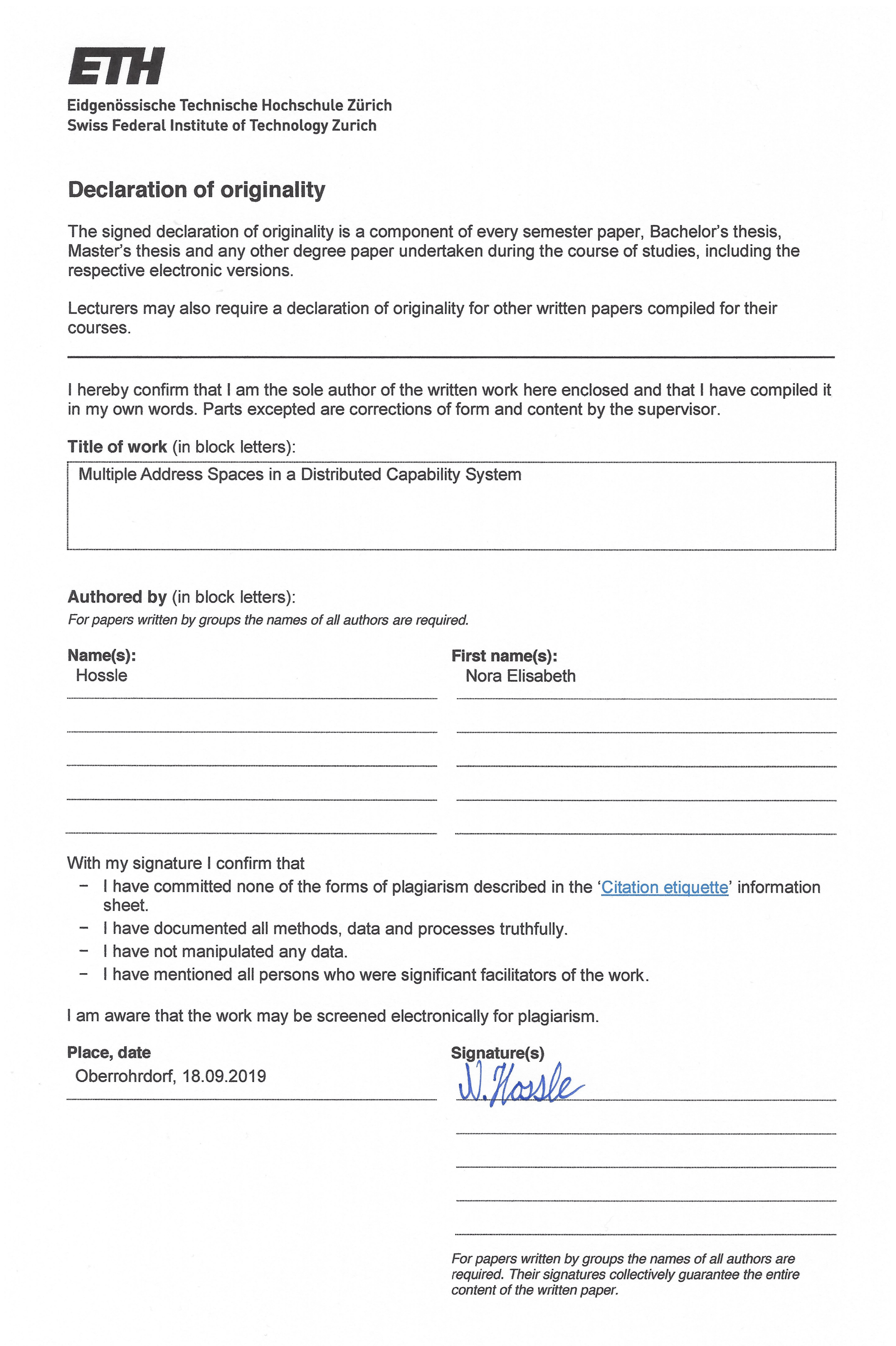
Nora Hossle

Supervised by

Prof. Timothy Roscoe

March 2019–September 2019

Informatik
Computer Science



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Multiple Address Spaces in a Distributed Capability System

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hossle	Nora Elisabeth

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Oberrohrdorf, 18.09.2019	N. Hossle

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Contents

1 Abstract **5**

2 Acknowledgements **6**

3 Introduction **7**

4 Background **9**
- 4.1 Baseline . . . 9
- 4.2 Access control and the access control matrix model . . . 10
- 4.3 Take and grant model . . . 11
  - 4.3.1 Rewriting rules . . . 11
- 4.4 seL4 . . . 12
  - 4.4.1 The seL4 capability system . . . 13
  - 4.4.2 Verification effort . . . 13
- 4.5 Barrelfish . . . 14
  - 4.5.1 Description of the Current Barrelfish Capability System . . . 14
- 4.6 Our paper . . . 15
- 4.7 Concluding remarks . . . 15

5 Model **16**
- 5.1 Introduction . . . 16
- 5.2 Many virtual address spaces but only one physical address space . . . 16
- 5.3 Sketching a new model with one eye on the decoding net model . . . 18
- 5.4 The address translation model as a take-grant model . . . 19
- 5.5 Access control subjects . . . 19
- 5.6 Choosing the "right" rights . . . 21
- 5.7 Representing state . . . 22
- 5.8 The model as a capability system . . . 22
  - 5.8.1 A formal description of capability operations . . . 24
- 5.9 Terminology . . . 27
- 5.10 Finally, the formal model . . . 28
- 5.11 Invariants . . . 30
- 5.12 Concluding remarks . . . 31

6 The executable Haskell specification **32**
- 6.1 Preliminary design decisions and design goals . . . 32
- 6.2 The specification's structure . . . 33
- 6.3 The specifications's data types . . . 34
- 6.4 The specification's state monad . . . 36
- 6.5 Invariants . . . 39
- 6.6 Conclusion . . . 40

7 Evaluation **41**
- 7.1 The paperTrace . . . 41
- 7.2 `updateLog` and `updateTrace` . . . 42
- 7.3 Managing dispatchers . . . 44
- 7.4 Modelling external events . . . 44
- 7.5 Limitations of the executable Haskell model . . . 46
- 7.6 Evaluation of the C implementation presented in [1] . . . 46

8 Conclusion **48**

9 Future work **49**

10 References **50**

List of Figures

1 Rough schematic overview of address translation 9
2 The access control matrix: A schematic example with the "classic" rights 10
3 The refinement layers in the verification of seL4 [12] 13
4 Visualisation of the old address translation model 17
5 A formalisation of the old model 17
6 Rough sketch of the multiple translations of an address from an arbitrary virtual address space to an arbitrary physical address space, VAS = virtual address space, PAS = physical address space 18
7 Definition of the nodes of the decoding net graph, taken from [1] 19
8 Take-grant model, start 20
9 Take-grant model, after applying the rules 20
10 SAC (Secure Access Control) authority (except self-authority) from [19] 21
11 Mapping, authorities and order: $G(Map) = Grant(Map(obj))$, $G(Access) = Grant(Access(obj))$ 22
12 The different object types, connected by valid retyping paths, inspired by [8] 23
13 Capability bookkeeping 24
14 The new model defined formally, part 1 29
15 The new model defined formally, part 2 30
16 Stack of abstraction 32
17 File structure of the executable specification 33
18 Code snippet from *BasicTypes.hs*, defining the basic address types 34
19 Code snippet from *BasicTypes.hs*, defining the authority data type 35
20 Code snippet from *BasicTypes.hs*, defining the object data type 35
21 Code snippet from *CapabilityType.hs*, defining the capability data type 35
22 Code snippet from *DispatcherType.hs*, defining the dispatcher data type 36
23 Code snippet from *KernelStateType.hs*, defining the KernelState data type 36
24 Venn diagram depicting the set of traces T containing the set of correct traces CT 36
25 Visualization of the state automaton underlying the Haskell specification, IS = Initial state, KSX = x-th KernelState, InvS = InvalidState, a.d.o. = any disallowed operation, s.a.o. = some allowed operation 37
26 Code snippet from *CapabilityOperationType.hs*, defining the data type CapabilityOperation . 37
27 Code snippet from *CapabilityOperationType.hs*, defining getters and setters 38
28 Code snippet from *CapabilityOperationType.hs*, defining the runner function 39
29 Example trace function called paperTrace 42
30 Log output of the paperTrace function, final KernelState, part I 43
31 Log output of the paperTrace function, final KernelState, part II 44
32 Code snippet showing the signatures of updateLog and updateTrace 45
33 Code snippet showing how dispatchers can be managed 45
34 Code snippet showing how external events can be managed 46

1 Abstract

Once upon a time each computing system had a single physical address space comprised of an ordered set of physical addresses. These addresses were unique over the whole system and every component of the system (core, MMU et cetera) agreed upon their meaning ...

This old and much loved model of computing systems is not only outdated and grossly oversimplifies current systems, indeed it may never have been appropriate at all. It is widely agreed upon that today's systems are much more heterogeneous and complex than they used to be. System programmers are dealing with whole networks of cores, different memory blocks, interconnects and a diverse set of other devices. It is entirely possible that resources are rendered at different physical addresses for different cores.

The purpose of this master thesis is to update the old model by introducing a network of address spaces (building on [11], [2]) with potentially more than one physical address space. It also aims to bring multiple address space support to the Barrelfish capability system. This is to be done by formally modelling the access rights of the different agents with respect to memory management. The new model gives rise to a Haskell model in the spirit of seL4's executable specification [12].

2 Acknowledgements

I would like to thank foremost my supervisors Prof. Dr. Timothy Roscoe and Reto Achermann for their valuable guidance and support. I also thank several other members of ETH's Systems Group (Dr. David Cock, Lukas Humbel and Daniel Schwyn) for their input and interest in my work.

3 Introduction

Modern computing systems' hardware is getting increasingly complex and heterogeneous. We run our software on machines with multiple cores (e.g. I'm writing this thesis on a machine with 4 physical and 8 logical cores). In our pockets we carry phones with not only multiple cores but heterogeneous cores and such things as "machine learning hardware acceleration"[21]. IOMMUs, multiple MMUs and (R)DMA are wide spread, virtual to physical address translation often relies on a multi-level hierarchy of page tables and virtualisation of whole or partial operating systems is becoming continuously more important. There is even research being performed to enable things such as compute bitwise operation directly in DRAM[18] or fit systems with *persistent* RAM (e.g. phase change memory[14]).

The multiplication of complexity and heterogeneity in hardware is a problem because it makes writing the code configuring and managing it much more difficult, time consuming and cumbersome. For example, we have to deal with the problem that the cores of the system are no longer guaranteed to share a uniform view on each other, let alone the diverse, potentially DMA capable devices of the system. Depending on the wildly varying guarantees given by the hardware the different cores may see updates to certain memory locations in different orders or there might be memory that only a certain secure coprocessor can access. It has also become much more difficult to write code (correctly) configuring the MMU(s), controlling (R)DMA etc. And code that is difficult to write is, of course, more error prone.

At least part of the difficulty in writing code managing address translation comes from the fact that the mental model in the heads of the programmers in no way reflects the complexity present in the hardware of today. The commonly used address translation model massively oversimplifies the state of things and abstracts away much more details than is useful. Unfortunately, being so simple to understand and used as a basic abstraction in Linux, makes the old model also very popular and hard to get rid of.

I will describe the old model in more detail in section 5 but for now, let's put it like this: Almost every agent of the system has its own virtual address space which is mapped by address translation, almost magically, to a very finite physical address space which is usually much smaller than all of the virtual address spaces combined, global and shared between all cores. Physical address can serve as unique and globally valid identifiers of resources throughout the system.

It is easy to see that this simple model falls short of capturing the complexity of multiple MMUs, IOMMUs, (R)DMA, virtual machines etc. This remains not without consequences: Incorrect assumptions about the memory management systems lead to bugs and bugs concerning this critical functionality often lead to security vulnerabilities. [1]

Clearly it is necessary to replace the "trusty old" address translation and memory management model with a new one that can govern the complexity and heterogeneity present in today's state of the art systems. One important addition will be the addition and support of *multiple* physical address spaces. I aim to develop such a new address translation model, formally define it and then bring support for it to barrelfish's capability system [7] by writing an executable specification in Haskell (inspired by seL4's executable specification [12]).

In the last part of this introductory section, I will now quickly provide the reader with a "roadmap" to the remainder of this thesis:

Section 4 is the background section. It lays the foundation of this thesis by talking about and familiarising the reader with the related work. Its aim is to make sure that the reader has understood some basic points concerning the memory management functionality of operating systems and to give an overview over some related work before we dive in. Some of the most important background topics are the take-grant model[15], [5], seL4[12], [6], [19], decodingNets[11], [2] and Barrelfish[7].

Section 5 is the model section. It contains the theoretical meat of this thesis. Its aim is to walk the reader through the development process of a new address translation model. First we investigate what's wrong with the old address translation model that everyone's already familiar with, then some key requirements for a new address translation model are identified. Finally, it formalises the newly developed model with the help of set notation.

In section 6 we discuss the executable Haskell specification developed from the new model defined in section 5. The Haskell implementation's modular structure is analysed and its most important data types are introduced to the reader with the help of some commented code extracts. The state monad that is central to the executable specifications's design is shown and design decisions made during development are explained.

The section 7 is the evaluation section. It gives several examples that illustrate how the executable Haskell specification can be used. It also familiarises the reader with the logging and tracing subsystem of the Haskell implementation: How they are implemented and how they can be used for debugging.

After the evaluation section I conclude my results and insights in the section 8 titled "Conclusion". Potential future work (mainly on the executable Haskell specification) is discussed in the section 9 called "Future Work". Finally, in section 10 all references are listed.

4 Background

This section will first outline a few basic concepts concerning address translation and address management. The aim is that it can serve the reader as a starting point before diving into the discussion of the related work. Several different topics will be touched upon: First traditional access control concepts such as the access control matrix will be reviewed briefly (mainly because it is such a ubiquitous and well established model for the protection state of a system). Then I will introduce the reader to take and grant models (which will come up again in section 5) before coming to the seL4 microkernel and its capability system [12][6][19]. Several part of the work done in the context of research on seL4 heavily influenced this thesis.

After discussing seL4 I will provide some background information concerning the Barrelfish operating system[7] which is itself heavily influenced by seL4. There is also a short subsection concerning a paper that I co-authored and that discusses some of the work that this thesis centres on. (Naturally, this means that there is some overlap between the two.)

4.1 Baseline

No computing system can provide its users with meaningful functionality without the ability to store data for later use. Since storing data only makes sense when one is also able to later retrieve it, each computing system needs addresses to label its data. The size of addresses has grown over the last decades but the basics have not changed much since the introduction of the idea of the virtual address space: We have some sort of agent abstraction (e.g. thread, process or dispatcher depending on the concrete operating system's terminology) that deals in virtual addresses only meaningful in its very own virtual address space. It is isolated from the virtual address spaces of all other agent's running on the same system concurrently. The operating system maintains the illusion that our agent is the only agent running and using the system's resources. This includes the illusion that all memory of the system belongs to our particular agent.

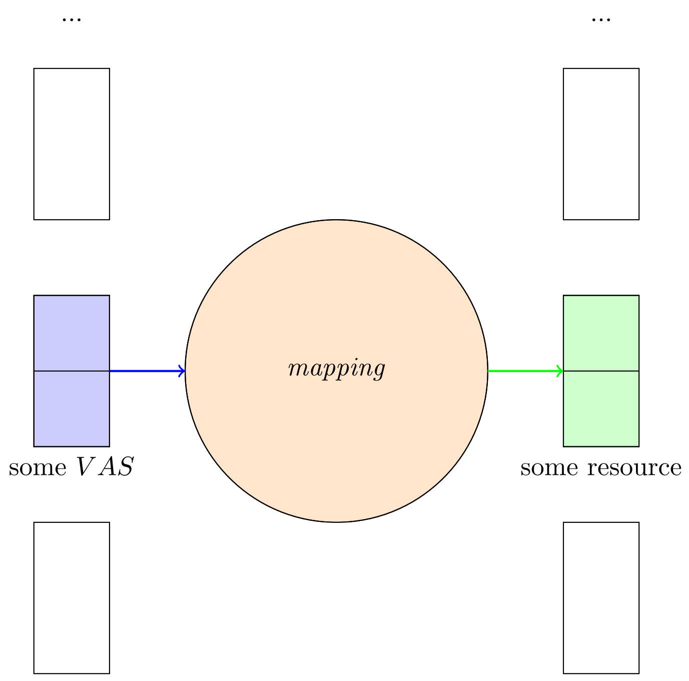


Figure 1: Rough schematic overview of address translation

Behind the scenes the operating system is running the show: It translates the virtual addresses issued by our agent into physical addresses referring to the system's resources (e.g. RAM, memory mapped devices), see figure 1. The assignment of virtual addresses to physical resources is normally referred to as *mapping* (and *unmapping*, e.g. if a page is swapped out to disk). It is done completely transparently (from the perspective of the agents) but is essential for both the correctness and the security of the system.

