A Principled Approach To Kernel Memory Management

A thesis submitted to the School of Computer Science and Engineering at the University of New South Wales in fulfilment of the requirements for the degree of Doctor of Philosophy.

Dhammika Elkaduwe

2010

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: Elkaduwe

First name:Dhammika

Other name/s: Karunadhipathi Wasala Herath Mudiyanse Ralahamilage Dhammika Darshana Bandara

Abbreviation for degree as given in the University calendar: PhD

School: Computer Science and Engineering

Faculty: Engineering

Title: A Principled Approach To Kernel Memory Management

Abstract 350 words maximum: (PLEASE TYPE)

Small kernels are a promising approach to secure and reliable system construction. These systems reduce the size of the kernel to a point where it is feasible to formally verify the implementation correctness of the kernel with respect to an abstract formal model of the kernel's behaviour. The abstract formal model facilitates the enforcement, and reasoning about the enforcement, of different policies between user-level components that constitute the system, and the formal verification connects the reasoning with the deployed kernel.

In this work, a model for managing the in-kernel memory of a formally verified, small kernel is designed and evaluated for its formal and empirical characteristics. The design eliminates all implicit memory allocations within the kernel by promoting all dynamically allocated kernel memory into first-class, explicitly allocated kernel objects. This reduces the problem of kernel memory management to that of controlling the authority to perform these explicit allocations and controlling the dissemination of authority over already allocated objects.

A formal security model capturing the behaviour of the kernel is developed by extending the take-grant model and formal analysis is carried out to demonstrate that the model is capable of enforcing spatial partitioning and isolation. The extension preserves the decidability of take-grant while providing the ability to reason about kernel memory consumption of components which is not feasible in the original model.

Performance of the model is evaluated using a prototype implementation based on an L4 microkernel. The analysis shows no performance degradation due to exporting all in-kernel memory allocations to user-level. When enforcing spatial partitioning over a para-virtualised Linux kernel the model shows performance improvements compared to a L4 based system enforcing a similar policy by run-time monitoring and shows similar performance to a L4 system that attempts no control over memory consumption, and a Xen-based system.

This work demonstrates the feasibility of exporting all in-kernel memory allocations to user-level through a capability-based, decidable, security model. The model shows no performance degradation and can be used to make strong formal guarantees on memory consumption of a component.

Declaration relating to disposition of project thesis/dissertation

....
Signature Witness Date

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

ORIGINALITY STATEMENT

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

Signed …….....

Date …….....

Related publications

[1] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch and Simon Winwood. **seL4: Formal verification of an OS kernel.** *Proceedings of the 22nd ACM Symposium on Operating Systems Principles,* Big Sky, MT, USA, October, 2009.

[2] Dhammika Elkaduwe, Gerwin Klein and Kevin Elphinstone. **Verified protection model of the seL4 microkernel.** *Proceedings of Verified Software: Theories, Tools and Experiments 2008,* Toronto, Canada, October, 2008.

[3] Dhammika Elkaduwe, Philip Derrin and Kevin Elphinstone. **Kernel design for isolation and assurance of physical memory.** *1st Workshop on Isolation and Integration in Embedded Systems,* Glasgow, UK, April, 2008.

[4] Dhammika Elkaduwe, Gerwin Klein and Kevin Elphinstone. **Verified protection model of the seL4 microkernel.** Technical Report NRL-1474, NICTA, October, 2007.

[5] Dhammika Elkaduwe, Philip Derrin and Kevin Elphinstone. **A memory allocation model for an embedded microkernel.** *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems,* Sydney, Australia, January, 2007.

[6] Dhammika Elkaduwe, Philip Derrin and Kevin Elphinstone. **Kernel data – first class citizens of the system.** *Proceedings of the 2nd International Workshop on Object Systems and Software Architectures ,* Victor Harbor, South Australia, Australia, January, 2006.

Abstract

Small kernels are a promising approach to secure and reliable system construction. These systems reduce the size of the kernel to a point where it is feasible to formally verify the implementation correctness of the kernel with respect to an abstract formal model of the kernel's behaviour. The system is composed of user-level components, isolated from one another using the kernel-provided mechanisms. The abstract formal model facilitates the enforcement, and reasoning about the enforcement of different policies between these user-level components. However, existing formal models only capture the application-level interface of a small kernel with no clear relationship between the externally visible access control model and the kernel's low-level management of physical memory.