It is well known that control of the mapping of virtual addresses to physical resources is practically equivalent with control of the whole system. This means that all operations changing the mappings present in the system have potential security implications and need to be performed by trusted agents only. Which is of course easier said than done ...

4.2 Access control and the access control matrix model

When it comes to access control in computer systems, it is quite ubiquitous: The access control matrix as a model for the protection state of the system [13]. The access control matrix model can be summed up as "describing the *privileges* of *subjects* on *objects*" [4]. This information is then "stored" in matrix form.

Note that actually storing the matrix in memory is inefficient. Not only would it consume quite a lot of bytes as its space requirements are in $O(m * n)$, m being the number of subjects in the system, n the number of objects currently present in the system, it would also have to be modified often. Each time a new object is created or a new subject is spawned, a column respectively a row would have to be added to the matrix. No matter if we stored the matrix in row-major or column-major format, either adding a subject or an object could potentially require us to copy the whole matrix.

To mitigate this, a lot of operating systems either project column- or row-wise, storing the rights each subject has with the object itself (e.g. Linux does a version of this) or storing the whole corresponding row of the matrix with the subject. The first solution is called ACL (Access Control Lists), the second is called capability based security where unforgeable tokens conveying authority over objects are stored somewhere with the subjects. [20]

	$object_0$	$object_1$	...	$object_n$
$subject_0$	$\{r\}$	$\{r, w, x\}$	...	$\{\}$
$subject_1$	$\{r, w\}$	$\{\}$	...	$\{r\}$
...	...	...	...	...
$subject_n$	$\{r, x\}$	$\{r, w, x\}$	...	$\{r, w, x\}$

Figure 2: The access control matrix: A schematic example with the "classic" rights

See figure 2 for a schematic example of an access control matrix. (In the example, the "classic" three rights read, write and execute are used but of course there are different sets of rights possible. The matrix can also be transposed.)

More formally, the access control matrix model represents the protection state of a system (relative to a set of privileges P) as the triple (S, O, M). S is the set of subjects, O the set of objects, M can either be a relation of the form $S \times O \times P$ determining the matrix or a function $S \times O \to \mathcal{P}(P)$ [4].

The focus of this access control model clearly lies on *authorisation* (as a security concept), not on how to enforce access control. The idea is that a reference monitor can act as an enforcer, either granting the request for performing an operation a subject has submitted or denying it. [4].

4.3 Take and grant model

“Vague or informal arguments are unacceptable since they are often wrong.”[15]

The take and grant model is an abstract formulation of a protection system [5]. It allows to express security properties as reachability questions in a graph modelling the system and thus to treat them formally. This section consists of a short introduction to the take and grant model as it is described in [15]:
With a take and grant model, we are given a directed and labelled graph G as well as a set of rewriting rules R. The graph is loop-free and finite, each edge is labelled with a subset of $\{r, w, c\}$. (If there is an edge from vertex p to q, labelled r (or w or c), this can be interpreted as “user p is able to read (respectively write or call) object q”. Note that all vertices can be both subject and object depending on the context, the take and grant model does not differentiate between the two.)
The goal is to determine if there exists a sequence of graphs $G_1, G_2, \ldots, G_n$ such that $G_1 = G$, G_{i+1} follows from G_i by applying some rule in R and for G_n some property X holds (X being some sort of protection violation). Note that property X is of the form “there is an edge from vertex p to vertex q with the label α”.
If one of the rules $\in R$ is applied to a protection graph G, yielding a new protection graph G’, we will write this as $G \vdash G'$. $G \vdash^* G'$ denotes the reflexive, transitive closure of the relation on protection graphs.

4.3.1 Rewriting rules

There exist five different rewriting rules (taken from [15], slightly rephrased):

Take Let x, y, and z be three distinct vertices in a protection graph G. Let there further exist an edge from vertex x to vertex y with some arbitrary label γ such that $r \in \gamma$ holds. (Notice that in all diagrams in this section, we simply label the edge with the right that must be contained in the set to apply the rule (for increased clarity).) There must also be an edge from y to z with label $\alpha \subseteq \{r, w, c\}$. The take rule allows one to add the edge from x to z with label α, yielding a new graph G'. Intuitively x takes the ability to do α to z from y. We represent this rule by:

$$\begin{tikzcd}
x \arrow[r, "\text{r}"] & y \arrow[r, "\alpha"] & z & \Longrightarrow & x \arrow[r, "\text{r}"] \arrow[rr, bend left, "\alpha"] & y \arrow[r, "\alpha"] & z
\end{tikzcd}$$

Grant Let x, y, and z again be distinct vertices in a protection graph G, and let there be an edge from x to y, labelled γ such that $w \in \gamma$, and let there also be an edge from x to z with label $\alpha \in \{r, w, c\}$. The grant rule allows one to add an edge from y to z with label α, yielding a new graph G'. Intuitively x grants y the ability to do α to z. In our representation:

$$\begin{tikzcd}
x \arrow[r, "\text{w}"] \arrow[rr, bend left, "\alpha"] & y & z & \Longrightarrow & x \arrow[r, "\text{w}"] \arrow[rr, bend left, "\alpha"] & y \arrow[r, "\alpha"] & z
\end{tikzcd}$$

Create Let x be any vertex in a protection graph G. **Create** allows one to add a *new* vertex n and an edge from x to n with label $\{r, w, c\}$, yielding a new graph G'. Intuitively speaking, x creates a new user that it can read, write, and call. In our representation:

$$\begin{tikzcd}
x & \Longrightarrow & x \arrow[r, "\{r,w,c\}"] & n
\end{tikzcd}$$

Call Let x, y, and z be distinct vertices in a protection graph G, and let $\alpha \subseteq \{r, w, c\}$ be the label on an edge from vertex x to vertex y and γ the label on an edge from x to z such that $c \in \gamma$. The **call** rule allows one to add a new vertex n, an edge from n to y with label α, and an edge from n to z with label r, yielding a new graph G'. Intuitively x is calling a program z and passing parameters y. The "process" is created to effect the call: n can read the program z and can α the parameters. In our representation:

$$\begin{tikzcd}
 & & & & & x \arrow[dl, "\alpha"'] \arrow[dr, "c"] & \\
 & x \arrow[dl, "\alpha"'] \arrow[dr, "c"] & & & y & & z \\
y & & z & \Longrightarrow & & n \arrow[ul, "\alpha"] \arrow[ur, "r"'] &
\end{tikzcd}$$

Remove Let x and y be distinct vertices in a protection graph G with an edge from x to y with label α. The **remove** rule allows one to remove the arc from x to y, yielding a new graph G'. Intuitively x removes its rights to y. In our representation:

$$\begin{tikzcd}
x \arrow[r, "\alpha"] & y & \Longrightarrow & x & & y
\end{tikzcd}$$

4.4 seL4

This subsection mainly concerns the following work: [12], [6], [19]. These are three papers published about the development and verification of seL4, a microkernel.

seL4 is a third-generation microkernel descendant from L4 which is a kernel series for embedded devices where security and competitive performance where main design goals. The seL4 (secure, embedded L4 [6]) microkernel is written in C and assembler. According to its creators, its performance is "comparable to other high-performance L4 kernels" [12].

The access control system of seL4 is capability based. seL4 also contains abstractions for virtual address spaces, inter-process communication (IPC) and threads. However, the structure of virtual address spaces is not defined by the kernel itself. Instead that responsibility is delegated to so called pager-threads. Just as in the Barrelfish OS (which will be discussed next) and Linux there is just one global physical address space. The IPC system is tightly integrated with the capability system, e.g. making use of a capability called *reply capability.*

seL4 was initially developed to run on an ARMv6-based platform, the verification work done also concerns this same platform.

4.4.1 The seL4 capability system

seL4's memory management system is implemented by making use of an abstraction called *capabilities*. These control access to physical memory [6]. Both so called *untyped* physical memory and so called *typed* kernel objects are represented by capabilities. (For now think of capabilities simply as unforgeable "keys" or handles that grant the holder authority over objects.) *untyped* capabilities can be *retyped* into *typed* capabilities to convert blocks of previously unused RAM into objects the kernel needs (e.g. page tables, thread control blocks etc.). [12]

All capabilities (as they are themselves memory objects and therefore need to be stored somewhere) are located on special kernel objects called *CNodes*. They are not accessible from user space and access is controlled with help of other capabilities representing the CNodes themselves. All CNodes together form the *CSpace*, a set of linked CNodes. There exists one CSpace per thread and it contains all capabilities that the thread can call its own. In fact, the unit of execution of seL4 are threads and each has its own TCB (thread control block), which is also where a thread's capabilities for the CSpace root (and the virtual address space root) are stored. [6]

Literally everything in seL4 is controlled via the use of capabilities. Any system call is a capability invocation, services and kernel abstractions are provided with the help of named kernel objects that agents are granted authority over by capabilities. In fact, with seL4 all memory (page tables, memory merely used by an application, you name it) is accounted for by capabilities. [6]

4.4.2 Verification effort

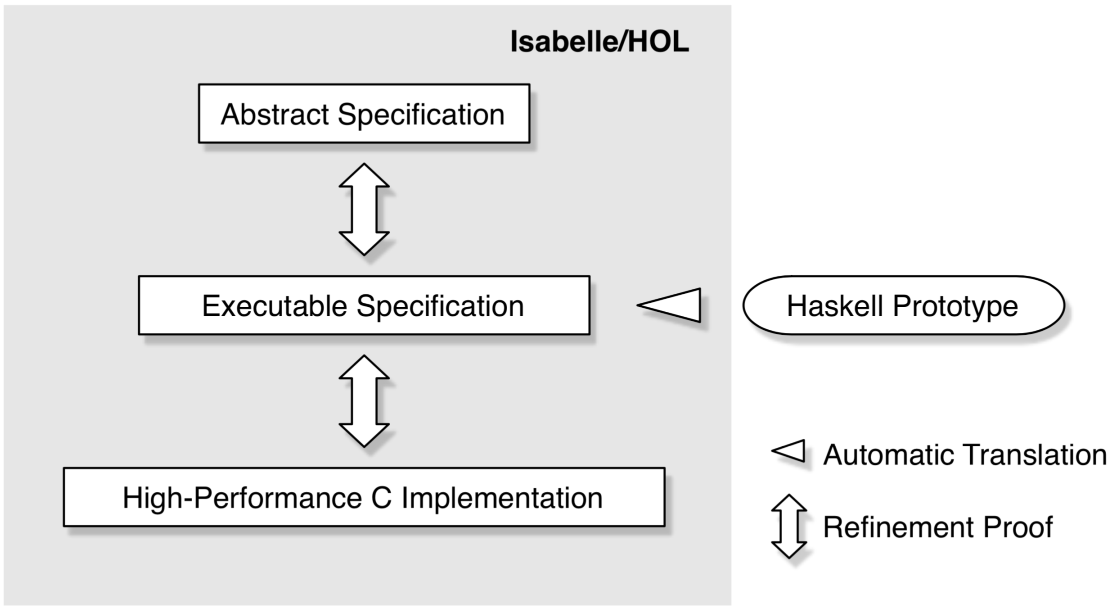


Figure 3: The refinement layers in the verification of seL4 [12]

seL4 is the first general-purpose kernel which's functional correctness has been formally proven. The verification of the seL4 microkernel was not only formal but has been machine-checked to be correct. (Functional correctness should be taken to mean that the actual implementation of the kernel conforms to the abstract specification. That specification (see the top layer of figure 3) contains design goals such as "the kernel will never crash" or "it will never perform an unsafe operation". [12]

The verification effort on seL4 made heavy use of the theorem prover Isabelle/HOL[17]. Verification was done with a technique the authors of [12] call “interactive, machine-assisted and machine-checked proof”. A big part of the proof was to show that each of the two lower layers in figure 3 refines the upper layer (this is called a refinement proof). If it is proven that one layer refines an upper layer and it is proven for that upper layer that a certain property holds, it is guaranteed that this property also holds for the lower layer. [12]

Note that in addition to the three layers there is also a “Haskell prototype” denoted on figure 3. This prototype is more abstract than the actual high-performance implementation (in C) and it is translated automatically into an executable specification that is then fed into the theorem prover. The executable specification contains all data structures and implementation details that will ultimately be present in the C source code. The “intermediate layer of Haskell” was inserted between the abstract specification and the high performance C implementation because it was accessible to OS designers but was still abstract enough to be translated into something that could be given to the theorem prover. [12]

seL4’s executable Haskell specification was a major inspiration for the implementation presented further on in section 6.

4.5 Barrelfish

Barrelfish OS [7] is a research operating system developed by ETH’s Systems Group. It is targeted to multi-core systems. It’s process abstraction is named *dispatcher* and a dispatcher can potentially contain multiple threads. While dispatchers are core local, multiple dispatchers running on different cores are said to form a *domain.*

The Barrelfish OS has an seL4-style capability system (including CNodes that form CSpaces) that it uses for both authorisation and resource management. It has better support for heterogeneous systems than seL4, also it organises the CSpaces of its processes, called dispatchers, differently. However, just like seL4 it uses the old addressing model that boasts only a single physical address space. [1]

4.5.1 Description of the Current Barrelfish Capability System

To my knowledge, as of yet no complete description of barrelfish’s capability system has been published. However, the interested reader may consult the official website (includes the corresponding documentation in the form of technotes) [7] or [9].

The capability system of Barrelfish is strongly inspired by the use of capabilities of seL4. Capabilities are typed to restrict the usage of the memory objects they refer to and the different capability types may be changed (retyped) according to clearly defined retyping rules. If a dispatcher stores a certain capability in its CSpace it may make use of the rights the capability confers to it by invoking the reference monitor API (implemented as a system call interface). [1]

In contrast to seL4’s capability system Barrelfish’s capability system is *distributed* over the different cores of the multi-core system it was designed for. Also, Barrelfish organises the CSpaces of its dispatcher’s in such a way that it can perform more efficient lookups concerning the capabilities, their ancestors, descendants and copies. It can also perform range queries. This is possible because the capabilities are organised as a balanced tree structure.

Another feature differentiating Barrelfish from seL4 are its *MappingCaps*: In Barrelfish the mapping of one resource to another (both represented by capabilities) causes a MappingCap to be created. This MappingCap embodies the installed mapping and is a descendant of the capability representing the resource that has been *mapped to.* The MappingCap can be used to *revoke* (remove) the installed mapping.

Remark: Barrelfish uses a DSL (domain specific language) to define its capability types called *Hamlet.* This makes it possible to generate certain C or assembly functions performing repetitive but error prone tasks such as accessing certain bits in bitfields automatically.

4.6 Our paper

Part of the work presented in this thesis is also mentioned/ discussed in an as of yet unpublished paper [1]. It has been submitted to a conference and was being reviewed at the time this thesis was completed.

In this paper we present a new least-privilege-based model of addressing which is partly based on the model presented in this thesis. The paper discusses the executable specification I wrote in Haskell (see section 6). It then shows and evaluates a fully functional implementation of the model written in C. That implementation extends the capability system of Barrelfish [7]. [1]

I quickly discuss the C implementation of the new model and its evaluation at the end of section 7.

4.7 Concluding remarks

We have now seen the traditional access control matrix model [13] that models the system's protection state as a matrix, the take-grant model [5], which is an abstract formulation of a protection system, and the two operating systems seL4 [12] and Barrelfish OS [7]. While the access control matrix model and the take-grant model are both quite abstract, making no mention of address spaces at all and relying on such things as references monitors to enforce their policies, seL4 and Barrelfish - while more concrete - both support only a single physical address space.

5 Model

This section is all about modelling. I will start with describing the address translation/ address space model that is commonly used and point out in detail where it falls short. I will then, step by step, identify how the old model could be improved upon, discuss how rights and state can be modelled and with the help of these considerations develop a new address translation/ address space model. Finally, I will formalise the new model. I will also state a number of invariants.

5.1 Introduction

"But fundamentally, computer science is a science of *abstraction* - creating the right model for thinking about a problem and devising the appropriate mechanizable techniques to solve it." [3]

Choosing the *right* model is often the first and (almost just as often) the most important step for solving a problem. If the model chosen is simply *wrong* (e.g. the problem to solve is not within the model's scope), solving a problem becomes much harder, if not impossible. However, a model can be much more than being altogether *right* or *wrong.* It can also either oversimplify a problem or, on the contrary, not simplify it enough, retaining an inappropriate level of complexity.

We have already established in the previous sections that the predominantly used model for address translation errs on the side of oversimplification. It follows that the model needs to be extended with more details so that the system modelled can be represented with more accuracy, but how exactly is this to be best accomplished? To be able to reason properly about address translation and the mapping of whole address ranges across a network of different address spaces, it is vital to establish a model that is not only correct but also specific and unambiguous. This means we need to specify our model as formally as possible. The final model needs to describe in clearly defined terms how an address that is part of some agent's virtual address space is translated (potentially multiple times) before it is resolved to an address identifying an actual physical resource (see figure 6 for a bird's eye's view).

In the following subsections of this chapter I will first point out the defining features of the old address translation model as already promised, then I will sketch a new abstraction for memory management and addressing. I will also discuss the strengths and limitations of the decoding net model [11][2]. After that I will discuss "choosing" the appropriate rights for our new model (by first framing the new model as a traditional take-grant model) and representing state. Finally, I will discuss the new model as a capability model and state some invariants in plain English before formalising the new address translation model with set notation.

5.2 Many virtual address spaces but only one physical address space

The old address translation and memory management model centres on the following thought: *There may be many virtual address spaces but there is only one physical address space, it is global and its addresses serve as unique identifiers of physical resources.* (See figure 4 for a visualisation of this). Each agent in the system has its own virtual address space which is conventionally defined by a contiguous array of addresses (e.g. in a 32-bit addresses, byte-addressable system the virtual address space will be 2^{32} bytes = 4GB big). However, all agents not only share the same physical address space but also *perceive* it the *same way.* They see the same resources located at the same physical addresses. This means that when thinking about address translation in terms of the old address translation abstraction there is no need to qualify physical addresses in any way. After all, the assumption is that it can ever happen that we have to deal with anything as ugly as aliasing with physical addresses.

See figure 5 for a formalisation of the old address translation model. Note that *resolve* is only a partial function since some virtual addresses might not be mapped right now. Accessing them may instead give rise to a page fault (or a memory protection fault if the accessing agent lacks the necessary rights to access a specific location).

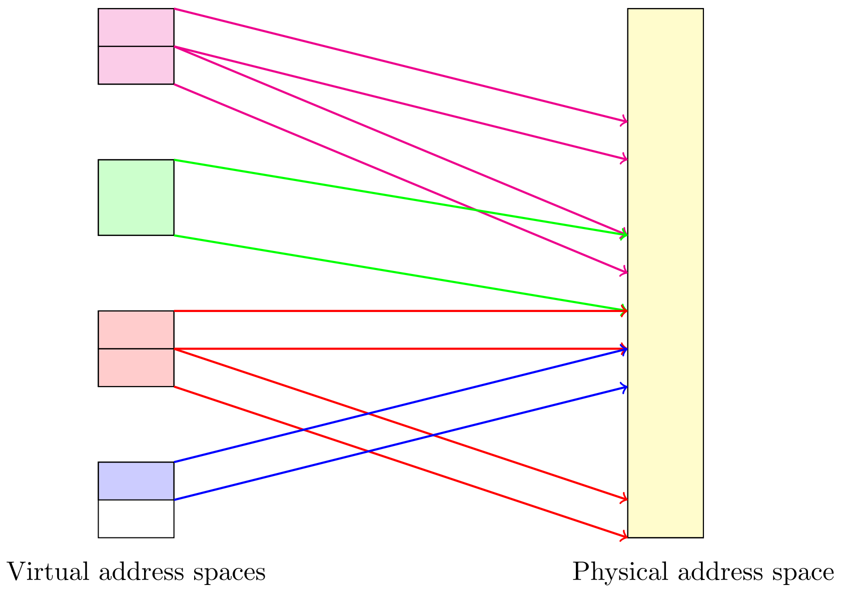


Figure 4: Visualisation of the old address translation model

The old operating system's memory management and addressing model is mostly inspired by the need to map the potentially many different virtual address spaces of the system's agents onto the finite physical resources of the system. On the virtual side of things, the model is quite practical and intuitive. However, on the physical side, *all* resources are represented by a single contiguous array of memory cells. Often simply called "the main memory", all memory-like resources of the system are lumped together without any regard to, say, varying access latencies (e.g. caused by caching, NUMA effects). We can perhaps sum it up best with a nice German saying that roughly translates to: "Throwing everything into the same pot."

One example of a security problem arising from flattening all memory resources of the system to one contiguous, uniform array would the phenomenon called memory-performance hogs (MPH). This is a phenomenon where an application manages to execute what is de facto a denial of service attack by hogging all memory bandwidth, starving all other applications running on the same hardware. [16]

$$\begin{aligned}
physBits &= \text{number of bits in physical addresses}\\
virtBits &= \text{number of bits in virtual addresses}\\
agents \subset \mathbb{N},\ physicalAddresses &= [0, \dots, 2^{physBits} - 1]\\
virtualAddresses &= agents \times [0, \dots, 2^{virtBits} - 1]\\
resolve : virtualAddresses &\rightharpoonup physicalAddresses
\end{aligned}$$

Figure 5: A formalisation of the old model

It is important to keep in mind that the "trusty old" address translation model was conceived at a time when most processors were single core processors. Because of this, it has no support whatsoever for systems where once core's view of the physical resources present in the system might not match those of the other cores and DMA capable devices (e.g. because not all memory is visible from all cores, there are whole networks of interconnects present in the system). This shortcoming is reflected in figure 5 by the fact that the *resolve* function does not depend on which agent resolves the virtual address $\in$ *virtualAddresses* (the *agents* part of *virtualAddresses* merely denotes which virtual address space the address is part of).

5.3 Sketching a new model with one eye on the decoding net model

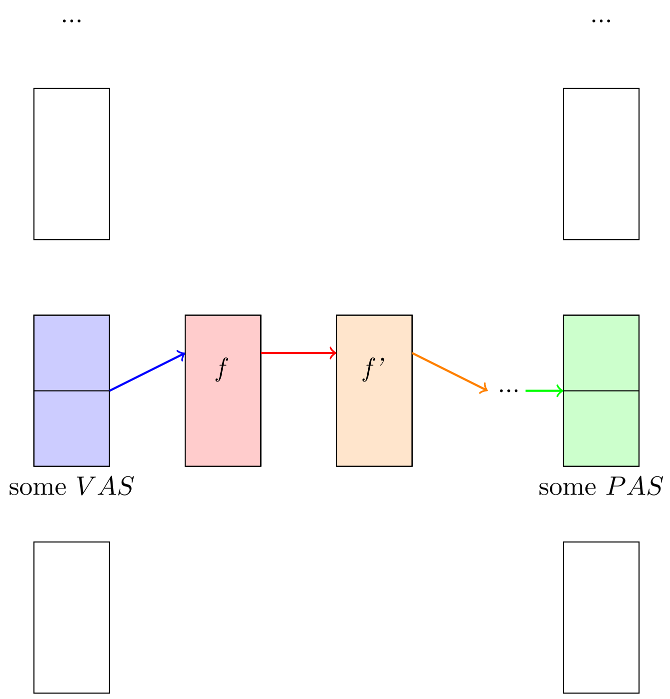


Figure 6: Rough sketch of the multiple translations of an address from an arbitrary virtual address space to an arbitrary physical address space, VAS = virtual address space, PAS = physical address space

So which parts of the established memory management abstraction commonly used for operating systems need to be changed? First we have to abolish the idea of a single, global physical address space. To do this we can build on the work presented in [11], [2]. In these papers *Achermann et al.* introduced a network of interconnected address spaces called a decoding net. They aim to formally specify the addressing behaviour of hardware. As a result the decoding net model is a much more elaborate model than 5 and capable of describing at an appropriate level of complexity even systems with high degrees of heterogeneity or interconnectivity.

The decoding net model centres on the idea of a *qualified name*. An address becomes a *name* when it is *qualified* by the identifier given to its *name space* or *address space*.[2]

A system is described by the decoding net model as a directed graph whose nodes are defined as given in figure 7. Note especially the distinction between *name* and *address*. A *name* is simply an *address* that has been qualified with a node identifier. This is necessary because with losing the abstraction of a single, global address space we've also lost the capability to identify resources uniquely by knowing their physical address. Any *name* can potentially be translated to any other *name*, even itself. (In fact, the decoding net model even allows a *name* to be translated to a whole set of *name*, thereby modelling indeterminism.) [2]

Think of the nodes in a decoding net as either virtual, intermediate or physical address spaces or devices. The edges of the decoding net graph denote that an address is translated from one address space to the corresponding address in another, new address space. This keeps up until an address is *accepted* by a node.

$$\begin{aligned} name &= \text{Name } nodeid\ address \\ node &= \text{Node } \mathbf{accept} :: address \\ &\qquad\quad \mathbf{translate} :: address \to \{name\} \end{aligned}$$

Figure 7: Definition of the nodes of the decoding net graph, taken from [1]

All nodes of a decoding net graph assign "incoming" addresses to two different sets. One set of addresses the node accepts, the other will be sent on to be translated further. Accepting an address means that the address denotes a concrete resource local to the node.

Note that we can identify all resources of the system uniquely by using their "last" address. This is the physical address of a resource qualified by the *nodeid* of the node of the decoding net that eventually accepts all addresses that denote our resource.

While the decoding net model is great as a *static* model it lacks any mechanism to model reconfiguration of the address translation structures of a system (e.g. updating a page table entry). This means that there is no way to change how any one address is translated which of course is a task central to the memory management functionality of any operating system.

The decoding net model logically also includes no concept of rights. (After all, having no state changes at all implies that there is no need to define which subjects would be allowed to change which objects in which ways.) This means for us that in addition to the concept of interconnected address spaces we also need to design a mechanism to update decoding networks as new page tables are installed, memory frames swapped out to disk etc.

This new mechanism needs to be implemented in a way such that subjects only perform the actions (updates to the decoding network) that they are "authorised" for (possess the necessary rights). Which types of rights these are we want to describe as explicitly as possible.

5.4 The address translation model as a take-grant model

To get a better grip on which types of rights we might want to include in our new model let's try to describe address translation by mapping address ranges with the help of a take and grant model. We start with the simplest case, where we only have to translate our virtual addresses once to get the corresponding final physical addresses. In our model, the graph $G = G_0$, we have three distinct nodes: An *agent*, a local virtual address range (*lvaddr_range*) and a physical address range (*paddr_range*) corresponding to some resource (e.g. a block of RAM). The *agent* has full rights ($\{r, w, c\}$) to both resources (they can read, write and execute any address in the two address ranges). Now the *agent* wants to map the local virtual address range to the physical address range (see 8).

After applying (one after the other) the rewrite rules four times (first create, then grant (twice) and at last take), we get the protection graph G_4 (see figure 9). This means, that mapping an address range to another address range can be seen as a sequence of rewriting rule applications. Removing a mapping can trivially be modelled as a sequence of applications of the remove rule.

The case where local virtual addresses are mapped directly to physical addresses can easily be generalised if we replace the local virtual address range with a intermediate address range (for all except the last step) and use the newly generated mapping node instead of a physical address range.

5.5 Access control subjects

The take-grant model turned out to be quite unwieldy. (Even in the simplest case the generated take-grant graph became quite convoluted.) One of the reasons for this is that the take-grant model does not differentiate between object and subject but most access control models do and thereby cut out some unnecessary model complexity. As we have seen, for the verification work done on the seL4 microkernel, threads served as

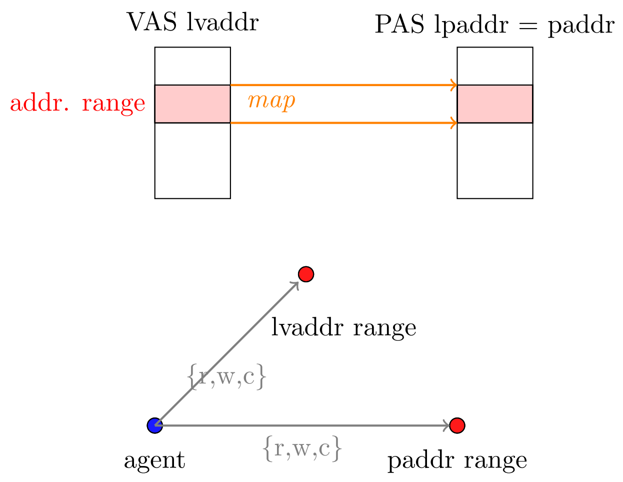


Figure 8: Take-grant model, start

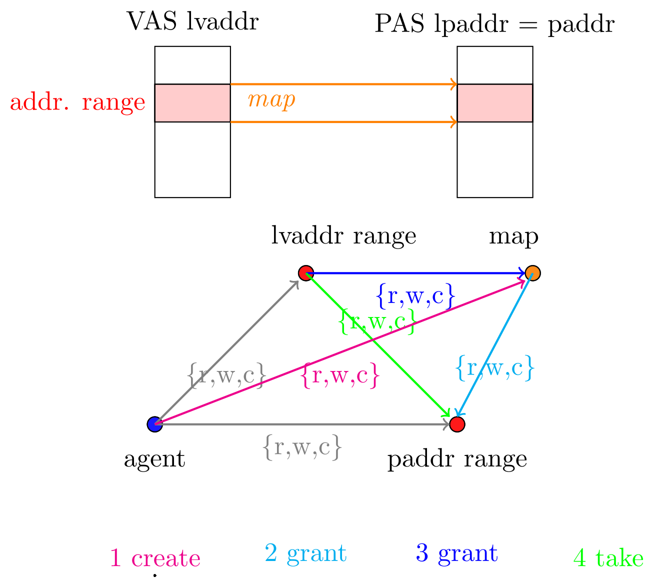


Figure 9: Take-grant model, after applying the rules

subjects [19]. Just as in Barrelfish [7], seL4 allows threads to share VSpace and CSpace. For our work here it seems sensible to choose either threads or dispatchers as they are used in Barrelfish [7] as our subjects.

Since one of the goals of this work is to build an addressing model applicable to Barrelfish, I chose dispatchers as my subjects.