In this work, a model for managing the in-kernel memory of a formally verified, small kernel is designed and evaluated for its formal and empirical characteristics. The design eliminates *all* implicit memory allocations within the kernel by promoting *all* dynamically allocated kernel memory into first-class, explicitly allocated kernel objects. This reduces the problem of physical memory management within the kernel to that of controlling the authority to perform these explicit allocations and controlling the dissemination of authority over already allocated objects.

A formal protection model that unifies in-kernel management of physical memory with access control is developed by extending the take-grant model. A formal analysis carried out on the above developed model demonstrates that the model is capable of enforcing spatial partitioning and isolation. The extension preserves the decidability of the original take-grant model while providing the ability to reason about kernel memory consumption of components which is not feasible in the original model.

Performance of the model is evaluated using a prototype implementation based on an L4 microkernel. The analysis shows no significant performance degradation due to exporting all in-kernel memory allocations to user-level. When enforcing spatial partitioning to a para-virtualised Linux kernel the model shows performance improvements compared to a L4 based system enforcing a similar policy by run-time monitoring and shows similar performance to a L4 system that attempts no control over memory consumption and a Xen based system.

This work demonstrates the feasibility of exporting *all* in-kernel memory allocations to user-level resource managers through a capability-based, decidable, protection model. The model shows no performance degradation in the scenarios examined and can be used to make strong formal guarantees on memory consumption of components.

Acknowledgement

This thesis would not have been possible without the help and the encouragement of many. In particular, lot of the credit for successfully completing this thesis should go to my supervisor Kevin Elphinstone. His persistent support, guidance and advice made this work a reality. I am grateful to my co-supervisor Gernot Heiser for his advice and feedback on the project when I needed them.

In addition to my supervisors, a number of others at ERTOS enriched the project with their knowledge and skills. The support and the guidance given by Gerwin Klein was instrumental for the formal modelling and analysis of the kernel's protection model. I would like to thank Carl Van Schaik, Peter Chubb and Ben Leslie for enriching the seL4 and seL4::Wombat implementations with their vast knowhow on low-level system programming. In addition to these direct involvements I want to thank all my colleagues at ERTOS for their input and the team spirit which made a difficult journey enjoyable.

Even with the help and support of all the above mentioned, this work would not have been possible without the support and the understanding of my better-half Risheeka. I greatly appreciate her understanding and support during a stressful period. Last but not least my deepest gratitude goes to my parents, without whom none of this would have been possible.

Contents

1 Introduction **1**
- 1.1 Microkernel-Based Systems . . . 2
- 1.2 A Formally Verified Microkernel . . . 3
- 1.3 Managing Kernel Memory . . . 4
- 1.4 Protection Models . . . 5
- 1.5 Aim and Scope of Research . . . 5
- 1.6 Contributions . . . 7
- 1.7 Overview of this thesis . . . 8

2 Related Work **9**
- 2.1 Kernel Memory Management . . . 9
  - 2.1.1 In-Kernel Policy . . . 10
  - 2.1.2 Kernel Memory as a Cache . . . 13
  - 2.1.3 User-Level Management . . . 14
  - 2.1.4 Summary . . . 15
- 2.2 Formal Modelling . . . 17
  - 2.2.1 Background . . . 18
  - 2.2.2 Classical Access-Control Models . . . 19
  - 2.2.3 Summary . . . 22

3 Managing Kernel Memory **23**
- 3.1 L4.verified – Formal Verification . . . 23
  - 3.1.1 Overview of the Methodology . . . 23
  - 3.1.2 Proof Effort: Case Study . . . 24
- 3.2 Summary of Existing Approaches . . . 27
  - 3.2.1 Reasoning About Memory . . . 28
- 3.3 seL4 Approach: Rationale . . . 28

4 Conceptual Model **30**
- 4.1 Overview of seL4 . . . 31
  - 4.1.1 Basic Kernel Model . . . 31
  - 4.1.2 System Structure . . . 31
  - 4.1.3 seL4 Memory Management . . . 32
  - 4.1.4 Kernel Objects . . . 33
- 4.2 Overview of Memory Management API . . . 34
- 4.3 Kernel Object Allocation . . . 34
  - 4.3.1 Type and Memory Safety of Kernel Objects . . . 36
  - 4.3.2 Preventing User-Level Access to Kernel Data . . . 36
  - 4.3.3 The Capability Derivation Tree . . . 36