5.6 Choosing the "right" rights

An important part of any access control model are the set of rights subjects can potentially hold with regard to the system's objects. The classic take-grant model only has three rights: Read, write and call. However, it is possible to define other rights, even sets of rights. For example, for verification of the seL4 microkernel different types of authority (sets of rights) were defined: "*Fig. [10] shows an abstract policy, only mildly simplified for presentation, that corresponds to a possible protection state of the SAC [(Secure Access Control)] at runtime. The objects in the system are grouped by labels according to the intention of the component architecture. [...] The communication endpoints between components have their own label to make the direction of information flow explicit. The edges in the figure are annotated with authority types [...].*"[19] This can be formally represented as:

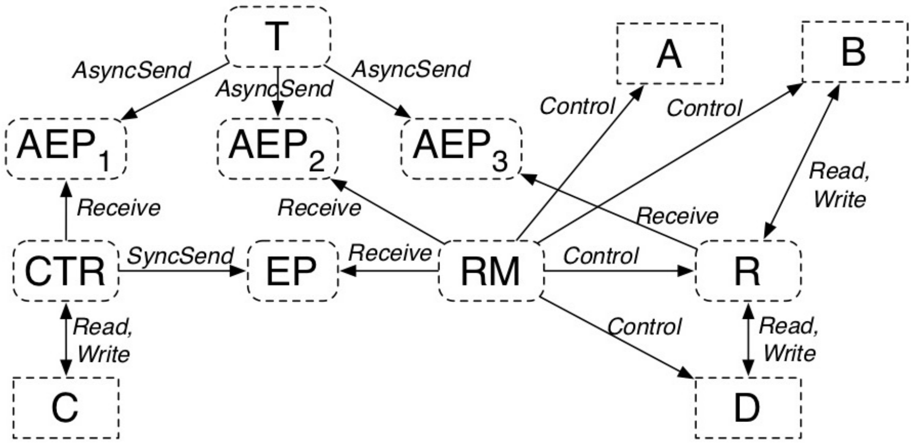


Figure 10: SAC (Secure Access Control) authority (except self-authority) from [19]

The different types of authority in seL4[19] - see also [10]

```
datatype auth =  Receive | SyncSend | AsyncSend | Reset | Grant
     | Write | Read | Control
```

By keeping in mind our special use case of managing memory we adapt this to:

```
// The most basic rights of the new address translation model.
right = Access | Map | Grant right
```

As it turns out, no more than these basic rights are needed [1]. *Read* and *Write* are fused together to the *Access* authority to keep the complexity low, sending and receiving is of no concern for us as we are only modelling the memory management. *Reset* also concerns sending and receiving. Instead we introduce a *Map* authority.

Now the *Access* authority refers to the set of rights necessary to be able to access a set of names, meaning a set of addresses (it does not necessarily have to be a contiguous range) qualified by the identifier of their address space. *Map* refers to the authority to map a set of addresses in one particular address space to another set of addresses in another particular address space for which we need to have the *Grant* authority. This authority can be thought of as having the authority to pass some other authority on to another agent

- or rescind it. We qualify it so that we either have a *Grant* authority for the *Map* authority or the *Access* authority (which are themselves qualified by a set of names). [1] puts it like this: *Grant* is "The right to insert *this* object into *some* address space" and *Map* is "The right to insert *some* object into *this* address space". Object being a memory object defined by a set of addresses and an address space identifier.

See figure 11 for an example of the minimum authorities needed to being able to install a chain of mappings over a number of different address spaces. Note the order in which the mappings m_1 to m_n are installed: m_1 is installed first, followed by m_2 and so forth. This order will later on be mandated by an invariant that forbids "dangling pointers" (a chain of mappings that leads nowhere, gets stuck).

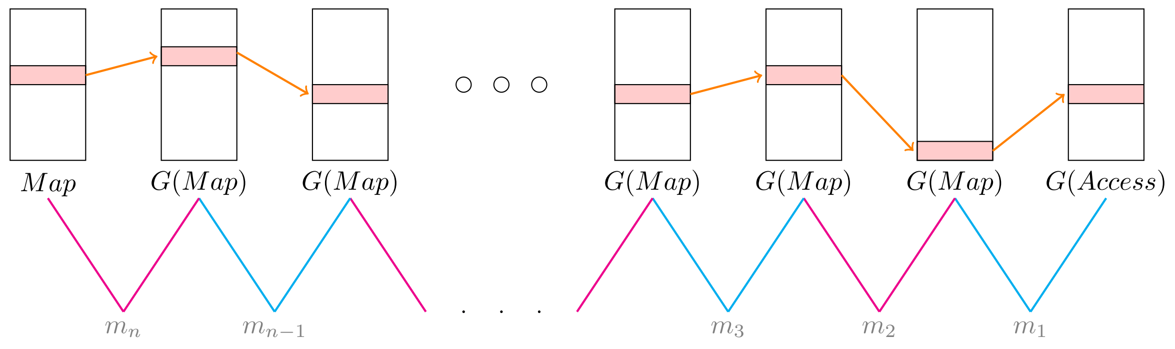


Figure 11: Mapping, authorities and order: $G(Map) = Grant(Map(obj))$, $G(Access) = Grant(Access(obj))$

5.7 Representing state

State can be represented in a number of different ways. seL4 uses kernel state for its verifications [19]. If we do the same for our system (we limit ourselves to state actually concerned with the translation of addresses from address space to address space, we are not interested in any other state), we have to keep track of all mappings installed in the system (which we will call the mapping database) as well as which agents have which authorities over which objects.

We also need to determine which types of (memory) objects our new model should have. For reasons of better applicability to Barrelfish, I choose object types similar to the capability types of the memory management system of Barrelfish [7]. (As already mentioned in the background section (section 4), barrelfish's memory management relies on capabilities to represent authority over memory objects [7].) See figure 12 for an overview of all different object types present in the new addressing model and which object types can be converted to which other object types. (Note that we refer to object type conversion as *retyping*, just as in Barrelfish.)

5.8 The model as a capability system

To make the new model more easily applicable to Barrelfish [7], I chose to represent authority over an object (defined as a set of names) as a *capability*. Here, a capability simply means "being capable of using a certain object in a certain way". This allows us to represent agents, or *dispatchers*, as their equivalent is called in

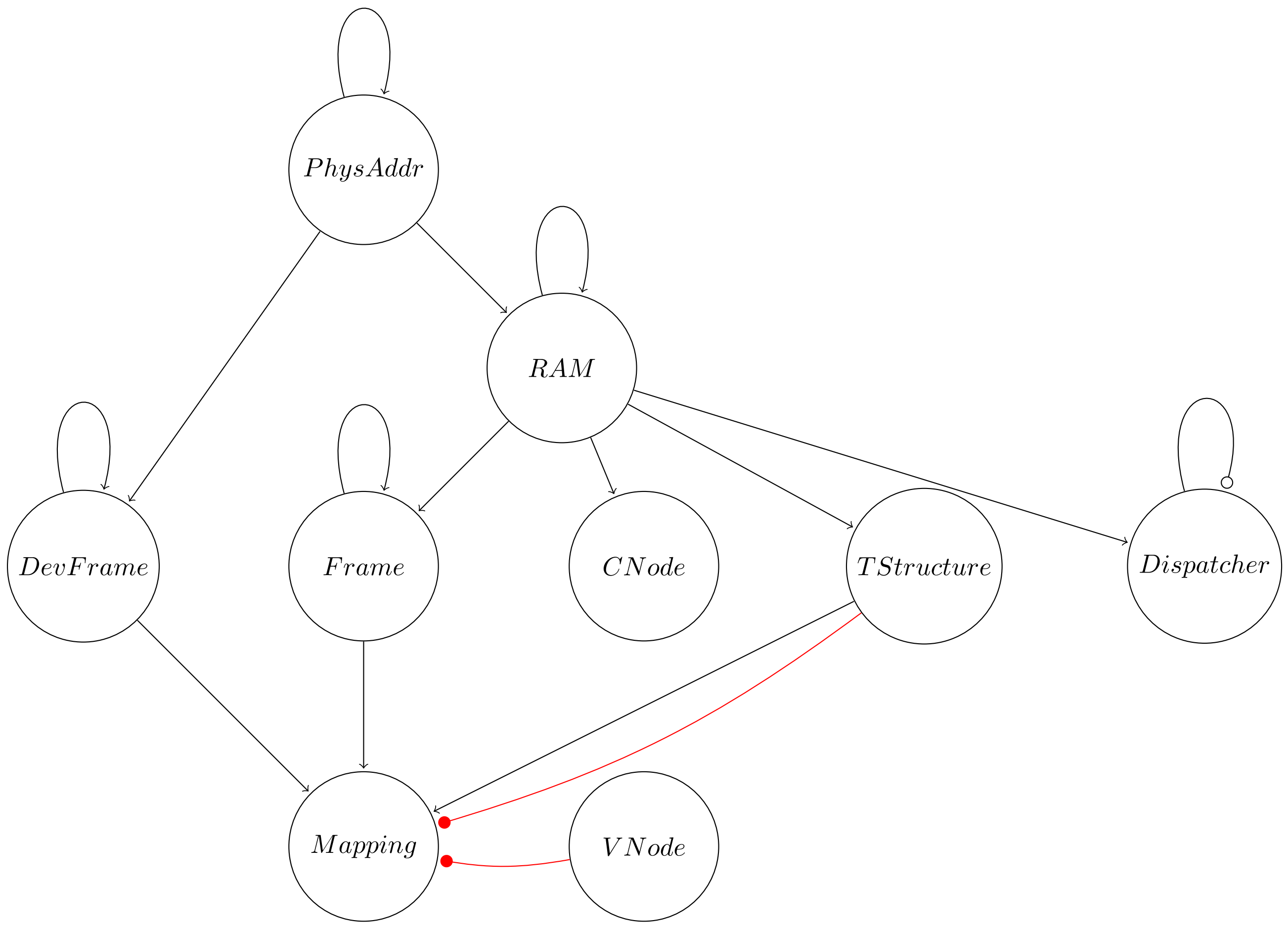


Figure 12: The different object types, connected by valid retyping paths, inspired by [8]

Barrelfish, as mere sets of capabilities. Those sets of capabilities will be tagged with a dispatcher identifier and an identifier for that dispatcher's virtual address space. The latter is necessary to know where in the decoding net address resolution should start.

In case of our new model, introducing capabilities simply means introducing an additional, slim layer of abstraction. We already have a concept of sets of rights (authority) an agent, now called a dispatcher, has on a specific resource or object. From now on we will simply call this construct a capability. This can be seen as a straightforward refinement of the new model (which before only had a concept of authority over a resource).

Note that capabilities also include some bookkeeping information: All capabilities that have arisen from being retyped from another capability, memorise the original capability as their ancestor. In addition, all capabilities that have arisen from being copied, retain a reference to the capability they are a copy of. See figure 13 for an illustration of this. This is the minimum of information we can store to always be able to compute all descendants and copies of a capability. (We define these terms the same way as they are defined in Barrelfish [7].)

For each object type the new addressing model has (see figure 12 there is a capability type, corresponding directly to the encapsulated object's type. If we have an authority over an object of type x, then a capability encapsulating this authority will be of capability type *xCap*. E.g. we have objects of type *Frame* and capabilities of type *FrameCap*.

The figure 12 not only shows us all object types and their retyping relations, it is also indicated which objects might appear "to the left" of a mapping operation. Let me explain this quickly: For each mapping operation we need an object being mapped and an object being mapped *to.* Inspired by Barrelfish[7], the *MappingCap* or mapping capability being created as a side-effect of this operation and indicating that a mapping is currently installed in the system is retyped from the object that is being mapped *to.* This we also call "being on the right side of a mapping operation" (consult figure 11). However, the newly created *MappingCap* capability will not be a descendant of the capability appearing to the "left" of the mapping operation, nor is there a retyping relation. An object type that can potentially be on the "left" side, is indicated in the figure with a red, special arrow ending at the *Mapping* object.

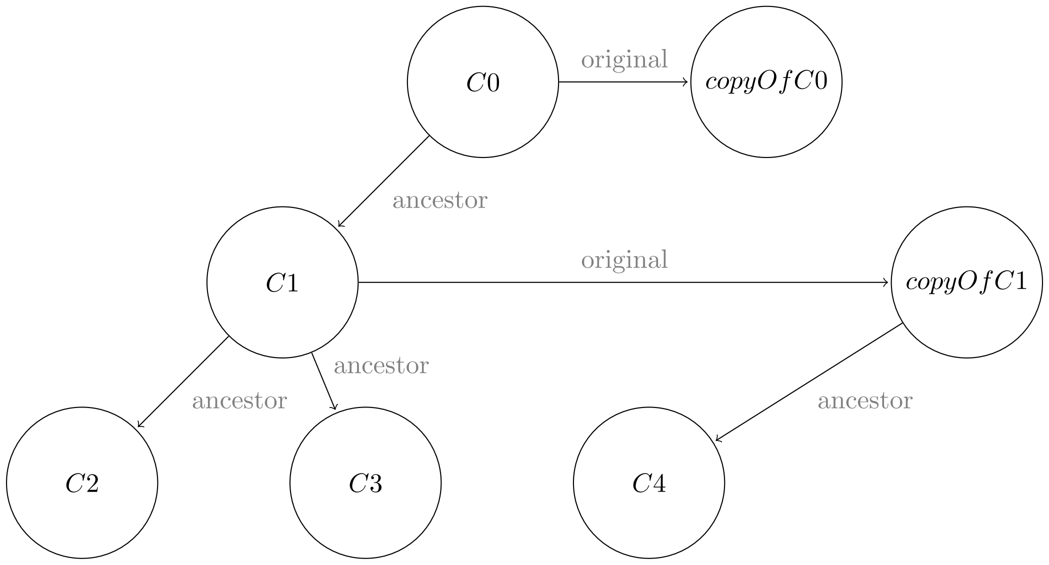


Figure 13: Capability bookkeeping

5.8.1 A formal description of capability operations

Any operation applied to our new addressing model can be expressed as an operation manipulating capabilities present in the kernel state. We will call these operations capability operations. This subsection will now describe formally the most important capability operations as they will be used in the new addressing model. Most definitions are inspired by definition of the corresponding capability operations in Barrelfish [7], which specified in [9]. The formal description of all capability operations serves as a specification for how these same operations should be implemented in a concrete system that is an instance of our new model. (Note that this is not necessarily identical to how these capability operations are actually implemented right now in Barrelfish.)

Copy The *copy* operation is described as: "*The copy operation must simply create a new copy in the target location, [...].*"[9] It is a quite straight forward operation, the only thing to keep in mind is that the bookkeeping part of the capability needs to be updated (omitted from the algorithm below) and that *MappingCap*s need to remain with the agent they were created for. (It is actually debatable if this is strictly necessary. Of course, any mapping from a virtual *VNodeCap* to some other capability only makes sense within the context of the corresponding virtual address space. However, other *MappingCaps* might be useful for other agents - if we accept that an agent only needs a MappingCap to prove it holds authority over a resource and that an agent might never have held any of the two original capabilities needed by the agent that created the *MappingCap.* This aside I chose to be conservative here and forbid the copying of *MappingCaps* across agents.)

Algorithm 1 copy (adapted from [9])

```
function copy(cap, srcDispatcher, dstDispatcher)
        if not cap is MappingCap do
            copy cap from cspace(srcDispatcher) to cspace(dstDispatcher)
```

Map There are different cases of the *map* operation that have to be considered. We can either install a mapping from the virtual address space of an agent to a translation structure object (this can be thought of as choosing a root page table/ directory etc.). Or we can install a mapping from a translation structure object to another translation structure object. Or we map a translation structure object to an accepting, mappable resource, e.g. a *Frame* object or a *DevFrame* object. (All other object types like *PhysAddr*, *CNode* etc. are not mappable.)

Note that *TStructure* objects are a generalisation of all objects commonly used to install address translations. A possible example instance would be a page table.

Algorithm 2 map

```
//note that disp is the installing dispatcher
function map(vnodeCap, tstructureCap, disp) //case 1
        if givesMapAuthority(vnodeCap) && givesGrantAuthority(tstructureCap)
           && mappingLegal(vnodeCap, tstructureCap, disp) do
             install mapping.

function map(tstructureCapL, tstructureCapR, disp) //case 2
        if givesMapAuthority(tstructureCapL) && givesGrantAuthority(tstructureCapR)
           && mappingLegal(tstructureCapL, tstructureCapR, disp) do
             install mapping.

function map(tstructureCap, frameCap, disp) //case 3
        if givesMapAuthority(tstructureCap) && givesGrantAuthority(frameCap)
           && mappingLegal(tstructureCap, frameCap, disp) do
             install mapping.

function map(tstructureCap, devFrameCap, disp) //case 4
        if givesMapAuthority(tstructureCap) && givesGrantAuthority(devFrameCap)
           && mappingLegal(tstructureCap, devFrameCap, disp) do
             install mapping.
```

Delete The *delete* operations deletes a capability from a dispatcher's CSpace. There are several different cases that need to be considered for this operation. First, we have to differentiate the cases where we are deleting the last copy of a capability from the case where there are other copies of the capability present in the kernel state (even if they are are potentially located in some other dispatcher's CSpace). If we *are* deleting the last copy of a capability we need to check if the capability being deleted is a *MappingCap*. If it is, we remove the installed mapping corresponding to the *MappingCap* from the model's state.

Removing a mapping can potentially trigger the removal of additional mappings. We do not allow *dangling pointers* and if the removal of a mapping would lead to one, any mapping now leading nowhere is also removed. Remember the order of installation of the mappings in figure 11. The order of removal that does not trigger any additional removals is the reverse order of installation. Any other order will trigger the removal of all mappings "to the left" of the one to be removed.

Algorithm 3 delete (adapted from [9])

```
function delete(cap, disp)
    if cap is MappingCap then
```

```
        if cap is lastCopy then
            remove mapping.

            if "there were other mappings depending on this one" then
                remove them.

    remove cap from cspace(disp)
```

Revoke The *revoke* operation is described for Barrelfish [7] as follows: "*We define revoke recursively: for each descendant, revoke and delete that descendant. Simultaneously, delete all copies of the target capability.*"[9]

I extend this slightly to also take care of the cases where the capability that is being revoked has been mapped or is being mapped to. If this is the case, the mapping needs to be removed. Note that since the *MappingCap* corresponding to a mapping is a descendant of the capability that was mapped into an address space, it is revoked and afterwards deleted automatically. This will then trigger the removal of the mapping itself.

Algorithm 5 revoke (adapted from [9])

```
function revoke(cap, disp)
        if isMapped do // we only have to check for the "left" side here
            remove mapping.

        for all immediate descendants on all cores do
            revoke descendant.
            delete descendant.

        for all copies on all cores do
            delete copy
```

Unmap This "operation" removes a mapping. However, the *revoke* operation is easily powerful enough to take care of removing mappings also. It is enough to simply revoke either the *MappingCap*, the "left" side capability (granting authority over the object being mapped) or the "right" side (granting authority over the object we mapped to) used to install the mapping.

Retype *Retype* is undoubtedly one of the most important capability operations. It can be used in several different ways: We can retype a capability to change its (and the underlying object's) type (as visualised in figure 12), we can use it to resize a capability (by partitioning the backing object) or both. In theory it would even be possible to allow retyping a capability to several descending capabilities directly (e.g. when partitioning a *RAMCap*). However this isn't done for simplicity. See also here:

"*To retype a capability, we must check that no other capabilities in the system conflict with the retype. If no conflict is found, the retyped capability is created [...]. [...] For the sake of simplicity, we only create one output capability per retype in the protocol specification, allowing retype operations to specify a single sub-region of the source capability that is used for the output capability.*"[9]

Algorithm 2 retype (adapted from [9])

```
function retype(oldCap, srcDsp, newCap, dstDsp)
    if retypeLegal(oldCap, srcDsp, newCap, dstDsp) then
            cspace(dstDsp) <- newCap.

function retypeLegal(oldCap, srcDsp, newCap, dstDsp)
    if isMapped oldCap then
```

```
        return False.

    if not type(oldCap) "can be retyped to" type(newCap) then
        return False.

    if not object(newCap) "is contained in" object(oldCap) then
        return False.

    if "there exists a conflicting descendant" then
        return False.
```

5.9 Terminology

Let's fix the terminology used in the remainder of this section. We want to clearly define the different terms before we use them to specify the new addressing model formally. First I will state what the model's notion of *addresses* and *names* are. We also want to define what a *dispatcher*, a *dispatcher identifier*, an *address space identifier* and an *address space type* are. I will quickly recap *Object*, *ObjectType*, *Authority* and *CSpace*. Finally I will discuss capability equality and the *NullCapability*.

Address An *address* is simply a natural number (we include zero in our definition of the natural numbers for this thesis). The new addressing model does not impose any restrictions on address size (e.g. a number of bits) to stay as generally applicable as possible. All *addresses* need to be qualified with an *address space identifier* to be truly meaningful.

Name A *name* is an *address* that has been qualified with an *address space identifier*. This definition is inspired by the definition of a *name* in the decoding net [2]. Note that while in the old addressing model physical addresses were used as unique global identifiers, the new model uses *names*.

Dispatcher We call the agents of the new address translation model *dispatchers*, after barrelfish's term for them [7]. *Dispatchers* each hold a set of capabilities, their *CSpace*, and are tagged with a *dispatcher identifier*. An *address space identifier* belonging to their virtual address space is associated with them.

Dispatcher identifier (*DispatcherId*) This is simply a *dispatcher*'s identifier (a natural number that is unique to that *dispatcher*).

Address space identifier (*AdressSpaceId*) Each *address space* will be given its own unique identifier (a natural number) that can be used to qualify its *addresses*. Note that the identifier of an *address space* is fixed at the moment of its creation, it cannot be changed.

Address space type (*AdressSpaceType*) There are three different types of address spaces: *Virtual*, *intermediate* and *physical*. Virtual address spaces are added to the state of the model when a new dispatcher is created. They are associated with the completely virtual resource *VNode* that has no actual physical equivalent. When a dispatcher is removed from the model's state, so is its virtual address space. Intermediate address spaces are address spaces given rise to by translation structure objects. All other object types give rise to physical address spaces.

Object An *object* is simply a set of *addresses* that all belong to the same *address space*. (An *object* is not allowed to be split between two *address spaces*.) This set of *addresses* is tagged with an *object type*. Each *object* gives rise to an *address space*: It's type determines the type of the *address space*, its own *address space identifier* determines the identifier the *address space* will get. It's set of *addresses* is the set of *addresses* that will constitute the *address space*.

ObjectType There are several different *object types*: *PhysAddr, RAM, Frame, DevFrame, CNode, VNode* and *TranslationStructure*. They determine which kind of *address space* the object gives rise to, if it is mappable etc.

Authority An *authority* refers to a set of rights. We have three different types: $Access$, Map and $Grant$. $Grant$ is qualified with either an $Access$ or Map *authority* while the other two types are qualified by an *object*.

CSpace A *CSpace* is a *dispatcher*'s set of capabilities. All capabilities that are present in a *dispatcher*'s CSpace can be used as proof of having a certain *authority* over an *object*.

NullCapability The *NullCapability* is implicitly part of each CSpace. It does not give any dispatcher any authority, it is simply used as a sort of placeholder where a capability is expected but none can be given.

Capability equality For capabilities we need to differentiate between the case where we have two different copies of the same capability (which are considered equal for all purposes where authority over objects is concerned) and the case where we are dealing with two capabilities giving authority over different objects. In the latter case we never consider the capabilities equal. In the former case we consider them to be different *instances* of the same capability.

5.10 Finally, the formal model

See the figures 14 and 15 for a formal definition of the model that has been developed in the remainder of this section. First the different types of the new addressing model are defined as subsets of other, already known types. (Note that I chose to define $addresses$, $addressSpaceIds$, $objects$, $mappings$ and $dispatchers$ as strict subsets. This was a conscious choice. For example $addresses$ is a subset of $\mathbb{N}$ which is a countable set but has an infinite size. In contrast, $addresses$ is a large but very much finite set.)

After the basic type definitions some expressions are defined. These are either needed to define certain predicates like $legalMapping$ that defines when a new mapping may be installed or they are needed to formally write down some of the invariants following in the next section more formally later on.

Some conventions: To "create" a new object I write $type(arg_1, ..., arg_n)$ for a type that has been defined as $type \subseteq (dom_1 \times \ldots \times dom_n)$ or $type \subset (dom_1 \times \ldots \times dom_n)$. I also treat the singleton set and its single element as if they were equivalent: $\{e\} \equiv e$. (This keeps the definitions much shorter and more readable, just imagine having an invisible helper that packs and unpacks the singleton sets' elements as appropriate.)

Also I omitted defining $hasDescendants$ formally which is defined equivalently to how it would be defined for the Barrelfish capability system [7]. I did this to avoid having to add the bookkeeping information to the model, complicating it unnecessarily. (Of course the Haskell implementation of the model as an executable specification described in the next section (see 6) contains all this.)

$$
\begin{aligned}
\mathit{addresses} &\subset \mathbb{N}\\
\mathit{addressSpaceIds} &\subset (\mathbb{N} \cup \{\mathit{InvalidID}\})\\
\mathit{addressSpaceTypes} &= \{\mathit{VirtualAS},\ \mathit{IntermediateAS},\ \mathit{PhysicalAS}\}\\
\mathit{addressSpaces} &\subseteq (\mathit{addressSpaceIds} \times \mathit{addressSpaceTypes} \times \mathit{addresses} \times \mathit{addresses})\\
\mathit{names} &\subseteq (\mathit{addressSpaceIds} \times \mathit{addresses})\\
\mathit{objectType} &= \{\mathit{PhysAddr}, \mathit{RAM}, \mathit{Frame}, \mathit{DevFrame}, \mathit{CNode}, \mathit{VNode}, \mathit{TranslationStructure}\}\\
\mathit{objects} &\subset (\mathit{objectType} \times \mathit{names} \times \mathbb{N})\\
\mathit{mappings} &\subset (\mathit{addressSpaces} \times \mathit{addresses} \times \mathit{addressSpaces} \times \mathit{addresses} \times \mathbb{N})\\
\\
\mathit{rights} &= \{\mathit{access},\ \mathit{map},\ \mathit{grant}(\mathit{access}),\ \mathit{grant}(\mathit{map})\}\\
\mathit{capabilities} &\subseteq ((\mathit{objects} \times \mathit{rights} \times \mathit{capabilities} \times \mathit{capabilities}) \cup \{\mathit{NullCap}\}\\
\mathit{cspaces} &\subseteq \mathcal{P}(\mathit{capabilities})\\
\mathit{dispatchers} &\subset (\mathbb{N} \times \mathit{cspaces} \times \mathit{addressSpaceIds})\\
\mathit{state} &= (\mathit{dispatchers},\ \mathit{mappings},\ \mathit{addressSpaces})\\
\\
\mathit{right}(c) &= \{r \in \mathit{rights} \mid\\
&\qquad \exists o \in \mathit{objects},\ c_a \in \mathit{capabilities},\ c_c \in \mathit{capabilities}\ .\ \mathit{capability}(o, r, c_a, c_c) = c\}\\
\mathit{cspace}(a) &= \{c \in \mathit{capabilities} \mid\\
&\qquad \exists s_0 \subset \mathit{capabilities}, s_1 \subset \mathit{mappings}, s_2 \subset \mathit{addressSpaces}\ .\ (c \in s_0) \wedge (\mathit{dispatcher}(s_0, s_1, s_3) = a)\}\\
\mathit{object}(c) &= \{o \in \mathit{objects} \mid\\
&\qquad \exists c_a, c_c \in \mathit{capabilities},\ r \in \mathit{rights}\ .\ (\mathit{capability}(o,\ r,\ c_a,\ c_c) = c\}\\
\mathit{rights}(a) &= \{(r, o), r \in \mathit{rights},\ o \in \mathit{objects} \mid\\
&\qquad \exists\ c \in \mathit{capabilities}.\ (c \in \mathit{cspace}(a)) \wedge (r \in \mathit{right}(c)) \wedge (o \in \mathit{object}(c)\}\\
\mathit{asid}(\mathit{as}) &= \{\mathit{asi} \in \mathit{addressSpaceIds} \mid\\
&\qquad \exists t \in \mathit{addressSpaceTypes}, a_0, a_1 \in \mathit{addresses}\ .\ \mathit{addressSpace}(\mathit{asi}, t, a_0, a_1) = \mathit{as}\}\\
\mathit{addressesIn}(\mathit{as}) &= \{a \in \mathit{addresses} \mid (a >= \mathit{lowerLimit}(\mathit{as})) \wedge (a < \mathit{upperLimit}(\mathit{as}))\}\\
\mathit{lowerLimit}(\mathit{as}) &= \{a \in \mathit{addresses} \mid\\
&\qquad \exists a_u \in \mathit{addresses}, \mathit{asi} \in \mathit{addressSpaceIds}, t \in \mathit{addressSpaceTypes}\ .\ \mathit{as} = \mathit{addressSpace}(\mathit{asi}, t, a, a_u)\}\\
\mathit{upperLimit}(\mathit{as}) &= \text{equivalent to } \mathit{lowerLimit}\text{, omitted}\\
\mathit{baseAddress}(o) &= \{n \in \mathit{names} \mid \exists s \in \mathbb{N}, t \in \mathit{objectType}\ .\ \mathit{object}(t, n, s) = o\}\\
\mathit{type}(o) &= \{t \in \mathit{objectType} \mid \exists s \in \mathbb{N}, n \in \mathit{names}\ .\ \mathit{object}(t, n, s) = o\}\\
\mathit{size}(o) &= \{s \in \mathbb{N} \mid \exists n \in \mathit{names}, t \in \mathit{objectType}\ .\ \mathit{object}(t, n, s) = o\}\\
\mathit{sizeMapping}(m) &= \{s \in \mathbb{N} \mid\\
&\qquad \exists \mathit{as}_{\mathit{from}}, \mathit{as}_{\mathit{to}} \in \mathit{addressSpaces}, a_{\mathit{of}}, a_{\mathit{ot}} \in \mathit{addresses}\ .\ \mathit{mapping}(\mathit{as}_{\mathit{from}}, a_{\mathit{of}}, \mathit{as}_{\mathit{to}}, a_{\mathit{ot}}, s) = m\}\\
\mathit{from}(m) &= \{\ \mathit{as}_{\mathit{from}} \in \mathit{addressSpaces} \mid\\
&\qquad \exists s \in \mathbb{N}, \mathit{as}_{\mathit{to}} \in \mathit{addressSpaces}, a_{\mathit{of}}, a_{\mathit{ot}} \in \mathit{addresses}\ .\ \mathit{mapping}(\mathit{as}_{\mathit{from}}, a_{\mathit{of}}, \mathit{as}_{\mathit{to}}, a_{\mathit{ot}}, s) = m\}\\
\mathit{offsetF}(m) &= \{\ a_{\mathit{of}} \in \mathit{addresses} \mid\\
&\qquad \exists s \in \mathbb{N}, \mathit{as}_{\mathit{from}}, \mathit{as}_{\mathit{to}} \in \mathit{addressSpaces}, a_{\mathit{ot}} \in \mathit{addresses}\ .\ \mathit{mapping}(\mathit{as}_{\mathit{from}}, a_{\mathit{of}}, \mathit{as}_{\mathit{to}}, a_{\mathit{ot}}, s) = m\}\\
\mathit{addr}(n) &= a \in \mathit{addresses} \mid \exists \mathit{asi} \in \mathit{addressSpaceIds}\ .\ \mathit{name}(\mathit{asi}, a) = n
\end{aligned}
$$

Figure 14: The new model defined formally, part 1

$$\begin{aligned}
isAccepting(asi, a) &= \exists o \in objects\ . \\
&(asid(o) = asi) \wedge (a >= addr(baseAddress(o))) \\
&\wedge (a < addr((baseAddress(o)) + size(o))) \wedge \\
&(type(o) \in \{Frame,\ DevFrame\}) \\
isMapped(asi, a) &= (\exists m \in mappings\ . \\
&(asid(from(m)) == asi) \wedge (offsetF(m) <= a) \\
&\wedge (a < (sizeMapping(m) + offsetF(m)))) \\
&\vee isAccepting(asi, a) \\
legalMapping(as_0, c_0, as_1, c_1, s) &= (\forall a \in addressesIn(as_1)\ .\ isMapped(asid(as_1, a)) \\
&\wedge (\neg hasDescendants(c_0)) \\
&\wedge (\neg isMapped(c_0) \\
&\wedge (size(object(c_0)) = size(object(c_1))) \\
\text{convention: } &map(o) = (map, o),\ access(o) = (access,\ o), \\
&grant(access(o)) = (grant(access), o),\ grant(map(o)) = (grant(map), o)
\end{aligned}$$

Figure 15: The new model defined formally, part 2

5.11 Invariants

I've identified several invariants for the new address translation model and I've written them down in "plain English" here to make it as painlessly as possible for the reader to follow my reasoning. Many of these invariants may seem to be of the type "Obviously this holds ..." but it can still be of use to write them all down to establish a clean starting point.

1. The *NullCapability* does not give any dispatcher any authority over any object.

2. All instances of the *NullCapability* are equal to each other, there exists only one *NullCapability.*

3. All dispatchers are spawned with a capability to their own virtual address space. Note that this is not a capability bestowing authority over an actual physical resource but over a purely "virtual" one. It is modelled as a *VNode* object.