4.3.4 Summary of the Retype Operation . . . 39
4.4 Recycling Memory . . . 39
4.5 Reusing Memory . . . 40
4.6 Managing Address Spaces . . . 41
4.6.1 Exception Model . . . 42
4.6.2 Capability Address Space . . . 43
4.6.3 Virtual Memory Address Space . . . 46
4.7 Implementation Details . . . 50
4.7.1 Untyped Memory Abstraction . . . 50
4.8 Summary . . . 52

5 Formal Model of Authority **53**
5.1 The Take-Grant Model . . . 54
5.1.1 The seL4 protection model — Informal Introduction . . . 56
5.2 Isabelle/HOL system . . . 57
5.3 The seL4 Protection model . . . 58
5.3.1 Semantic Entities . . . 58
5.3.2 Protection State . . . 59
5.3.3 Operational Semantics . . . 60
5.3.4 The Initial System State . . . 64
5.3.5 Fundamental Invariant of the Model . . . 65
5.4 Informal Correspondence to seL4 . . . 65
5.4.1 Concrete and Abstract Protection States . . . 66
5.4.2 Concrete and Abstract Operations . . . 68
5.5 Summary . . . 70

6 Formal Analysis of the Protection Model **71**
6.1 Informal Discussion of Requirements . . . 71
6.1.1 Capability Based Protection . . . 72
6.2 Predicates . . . 73
6.3 Necessary Conditions for Propagation of Authority . . . 75
6.4 Enforcing Subsystems . . . 82
6.4.1 Example System . . . 86
6.4.2 Relation to Concrete System . . . 89
6.4.3 Reducing Resource Manager's Complexity . . . 89
6.5 Information Access Control . . . 90
6.5.1 Isolated Components . . . 90
6.5.2 Proof of Isolation . . . 92
6.5.3 Enforcing Isolation . . . 95
6.6 Summary . . . 96

7 seL4::Pistachio **97**
7.1 Cost of Managing Kernel Objects . . . 98
7.1.1 Cost of Allocating Kernel Objects . . . 98
7.1.2 Prototype Allocator . . . 102
7.1.3 Memory Overhead of Untyped Capabilities . . . 102
7.1.4 Cost of Revoking Memory . . . 103
7.1.5 Cost of Recycling Kernel Objects . . . 105
7.1.6 Different Management Policies . . . 106

7.1.7 Limitations . . . 106
7.2 Cost of Capabilities . . . 107
7.2.1 Layout of a CSpace . . . 107
7.2.2 Capability Lookup Cost . . . 108
7.2.3 Managing a CSpace . . . 111
7.3 Performance of Kernel Primitives . . . 113
7.3.1 IPC operation . . . 114
7.3.2 Thread Operations . . . 114
7.3.3 Performance of VSpace Operations . . . 115
7.4 Summary . . . 119

8 Performance Evaluation **120**
8.1 seL4::Wombat . . . 121
8.1.1 Iwana— Best-Effort Allocator . . . 123
8.2 L4/Iguana System . . . 125
8.2.1 Spatial Partitioning . . . 126
8.3 Wombat::Insecure system . . . 127
8.4 Results . . . 127
8.4.1 Lmbench Results . . . 128
8.4.2 AIM9 Benchmark Suite . . . 133
8.5 Summary . . . 134

9 Conclusion **135**

List of Figures

1.1 Microkernel-based system construction 2

3.1 L4.verified approach to kernel verification. 24

4.1 Example system configuration. 32
4.2 Allocation of different kernel object types using the retype method 35
4.3 The capability derivation tree. 37
4.4 Propagation of untyped capabilities using mint and imitate operations. . . 38
4.5 An example CSpace layout. 44
4.6 Deleting CNode objects 45
4.7 Creating a virtual address space 47

5.1 Take-grant authority distribution rules 55
5.2 Graphical representation of seL4 authority distribution rules 63

6.1 Example system configuration 72
6.2 The effect of self-referencing capabilities 76
6.3 The effect of creating new entities 77
6.4 The effect of SysCreate on connected 80
6.5 Example System Configuration 83
6.6 Example Subsystem Configuration 88

7.1 Variation of the object allocation cost. 100
7.2 Example system with diverse resource management policies 107
7.3 The variation of capability lookup cost for a single-level tree. 110
7.4 The variation of capability lookup cost for a two-level tree. 110