4. If a dispatcher is able to read/write (access) an arbitrary address a that is part of address space x, they must hold a capability (this could potentially be "only" a mapping capability) for a range of addresses in address space x that contains address a. The reverse implication **DOES NOT** hold. (It may be - at least in theory - that an agent holds a capability for, say, a frame capability but is unable to map it into it's own virtual address space. This could happen if the agent has previously - and foolishly - deleted its capability to its own virtual address space.)

5. If an agent is able to install a mapping from address a to address b, they must have the right to grant access authority over address b and have either map authority over address a or be able to grant it to themselves. The reverse implication **DOES NOT** hold. (For the same reason as given above.)

6. A mapping from address a to b may only be installed if either b is an *accepting* address (part of an accepting address range) or an arbitrary mapping from b to some other address is already installed. (This invariant is also called the "No dangling pointers!" invariant.)

7. The last copy of a mapping from address a to b will only be removed if a is either a virtual address part of some dispatcher's virtual address space or no mapping currently installed in the system maps any address to address a. (This is enforced by cascading deletes in case the "No dangling pointers!" invariant would otherwise be violated.)
8. If any of the capabilities used for installing a mapping is revoked, the mapping must be removed.
9. Mappings can never be changed, only removed once they have been installed.
10. Direct "remapping" is disallowed. A mapping first has to be deleted before its left capability can be remapped.
11. MappingCaps are dispatcher local, they may not be transferred to another dispatcher.
12. Mappings are removed when the last copy of a MappingCap is deleted.

5.12 Concluding remarks

At this point of this thesis we have defined a new addressing model. Most importantly, it overhauls the old model by abolishing the idea of a single physical address space, swapping it out for address spaces that are added to the model's state whenever new resources are added to the system.

After investigating the shortcomings of the old addressing model, studying the choice of the "right" rights and representing the model's state we sketched a new addressing model. We extended it further by adding the extra layer of the capabilities to it to make the model better applicable to Barrelfish [7]. We then specified the how the capability operations need to be implemented before specifying the new model formally with the help of set notation. Finally, we looked at some invariants that need to hold.

We will now, in the next section, section 6, go a step further down the road of formalisation and build an executable Haskell specification that implements our new addressing model.

6 The executable Haskell specification

This section presents an implementation of the new addressing model developed in the previous section. I developed an *executable specification* in Haskell that implements the more abstract model of the previous section. The role of the Haskell implementation is inspired by seL4's executable Haskell specification [12]. It is intended to serve as an actually executable instance of the model, making it possible to use it as an executable model of a concrete system implementing the new addressing model.

When determining the Haskell implementation's level of abstraction, one could say that it is sandwiched between the abstraction layer of the just defined address space and memory management model and that of an actual, concrete operating system implementing said model (see figure 16). An example would be the extension of the Barrelfish OS presented in [1], called Barrelfish/MAS. Having an executable model make it possible to generate traces (with the executable Haskell specification) that correspond directly to log traces logging the state of a suitable OS's capability system.

Note that the set of *authorities* used in the Haskell model (as defined for the new addressing model in the previous section) are essentially the same as the set of rights that is presented in [1], tough in the paper they are written down a bit more concisely.

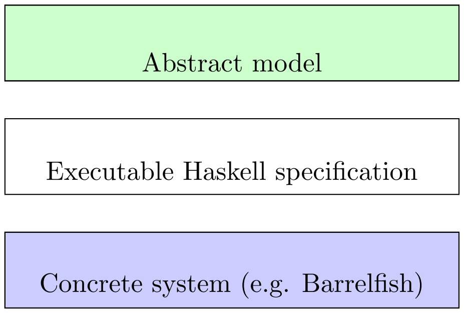

Abstract model
Executable Haskell specification
Concrete system (e.g. Barrelfish)

Figure 16: Stack of abstraction

The following subsections will first discuss design decisions made and goals set before developing the Haskell specification. Then I will proceed to analyse the structure of the executable specification, describe its important data types and state some invariants. I will also shed some light on the specific implementation of the state monad the executable specification employs.

6.1 Preliminary design decisions and design goals

During the very early stages of model development, I experimentally defined some of the necessary data types using Haskell's syntax. Haskell is a functional language that allows the user to elegantly and with very little overhead define different parts of an abstract model as algebraic data types, yielding easy to read definitions (even for readers not familiar with Haskell's syntax). This, together with being inspired by seL4's use of Haskell for rapid prototyping [12], lead me to choose to also develop an executable specification of the new addressing model in Haskell.

My design goals for the Haskell model were as follows:

- Develop a model that can be *compiled* and therefore automatically checked by a compiler.

- Develop an *executable* model, making it possible to simulate changes of the model state caused by a specific operation by simply executing that very same operation in the Haskell model.
- Develop a model that allows its user to model whole *sequences of operations*, and having the generated output correspond to log traces of a concrete system implementing the model.
- Develop a model that separates the set of possible traces (sequences of capability operations) into sets of *valid* and *invalid* traces.
- Develop a model that will be applicable to traces of the capability system of the Barrelfish OS.

I will now start presenting the executable specification by first describing the structure of the Haskell implementation in the next subsection.

6.2 The specification's structure

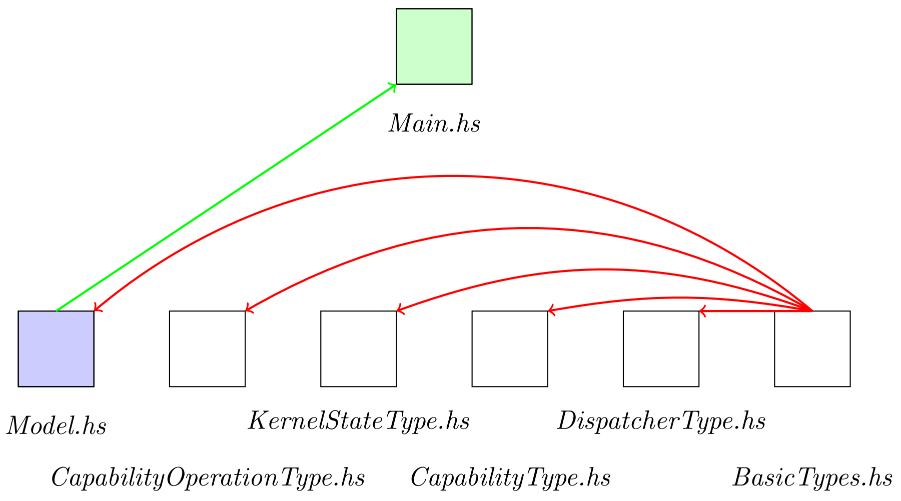


Figure 17: File structure of the executable specification

The executable Haskell model consists of several Haskell files (our modules): *Main.hs*, *Model.hs*, *CapabilityOperationType.hs*, *KernelStateType.hs*, *CapabilityType.hs*, *DispatcherType.hs* and *BasicTypes.h*. Each of these files is imported by all of the files listed before it in the enumeration just given (as can also be seen in figure 17, note that only the "is imported by" relations of *Model.hs* and *BasicTypes.hs* are shown to save space). The exception is *Main.hs* which only imports *Model.hs*. The interface exposed by *Model.hs* should be seen as the interface of the model itself.

I will give a quick description of the content of the files at this point. It should be taken only as a means of orientation, more details will be given later on in this section.

- *Main.hs* contains some example trace functions, which model sequences of capability operations, illustrating thereby how the Haskell specification should be used.
- *Model.hs* contains all the specifications's implementations of capability operations, defining them as monadic Haskell functions. The full list is: removeDispatcher, spawnDispatcher, retype, copy, access,

delete, revoke, create, map, init. In addition, *Model.hs* contains some getter functions (also monadic, e.g. getVSpaceCap) and a number of helper functions.

- *CapabilityOperationType.hs* is mainly concerned with my implementation of the state monad. It also implements logging and tracing functions for debugging. Examples would be `updateLog` and `updateTrace`.
- *KernelStateType.hs* defines the algebraic data type called `KernelState` that represents the (biggest part of the) state of the model.
- *DispatcherType.hs* defines the `Dispatcher` data type.
- *BasicTypes.hs* contains some basic type definitions, e.g. `Address` and `Authority`.

Now, we will move on to a more in-depth descriptions of the Haskell specifications data types.

6.3 The specifications's data types

Let's start with the most basic building blocks: The executable Haskell specification models any address simply as a natural number (see figure 18). It does not concern itself with the bit-length of addresses since this is - at this level of abstraction - of no further relevance and we want to avoid losing generality unnecessarily.

`AddressSpaces` are modelled as contiguous address ranges that are tagged with a unique identifier and an address type. It would easily have been possible to model them simply as address sets but to keep the model as lean as possible and avoid dealing with a lot of hairy edge conditions, the assumption was made that address spaces are always contiguous ranges of addresses. (This assumption does not lead to a loss of generality since any set of addresses that do not form a contiguous range can always be modelled by another set of contiguous addresses by simply ordering the original set.)

```haskell
data Address = Address Natural deriving (Eq, Ord)
-- ...

data AddressSpaceID = AddressSpaceID Natural | InvalidASID deriving (Eq, Ord)
-- ...
data AddressSpace =
          VASpace { asi :: AddressSpaceID, start :: Address, end :: Address }
        | IASpace { asi :: AddressSpaceID, start :: Address, end :: Address }
        | PASpace { asi :: AddressSpaceID, start :: Address, end :: Address }
        | NullAddressSpace
-- ...

```

Figure 18: Code snippet from *BasicTypes.hs*, defining the basic address types

The data type called `Authority` (see figure 19) models the different types of authority an agent can have over a resource. Any resource is simply represented as an `Object` in the Haskell specification. An agent can have either the authority to *access* or to *map* an object. It is not possible to have both authorities at the same time on the same object. It is necessary to ensure this to prevent an agent directly accessing and potentially writing any object that is interpreted as an address translation structure (e.g. a page table).

It is possible for an agent to have the grant authority on either an access authority or a map authority (always with respect to a specific object). The data type `Authority` is recursively defined, allowing the `Grant` data constructor to be applied to either an instance of the data type `Authority` constructed with the

```haskell
data Authority =
  Access Object |
  Map Object |
  Grant Authority deriving (Eq, Ord, Show)

```

Figure 19: Code snippet from *BasicTypes.hs*, defining the authority data type

```haskell
data Object =
    PhysAddr { baseAddress :: Name, size :: Natural } |
    RAM { baseAddress :: Name, size :: Natural } |
    Frame { baseAddress :: Name, size :: Natural } |
    DevFrame { baseAddress :: Name, size :: Natural } |
    CNode { baseAddress :: Name, size :: Natural } |
    VNode { baseAddress :: Name, size :: Natural} |
    TranslationStructure { baseAddress :: Name, size :: Natural }
    deriving (Eq, Ord)
```

Figure 20: Code snippet from *BasicTypes.hs*, defining the object data type

`Access` constructor or the `Map` constructor. The invariant mandating that no agent ever has both the access and the map right to the same object, is not affected by this.

Any authority an agent has is always defined with respect to a specific resource, represented in the executable specification by the data type `Object`. For its definition, see figure 20. The different data constructors of the algebraic data type `Object` correspond to the different types of resources represented in the Haskell specification.

To match the Barrelfish OS as closely as possible, the Haskell specification also calls authorities (which are sets of rights) on objects capabilities. In the executable specification a capability is simply an encapsulation of authority over a memory object (plus some bookkeeping information). There are different types of `Capability` objects, corresponding to the different `Object` types just introduced. Agents, which are called `Dispatchers` in the Haskell specification, just like in Barrelfish, hold each a set of capabilities (which we refer to as their CSpace).

```haskell
data Capability =
  PhysAddrCap { rights :: Authority,
    ancestor :: Capability, copiedFrom :: Capability } |
  RAMCap { rights :: Authority, ... } |
  FrameCap { rights :: Authority, ... } |
  DevFrameCap { rights :: Authority, ... } |
  TSCap { rights :: Authority, ... } |
  CNodeCap { rights :: Authority, ... } |
  VSpaceCap { rights :: Authority, ...} |
  DispatcherCap { dispatcherID :: Natural, ... } |
  ASCap { rights :: Authority, ... } |
  MappingCap { left :: Capability,
    right :: Capability, mapping :: Mapping, ... } |
  KernelCap |
  NullCap deriving (Ord)
```

Figure 21: Code snippet from *CapabilityType.hs*, defining the capability data type

For the definition of the `Dispatcher` data type see figure 22. Dispatchers are simply modelled as sets of capabilities (their CSpace) that are tagged with both a `DispatcherID` and an `AddressSpaceID`. The latter links a dispatcher to its virtual address space that will be added to the executable specification's state when the dispatcher is created, and removed, when the dispatcher itself is removed again.

```haskell
data Dispatcher = Dispatcher DispatcherID (Set Capability) AddressSpaceID

```

Figure 22: Code snippet from *DispatcherType.hs*, defining the dispatcher data type

One of the most important data types of the Haskell specification is the data type `KernelState`. It represents the state of the whole specification (with exception of tracing and logging information). All valid `KernelState` objects are comprised of three components: First there is the set of currently active dispatcher objects, then a `MappingDB` object which stores all installed mappings and last but not least there is a set of all used `AddressSpace` objects. Note that some of this information is redundant, e.g. the set of address space could be dropped. The reason for this is that facilitates logging and allows easier debugging of the executable specification.

```haskell
data KernelState =
  KernelState (Set Dispatcher) MappingDB (Set AddressSpace) |
  InvalidState deriving (Eq)

```

Figure 23: Code snippet from *KernelStateType.hs*, defining the KernelState data type

There is exists an alternative data constructor for the data type `KernelState` which is called `InvalidState`. The executable specification will transition into this state whenever an operation would otherwise lead to an inconsistent or disallowed state of the Haskell spec.

6.4 The specification's state monad

I implemented the Haskell specification using my own, specifically for this executable specification defined state monad, called `CapabilityOperation`. This allowed me to express changes to the Haskell spec's state as a sequence of capability operations such as creating, copying, retyping, mapping a specific capability etc. These sequences of capability operations can be thought of as modelling log traces of a concrete, the new addressing model implementing operating system's capability system. Each log trace of such a system's capability system corresponds vice versa to a specific sequence of `KernelStates` observed when executing the equivalent capability operations in the Haskell specification.

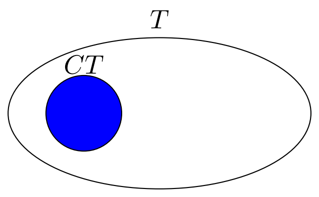


Figure 24: Venn diagram depicting the set of traces T containing the set of correct traces CT

Note that contained within the set of all possible traces T, there is a set of correct traces $CT \in T$ that correspond to sequences of consistent `KernelStates`. All other traces indicate that execution had to be aborted at some point since an operation was applied that would otherwise have led to transitioning to an inconsistent/ disallowed system state. See figure 24 for a visualisation.

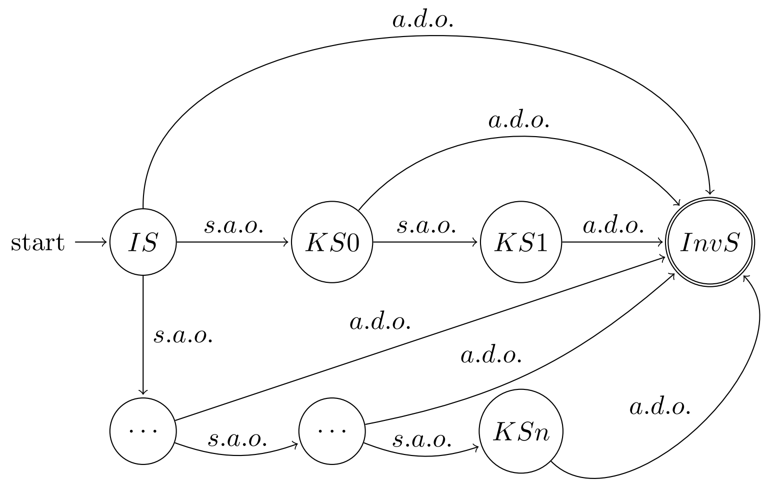


Figure 25: Visualization of the state automaton underlying the Haskell specification, IS = Initial state, KSX = x-th KernelState, InvS = InvalidState, a.d.o. = any disallowed operation, s.a.o. = some allowed operation

Behind the scenes the Haskell specification behaves like a finite state automaton (albeit one with a very large number of states). Each state of the FSA corresponds to a specific `KernelState`, the total number of states is equal to the number of possible different `KernelStates` that can theoretically be composed.

```haskell
-- | unCapOp is a helper function for implementing bind.
unCapOp :: (CapabilityOperation a) -> (State -> (a, State) )
unCapOp (CapabilityOperation f) = f


data CapabilityOperation a = CapabilityOperation (State -> (a, State))
instance Monad (CapabilityOperation) where
    --(>>=) :: (CapabilityOperation a) -> (a -> (CapabilityOperation c)) ->
    --  (CapabilityOperation c)
    f >>= g =
        CapabilityOperation (\ks -> let (a, ks') = unCapOp f ks in unCapOp (g a) ks')
    --return :: a -> (CapabilityOperation a)
    return x = CapabilityOperation (\ks -> (x, ks))
instance Applicative (CapabilityOperation) where
    pure = return
    (<*>) = ap
instance Functor (CapabilityOperation) where
    fmap f (CapabilityOperation m) =
        CapabilityOperation $ \ks -> let (a, ks') = m ks in (f a, ks')
```

Figure 26: Code snippet from *CapabilityOperationType.hs*, defining the data type CapabilityOperation

Figure 25 sketches the state automaton of the Haskell spec. There is one accepting state, namely the `InvalidState`, an initial state and n different `KernelStates`. (Note that since system exit is not explicitly modelled, it is always possible to extend a trace of `CapabilityOperations` further as long as no operations have been performed that were disallowed with regard to the `KernelState` that was the most recent one when they were being executed.)

One of the most far-reaching design decisions made when developing the model was the introduction of the `InvalidState`. It dictates how operations that - if applied - would lead to an undesirable state (either because it would be inconsistent or forbidden) are handled. It allows for an unequivocal partition of all traces (sequences of simulated capability operations) into valid and invalid ones. Once the state of the model has transitioned to `InvalidState`, any further operation (implemented by a monadic function) that the user requests to be executed will no longer have any influence on the `KernelState`. From this it follows, that it is enough to check the end state of the executable specification to differentiate between valid and invalid traces. This is desirable because it facilitates making statements about sets of traces.

See figure 26 for the actual Haskell definition of the state monad. Note that the specification's state is not parameterised with the data type `KernelState` directly but with the data type `State`, a wrapper data type that contains a `KernelState` and combines it with some book keeping information. Important is that `CapabilityOperation` is an instance of the `Monad` type class. It is therefore mandatory for it to implement `bind` and `return`. In addition, the three fundamental monad laws have to hold. The laws are as follows:

$$\begin{array}{llcr} 1. & (return\ x) >>= f & == & f\ x \\ 2. & m >>= return & == & m \\ 3. & (m >>= f) >>= g & == & m >>= (\backslash x \rightarrow f\ x >>= g) \end{array}$$

For some more details regarding the difference between `State` and `KernelState` take a look at figure 27. It shows the crucial getter and setter (or in this case putter) functions that are used mostly behind the scenes to implement the `State` monad, logging and tracing functionality and be able to manipulate the state of the executable specification directly where absolutely necessary (mostly in the functions that simulate the capability operations).

```haskell
-- | This is a getter for the KernelState.
getKernelState :: (CapabilityOperation (KernelState))
getKernelState = CapabilityOperation (\s@(State ks _ _) -> (ks, s))

-- | This is a setter for setting the KernelState.
putKernelState :: KernelState -> CapabilityOperation ()
putKernelState new_ks = CapabilityOperation (\s@(State ks t l) -> ((), State new_ks t l))

getState :: (CapabilityOperation State)
getState = CapabilityOperation (\s -> (s, s))

putState :: State -> CapabilityOperation ()
putState new_s = CapabilityOperation (\s -> ((), new_s))
```

Figure 27: Code snippet from *CapabilityOperationType.hs*, defining getters and setters

A monad can be viewed as sort of promise to perform an operation at some point in the future. To *actually* trigger execution the function `runCapabilityOperation` is used. It's exact definition is given in figure 28 but for the most part it is enough to know that it will run any well-formed trace function it is given.

```
runCapabilityOperation :: (CapabilityOperation a) -> (a, State)
runCapabilityOperation op =
    let ks = (KernelState DS.empty (MappingDB DS.empty) DS.empty) in unCapOp op (State ks (Trace [
```

Figure 28: Code snippet from *CapabilityOperationType.hs*, defining the runner function

Let me finish this subsection with some facts about `AddressSpaces` as they are used and treated as in the executable specification. Address spaces are not managed explicitly by the user of the Haskell spec. An address spaces is created and added to the `KernelState` automatically when a new `Capability` for a memory object located within this address spaces is created. (Note that capabilities may only be created by a special, privileged `Dispatcher`. This `Dispatcher` is used to model external events such as new resources being connected to the system.) Once no capability present in any of the `Dispatchers`' CSpaces references an address space anymore, it is removed from the `KernelState`.

If an agent wants to trigger the removal of an address space, it can simply `revoke` and then delete the capability that originally lead to the address space being added to the `KernelState` in the first place (or one of that capability's copies). (Note that revoking a capability ensures that all its copies and descendants that were created by retyping the original capability are deleted.)

6.5 Invariants

The most important invariant of the executable specification is without doubt that no agent/ dispatcher has ever *both* the map authority and the access authority to one and the same `Object`. This is enforced automatically by the fact that on retyping the type of the `Object` determines if an agent can have either a map or an access authority on the `Object`. (Note that it would theoretically be possible for the trusted dispatcher *priv_d* to break this invariant but since it is per definition *trusted to not break any invariants*, this does not matter. See the next section for more details regarding *priv_d*.)

$$\forall a \in dispatchers,\ o \in objects\ .\ (map(o) \in a \rightarrow \neg(access(o) \in rights(a))) \land \tag{1}$$
$$(access(o) \in rights(a) \rightarrow \neg(map(o) \in rights(a))) \tag{2}$$

Another important invariant is the "No dangling pointers!" invariant. Any access to an address that is mapped must (after an arbitrary, but finite) number of translation steps finally be accepted by an object. This enforced recursively: If it is assumed that the invariant holds for all already installed mappings, any mapping from an unmapped address to an already mapped address will provably preserve the invariant (the proof is by induction).

This is enforced by the `map` function checking that no address range in any address space is every mapped to any address range in any address space that is not already mapped (or composed of accepting addresses). If the user still tries to install such an illegal mapping, the specifications's state will simply transition to `IllegalState` as described previously.

$$\forall m \in mappings\ .\ \forall a \in to(m)\ .\ isMapped(asid(m), a) \tag{3}$$

Another invariant is that if there is no capability of any kind referencing an object (in any CSpace of any agent) then no agent has any authority over that object. This holds trivially for the executable specification as the the set of authorities an agent has is represented as a set of capabilities (its CSpace).

$$\begin{aligned}
&\forall o \in objects\ .\ \neg(\exists c \in capabilities\ .\ object(c) = o) \rightarrow && \qquad (4)\\
&\qquad \neg(\exists a \in dispatchers\ . \qquad (map(o) \in rights(a)\vee && \qquad (5)\\
&\qquad\qquad (map(o) \in rights(a)\vee && \qquad (6)\\
&\qquad\qquad (grant(map(o)) \in rights(a)\vee && \qquad (7)\\
&\qquad\qquad (grant(access(o)) \in rights(a)) && \qquad (8)
\end{aligned}$$

6.6 Conclusion

I want to conclude this section with some thoughts on lessons I learned while implementing the executable Haskell specification. First, I am still convinced that Haskell is the right language for this task. It is both suitably high level to be used for modelling while at the same time it is rigorous enough to serve as a specification. Most importantly, it is - unlike some of the more arcane modelling tools and languages that are out there - still relatively easy to debug and well documented.

One thing I would do differently is that I would reverse the order of implementation of certain sections. I would implement the logging and debugging functionality first to facilitate early testing of the other functions. As it was, I developed many parts of the model "top down". I sketched how different functions needed to work together, figuring out the details (and how to test certain functionality) later on.

Of course, as with any piece of code longer than three lines, there are still additional features that could be added and parts that could be improved upon. For more details, please see the future work section.

7 Evaluation

The executable Haskell specification has of yet mainly been tested and evaluated by a few carefully created example traces. I decided to favour this form of evaluation for two reasons: Firstly, the number of valid traces is quite small when compared with the number of possible traces. In other words, there exist many traces that simply go to `InvalidState` in more or (often) less interesting ways. Secondly, the logs generated by the model are very detailed and therefore time consuming to comb through manually. This has made it impractical to randomly generate traces to test the Haskell model. Instead I chose to create a few example trace functions that exercise specific parts of the executable specification.

This section contains several subsections showcasing different parts of the executable spec by providing some example trace functions and logs generated by the executable Haskell specification. First, there is a subsection walking the reader through an example trace function called `paperTrace`. This function maps a memory frame into a dispatcher's virtual address space. There follows a subsection describing the logging and debugging system of the Haskell spec and a subsection describing dispatcher management. Finally there is a trace function serving as an example of how external events can be modelled.

After the examples used to evaluate the executable Haskell specification, there is a subsection about its limitations. This is then be followed by a subsection concerning an actual operating system implementation of the new addressing model (and the Haskell specification).

7.1 The paperTrace

For an example of how to use the executable Haskell specification, see figure 29. It shows a Haskell function called `paperTrace` (the example function of [1] is based on this function). First `paperTrace` calls the `init` function. This function needs to be called at the beginning of each and every "trace function" (A function modelling a sequence of capability operations). It sets up the spec's state: Initialising the `KernelState`, starting two initial dispatchers and setting up the logging. It also returns the `DispatcherID` of the two initial dispatchers and a `Capability` to a block of memory.

After initialising the spec's state with `init`, `paperTrace` calls `getPartitionCapability` three times. This is a helper function that returns a `Capability` object but does not actually change the spec's state in any way. The function simply assembles a new `Capability` object referring to a partition of the resource referred to by the original `Capability` object. (In the example we partition the memory represented by `mem` into three equally sized parts.) The new `Capability` object returned by `getPartitionCapability` can later be passed as an argument to the `retype` function.

All functions actually changing the executable specification's state are monadic functions. (They can be recognised easily by the typical `<-` syntax used to bind their return value to a name.) `retype` is such a function and it is called by `paperTrace` to retype a number of capabilities. First, `mem` is retyped to get two smaller memory capabilities. Then it is used to change the type of different capabilities. (Note how the helper function `changeTypeOfCap` is used to compute the corresponding capability of a different type to later be able to pass it as an argument to the `retype` function.)

Next, `getVSpaceCap` is called to get a reference (in the form of a `Capability` object) to the dispatcher's VSpace. Note that all dispatchers receive such a capability when they are created. VSpace capabilities do not refer to any *physical* resource but to the *immaterial* resource of the dispatcher's VSpace (an address space comprised of all addresses the dispatcher can possibly emit). Following this line of reasoning, it makes sense that all newly created dispatchers automatically receive such a capability.

After retyping `vsCap` to a descending capability referring to a fraction of the VSpace, `paperTrace` first maps a `FrameCap` capability to a `TSCap` (translation structure capability). Then the same `TSCap` is mapped to our new `VSpaceCap`. Now the dispatcher is able to access any part of the memory frame represented by the just mapped `FrameCap`, it just has to pass the corresponding address to the `access` function.

Finally, `paperTrace` returns the current specification state in the form of the `KernelState` object used to represent the state of the system. After executing `paperTrace`, the main function will print the final `KernelState` to the console, followed by a trace of all observed `KernelState`s.

```haskell
paperTrace :: (CapabilityOperation KernelState)
paperTrace = do
    -- initialize the system state
    (init_d_id, priv_d_id, mem) <- init

    let mem_1of3 = getPartitionCapability mem 3 1 -- helper function
        mem_2of3 = getPartitionCapability mem 3 2
        mem_3of3 = getPartitionCapability mem 3 3

    -- retype mem into two smaller chunks of memory
    res <- retype mem init_d_id mem_2of3 init_d_id
    res <- retype mem init_d_id mem_3of3 init_d_id

    -- change the type of some of the capabilities
    let ramCap01 = changeTypeOfCap mem_2of3 RAMC -- helper function
    res <- retype mem_2of3 init_d_id ramCap01 init_d_id
    let frameCap01 = changeTypeOfCap ramCap01 FC
    res <- retype ramCap01 init_d_id frameCap01 init_d_id
    let ramCap02 = changeTypeOfCap mem_3of3 RAMC
    res <- retype mem_3of3 init_d_id ramCap02 init_d_id
    let tsCap01 = changeTypeOfCap ramCap02 TSC
    res <- retype ramCap02 init_d_id tsCap01 init_d_id

    -- get the capability to the VSpace of the dispatcher and retype it
    vsCap <- getVSpaceCap init_d_id
    let vsCap_1of24 = getPartitionCapability vsCap 24 1 -- helper function
    res <- retype vsCap init_d_id vsCap_1of24 init_d_id

    -- perform the mapping
    mapCap01 <- Model.map tsCap01 frameCap01 init_d_id
    mapCap02 <- Model.map vsCap_1of24 tsCap01 init_d_id
    -- vsCap_1of24 is mapped to tsCap01 is mapped to frameCap01

    ks <- getKernelState
    return ks

```

Figure 29: Example trace function called paperTrace

For an excerpt of the logging information printed to the console when running `paperTrace`, see the figures 30 and 31. They show the final `KernelState` logged to the console (and formatted somewhat more nicely for this report).

7.2 `updateLog` and `updateTrace`

It is vital for development (in any language, any framework) to have a way of finding out "what's going on internally". This is especially true when observing unexpected or unwanted behaviour. Model development is no exception to this rule; nothing is more frustrating than being able to tell "that something's not right" but at the same time not being able to pinpoint the source of the problem.

The executable Haskell spec relies mainly on the two functions called `updateLog` and `updateTrace` (see 7.2) to visualise the internal state of the specification for the user. `updateLog` is similar to a print statement. It allows the user to print a `String` of any form to the console at any point in a trace function (a function comprised of `CapabilityOperations`). This is extremely useful for debugging since the usual `putStrLn`

```
KernelState -->
Dispatcher 1, VASID=ASID 2:
[   PhysAddrCap<0x2948e84864b8043a>: PAO{ba=Name{asid=ASID 1, addr=0x00000000},
                                      size=1073741824}
                                      a=0x00000000 cf=0x00000000 r=g-a
    PhysAddrCap<0x5291d090b41ab31e>: PAO{ba=Name {asid=ASID 1, addr=0x15555555},
                                      size=357913941}
                                      a=0x2948e84864b8043a cf=0x00000000 r=g-a
    PhysAddrCap<0x5291d090c9700873>: PAO{ba=Name {asid=ASID 1, addr=0x2aaaaaaa},
                                      size=357913941}
                                      a=0x2948e84864b8043a cf=0x00000000 r=g-a
    RAMCap<0x7bdab8d9037d6202>: RAMO{ba=Name {asid=ASID 1, addr=0x15555555},
                                 size=357913941}
                                 a=0x5291d090b41ab31e cf=0x00000000 r=g-a
    RAMCap<0x7bdab8d92e280cac>: RAMO{ba=Name {asid=ASID 1, addr=0x2aaaaaaa},
                                 size=357913941}
                                 a=0x5291d090c9700873 cf=0x00000000 r=g-a
    FrameCap<0xa523a12152e010e6>: FO{ba=Name {asid=ASID 1, addr=0x15555555},
                                   size=357913941}
                                   a=0x7bdab8d9037d6202 cf=0x00000000 r=g-a
    TSCap<0xa523a12192e010e5>: TSO{ba=Name {asid=ASID 1, addr=0x2aaaaaaa},
                                size=357913941}
                                a=0x7bdab8d92e280cac cf=0x00000000 r=gm-
    CNodeCap<0x5291d0908a13df7e>: CNO{ba=Name {asid=ASID 1, addr=0x00000000},
                                   size=10737418}
                                   a=0x2948e84864b8043a cf=0x00000000 r=g-a
    VSpaceCap<0x9a242dcfa7dd5c83>: VNO{ba=Name {asid=ASID 2, addr=0x00000000},
                                    size=357913941}
                                    a=0x4d1216e8c9440397 cf=0x00000000 r=-m-
    VSpaceCap<0x4d1216e8c9440397>: VNO{ba=Name {asid=ASID 2, addr=0x00000000},
                                    size=8589934592}
                                    a=0x00000000 cf=0x00000000 r=-m-
    DispatcherCap<0xdc36d1615b7400a5>:1 a=0x00000000 cf=0x00000000
    MappingCap: mapping=Mapping{from=IntrAddressSpace id=1 0x2aaaaaaa - 0x3fffffff,
                              to=PhysAddressSpace id=1 0x15555555 - 0x2aaaaaaa,
                              offFrom=0x2aaaaaaa, offTo=0x15555555, size=357913941}
    MappingCap: mapping=Mapping{from=VrtlAddressSpace id=2 0x00000000 - 0x15555555,
                              to=IntrAddressSpace id=1 0x2aaaaaaa - 0x3fffffff,
                              offFrom=0x00000000, offTo=0x2aaaaaaa, size=357913941}
]
Dispatcher 2, VASID=ASID 3:
[   CNodeCap<0x5291d0908a13df7e>: CNO{ba=Name {asid=ASID 1, addr=0x00000000},
                                   size=10737418} a=0x2948e84864b8043a cf=0x00000000
                                   r=g-a
    VSpaceCap<0x4d1216e8c9440398>: VNO{ba=Name {asid=ASID 3, addr=0x00000000},
                                   size=8589934592} a=0x00000000 cf=0x00000000
                                   r=-m-
    DispatcherCap<0xdc36d1615b7400a6>:2 a=0x00000000 cf=0x00000000
    KernelCap
]

...

```

Figure 30: Log output of the paperTrace function, final KernelState, part I

```
MappingDB (fromList [
Mapping {from=VrtlAddressSpace id=2 0x00000000 - 0x15555555,
         to=IntrAddressSpace id=1 0x2aaaaaaa - 0x3fffffff,
         offFrom=0x00000000, offTo=0x2aaaaaaa, size=357913941},
Mapping {from=IntrAddressSpace id=1 0x2aaaaaaa - 0x3fffffff,
         to=PhysAddressSpace id=1 0x15555555 - 0x2aaaaaaa,
         offFrom=0x2aaaaaaa, offTo=0x15555555, size=357913941}])
asdb={
VrtlAddressSpace id=2 0x00000000 - 0x0000000015555555
IntrAddressSpace id=1 0x2aaaaaaa - 0x000000003fffffff
VrtlAddressSpace id=2 0x00000000 - 0x00000001ffffffff
VrtlAddressSpace id=3 0x00000000 - 0x00000001ffffffff
PhysAddressSpace id=1 0x00000000 - 0x0000000040000000
PhysAddressSpace id=1 0x15555555 - 0x000000002aaaaaaa
PhysAddressSpace id=1 0x2aaaaaaa - 0x000000003fffffff
NullAddressSpace
}
```

Figure 31: Log output of the paperTrace function, final KernelState, part II

cannot be used for that purpose (due to being of type `String -> (IO ())`).

`updateTrace` is more powerful than `updateLog`. It essentially takes a snapshot of the spec's state whenever it is called, labelling that snapshot with the `String` passed to it as an argument. The snapshot taken is of the type `KernelState` and includes all dispatchers (with their complete CSpaces) present in the system, all currently installed mappings and all address spaces. (See the figures 30 and 31 for an example of how a single `KernelState` looks like when logged to the console.) Behind the scenes the Haskell specification keeps track not only of the current `KernelState` but also all ever taken snapshots. These are ordered to form a trace and printed to the console eventually.

All monadic functions forming the core of the Haskell specification (`retype`, `create`, `map`, `spawnDispatcher`, `removeDispatcher`, `access`, `delete`, `revoke`, `copy`, `init` and many of their subfunctions) are set-up to call `updateTrace` on entry and exit. The label normally not only contains the name of the function called but also a number indicating which path the execution took when exiting the function. However, this is only a convention and not enforced in any way.

7.3 Managing dispatchers

Figure 33 shows how the executable Haskell spec represents spawning and removing dispatchers. The corresponding monadic functions are named `spawnDispatcher` and `removeDispatcher`. Their use is straight forward: `spawnDispatcher` takes as arguments the `DispatcherID` of the original dispatcher spawning a new one and a set of initial capabilities. Note that the new dispatcher will be created with copies of the initial capabilities present in its CSpace, remember to update the capability reference when referring to this copies.

The `removeDispatcher` function is even more simple to use. It accepts the `DispatcherID` of the to be removed dispatcher as its argument and returns `True` upon successful removal.

7.4 Modelling external events

The Haskell specification represents an external event such as hot-plugging an additional block of RAM etc. with help of the special dispatcher called *priv_d*. It is created by the `init` function which also returns its `DispatcherID` to the user so that they may reference *priv_d* and have it perform certain capability operations. In *priv_d*'s CSpace a special, unique capability is stored, called `KernelCap`. Only a dispatcher

```
updateTrace :: String -> CapabilityOperation ()
updateTrace label = do
    (State ks trace log) <- getState
    let l = Label label
        new_trace = extendTrace (l, ks) trace
    putState (State ks new_trace log)
    return ()

updateLog :: String -> CapabilityOperation ()
updateLog l = do
    (State ks trace log) <- getState
    let new_log = extendLog l log
    putState (State ks trace new_log)
    return ()

```

Figure 32: Code snippet showing the signatures of updateLog and updateTrace

```
manageDispatchersTrace :: (CapabilityOperation KernelState)
manageDispatchersTrace = do
    (init_d_id , priv_d_id , mem) <- init

    -- prepare arguments for retyping and spawning
    let mem_5of10 = getPartitionCapability mem 10 5
        ramCap01 = changeTypeOfCap mem_5of10 RAMC
        cnodeCap = changeTypeOfCap ramCap01 CNC
        initialCaps = insert cnodeCap empty

    res <- retype mem init_d_id mem_5of10 init_d_id
    res <- retype mem_5of10 init_d_id ramCap01 init_d_id
    res <- retype ramCap01 init_d_id cnodeCap init_d_id

    -- spawn and remove a new dispatcher
    id <- spawnDispatcher init_d_id initialCaps
    res <- removeDispatcher id

    ks <- getKernelState
    return ks

```

Figure 33: Code snippet showing how dispatchers can be managed

```haskell
-- example trace to show of the modelling of external events
modelExternalEvent :: (CapabilityOperation KernelState)
modelExternalEvent = do
    (init_d_id , priv_d_id , mem) <- init

    asi <- getNewAddressSpaceID
    let baseAddress = Name asi 0x0
        new_obj = PhysAddr baseAddress (1 `shift` 32)
        rights = Grant (Access new_obj)
        new_mem = PhysAddrCap rights NullCap NullCap

    res <- create new_mem (getAddrSpace new_obj) priv_d_id

    ks <- getKernelState
    return ks

```

Figure 34: Code snippet showing how external events can be managed

with this capability present in its CSpace is authorised to use the `create` function to add new capabilities to the systems state by creating them "out of thin air".

See figure 34 for an example. After initialising the specification's state with `init`, we first get a fresh `AddressSpaceID`. Then we construct a new object by manually setting its base address (this can be anything we like since it will be located in a new address space anyway) and its size. *priv_d* finally creates our new block of memory as `PhysAddr` capability.

Note that it is possible for *priv_d* to pass on its special `KernelCap` capability. This would be equivalent to a trusted agent declaring other agents to also be trusted and might correspond to a trusted dispatcher starting another dispatcher that needs to be endowed with additional rights because it serves a special function (as a memory server or some kind of driver).

7.5 Limitations of the executable Haskell model

There are some parts of the executable specification that currently model their corresponding parts of the Barrelfish OS rather coarsely. While remaining at a high level of abstraction can sometimes be beneficial to reduce complexity, it might become necessary in the future to "close the gap" between the abstract and the concrete further. For a start, this might be done by implementing the check that a newly spawned dispatcher has been given all necessary starting capabilities in greater detail. Currently it only asks for a merely symbolic `CNode` capability. However, it might be worth considering if it should check for the presence of capabilities necessary to build translation structures or enforce that a minimum size of physical capabilities have been given to the newly spawned dispatcher.

Another point is that right now capability storage is only modelled at a very high level. A dispatcher must receive a capability for a "symbolic" `CNode` but no minimum size of that capability is enforced and no additional capabilities are needed once the CSpace of the dispatcher grows beyond a certain point. Since the CSpace is only modelled as a set, a dispatcher can hold an arbitrary number of capabilities without needing any additional `CNode` capabilities.

7.6 Evaluation of the C implementation presented in [1]

In [1] the interested reader can find the description and evaluation of a fully functional (and fully concrete) implementation of the new addressing model. This implementation, written in C and following the executable Haskell specification, is an extension of the Barrelfish OS [7],[1].

The new implementation, called Barrelfish/MAS, adds support of multiple physical address spaces to the Barrelfish OS. Barrelfish/MAS, in contrast to “vanilla” Barrelfish, deals with address spaces explicitly in all its capability operations. It does this by, among other things, “having capabilities which refer to memory objects hold the object’s canonical base name, the size of the object they are referring to, as well as its type and rights” [1]. The canonical name is implemented as a combination of an address space identifier (ASID) and the address within the to the address space identifier corresponding address space (just like *name* is defined in the new addressing model and in the Haskell specification).

The hardware topology Barrelfish is running on (meaning the whole decoding net comprised of different address spaces) is explored and discovered at runtime. All information gained about the topology is stored in the SKB (System Knowledge Base), a database that can be queried by different parts of the system.

How does Barrelfish/MAS (and the new addressing model’s) performance compare to a more traditional operating system’s performance? All performance evaluations presented in the paper were done using a dual-socket Intel Xeon E5 v2 2600 (“Ivy Bridge”) with 256GB of main memory and 10 cores per socket. As an example Linux OS Ubuntu 18.04LTS, kernel version 4.15 was chosen. The evaluations performed show Barrelfish/MAS to have comparable (and sometimes even better) performance than Linux’s virtual memory system.[1]

8 Conclusion

This thesis investigates how best to bring support for multiple, physical address spaces to an operating system based on capability management (like Barrelfish OS [7]). After identifying the old addressing model as a potential hazard for correct and secure memory management due to today's systems having become to complex for it, we've identified the old addressing model's single global physical address space as its major flaw. After acknowledging that the old model needs to be updated, we've developed a new addressing model (see 14 and 15) to replace the old, oversimplified one 5.

To be able to do this, we've built on the pre-existing work concerning the decoding net model by Achermann et al. [11],[2]. It models modern hardware systems as complex networks of address spaces but does not contain methods to model updating or reconfiguring address translation at runtime.

After fitting our new model with capabilities by wrapping its concept of sets of rights called *authorities* with an extra layer of abstraction to match the capability system of the Barrelfish OS [7] and specifying its operations, an executable implementation of the new addressing model, written in Haskell, has been presented. We've taken a quick look at seL4's version of an executable specification, which was an inspiration for our executable specification.

We have also seen the Haskell specification's basic data types (matching those of the new addressing model) and looked at the implementation of its state monad and tracing functionality.

We've then evaluated the Haskell specification implementing our new addressing model with the help of a few carefully crafted trace functions. We saw how the mapping of address ranges is modelled by the Haskell specification, how dispatcher's are managed and how an external event is represented. An example log generated by the executable specification was presented to show how detailed the capability system's state is modelled.

By citing [1] I reason that it is possible that the newly developed addressing model can be used to configure real, complex systems and scales well enough even when confronted with pathological systems. This thesis hopes therefore to be able to contribute to more reliable memory management in the future.

9 Future work

This section concerns possible future work to be done on the executable Haskell specification. A potential starting point would be to allow the user of the model to modify mappings. Right now, there is no way to declare a mapping read-only but a downgrading of authority from read and write (expressed by the *Access* authority) to read-only could be implemented without to much additional work.

Another so called "low hanging fruit" would be to allow dispatchers to either have more than one virtual address spaces or to implement a way to switch out the virtual address space of a dispatcher after the dispatcher has already been spawned (or created). Right now a dispatcher has a field for the `AddressSpaceID` of its virtual address space but this field could be switched out for either a set or even a list of address space identifiers. This would be easy enough to do, however, the `access` function would also have to be overhauled. Switching out the virtual address space might be a bit more tricky because it would involve deciding what to do with the mapping already installed into the old address space.

Another logical extension of the Haskell model would be to allow the installation of more general functions into the translation structures. As of now, mappings are defined by the two address spaces they connect, two offsets (one for each address space) and a size. This could be made more general to allow the modelling of additional address translations schemes (e.g. multi-stage translation).

Right now, the input address range is determined by the location in memory of a translation structure (qualified by the address space identifier it is part of, of course). It is theoretically conceivable that this assumption could be loosened. I did not implement this due to time constraints and to keep the model as lean as possible. It stands to reason that to support this, a lot of edge cases would need to be handled.

The most exciting (but also most far out into the future) extension to the executable specification would undoubtedly be to automatically generate Haskell trace functions from Barrelfish[7] log traces and to be able to check them for compliance with a set of invariants. This would enable the Haskell model to ensure that the specification it embodies is actually translated accurately to a concrete system.

10 References

[1] Achermann, R., Hossle, N., Humbel, L., Schwyn, D., Cock, D., and Roscoe, T. A least-privilege memory protection model for modern hardware. Unpublished, submitted to a conference.

[2] Achermann, R., Humbel, L., Cock, D., and Roscoe, T. Physical addressing on real hardware in isabelle/hol. In *Interactive Theorem Proving* (Cham, 2018), J. Avigad and A. Mahboubi, Eds., Springer International Publishing, pp. 1–19.

[3] Aho, A., and Ullman, J. *Foundations of Computer Science.* W. H. Freeman, 1994.

[4] Basin, D. Access Control I, 2017. Part of the slides made accessible as part of the 2017 Information Security course.

[5] Bishop, M. Applying the take-grant protection model. Tech. rep., Dartmouth College, Hanover, NH, USA, 1990.

[6] Elkaduwe, D., Klein, G., and Elphinstone, K. Verified protection model of the sel4 microkernel. In *Proceedings of the 2Nd International Conference on Verified Software: Theories, Tools, Experiments* (Berlin, Heidelberg, 2008), VSTTE '08, Springer-Verlag, pp. 99–114.

[7] ETHZ's systems group. The Barrelfish operating system.

[8] ETHZ's systems group. TN-013-CapabilityManagement.pdf.

[9] Gerber, S. *Authorization, Protection, and Allocation of Memory in a Large System.* Doctoral thesis, ETHZ, 2018.

[10] https://github.com/seL4/l4v/graphs/contributors. seL4 Github repository.

[11] Humbel, L., Achermann, R., Cock, D., and Roscoe, T. Towards correct-by-construction interrupt routing on real hardware. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (New York, NY, USA, 2017), PLOS'17, ACM, pp. 8–14.

[12] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 207–220.

[13] Lampson, B. W. Protection. *SIGOPS Oper. Syst. Rev. 8*, 1 (Jan. 1974), 18–24.

[14] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 2–13.

[15] Lipton, R. J., and Snyder, L. A linear time algorithm for deciding subject security. *J. ACM 24*, 3 (July 1977), 455–464.

[16] Moscibroda, T., and Mutlu, O. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), SS'07, USENIX Association, pp. 18:1–18:18.

[17] Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[18] Seshadri, V., Lee, D., Mullins, T., Hassan, H., Boroumand, A., Kim, J., Kozuch, M. A., Mutlu, O., Gibbons, P. B., and Mowry, T. C. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, ACM, pp. 273–287.

[19] Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., and Klein, G. sel4 enforces integrity. In *Proceedings of the Second International Conference on Interactive Theorem Proving* (Berlin, Heidelberg, 2011), ITP'11, Springer-Verlag, pp. 325–340.

[20] Wikipedia's community. Wikipedia, Access Control Matrix, May 2019.

[21] Wikipedia's community. Wikipedia, L4 microkernel family, August 2019.