8.1 The seL4::Wombat system. 122
8.2 Wombat system configurations. 126
8.3 The variation of pipe bandwidth of seL4, normalised to native Linux. . . . 131

List of Tables

2.1 Properties of existing kernel memory management schemes 16

3.1 Break-down of lines of Isabelle proof scripts related to kernel memory management. 25

4.1 Summary of the kernel memory management API. 34

5.1 Summary of the operations in the seL4 protection model. 56
5.2 Correspondence between concrete kernel objects and protection state entities. 66
5.3 Relationship between the operations of the concrete kernel and those of the protection model. 68

7.1 The cost of allocating various kernel objects. 99
7.2 The cost of reusing memory. 104
7.3 The cost of recycling kernel objects. 105
7.4 Performance of capability lookups 109
7.5 Performance of CSpace management operations 112
7.6 Cost of IPC operation. 113
7.7 Performance of thread operations. 116
7.8 Summary of VM benchmarks 117
7.9 Cost of managing virtual memory 118

8.1 Summary of sub-tests in benchmarks. 127
8.2 Results of the lmbench benchmarking suite. 129
8.3 Performance of seL4 and Xen on ARM. 132
8.4 Results of the AIM9 benchmarking suite. 133

Chapter 1

Introduction

Society has become increasingly dependent on computer systems. Consequently, the security and reliability of computer systems is a significant issue.

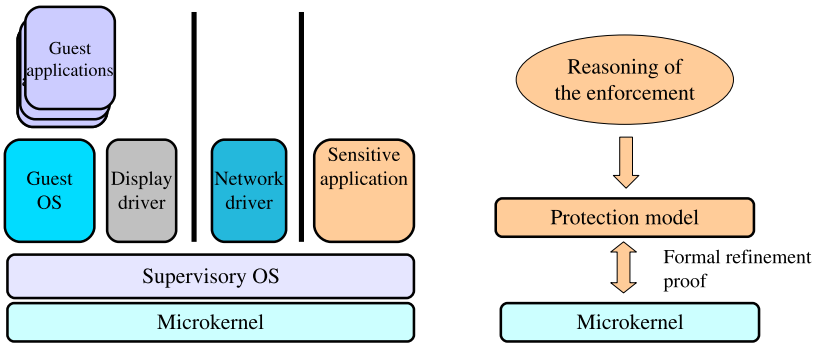
A key component in constructing a secure and reliable computer system is the operating-system (OS) kernel. The kernel, defined as the part of the system that executes in the (most) privileged mode of the processor, has unlimited access to hardware resources. Therefore, any defect within the kernel is capable of undermining the security and reliability of the entire system.

Mainstream OSes, unfortunately, are far from being secure and reliable. A major factor of their poor track-record is their large size and the monolithic design. In terms of size, they feature millions of lines of code. As an example, the source tree of Linux-2.6.27 is composed of just under 6.4 million lines of code [osU08]. It is unlikely that this entire source tree will be included in a single OS image, but since the design is monolithic, whatever the portion that is selected — which is generally a large subset of the source tree (except when the OS configuration is highly specialised) — runs in the most privileged mode of the processor.

General wisdom is that bugs in any sizable code base are inevitable. Even well-engineered software can expect to have in the order of 3 to 6 bugs per thousand lines of code [Hat97]. Some parts of a monolithic kernel can have a much higher defect density—Chou et.al. [CYC+01] reports two to 75 bugs per thousand lines of device driver code — putting the number of potential defects that can undermine the system's security and reliability within a monolithic system into the tens of thousands.

A widely accepted approach to constructing secure and reliable system software is to reduce the amount of kernel code in order to minimise the exposure to bugs. This is a primary motivation behind *microkernels* [BH70, WCC+74, ABB+86, SFS96, HPHS04, Lie93], *separation kernels* [IAD07, Rus81], *isolation kernels* [WSG02], *MILS* (multiple independent levels of security) architecture [AFOTH06] and small *hypervisors* [SPHH06, SLQP07]. In addition to reducing the potential for defects, the reduction in the code size means that it is feasible to guarantee the absence of defects within the kernel through formal *verification* [TKH05, HT05a, SLQP07].

This thesis focuses on the approach of using a formally verified microkernel as a reliable foundation for a wide rage of application domains, including those that requires strong isolation guarantees. Specifically, the thesis examines the issue of what is an appropriate model for managing the kernel's internal memory, so that, once implemented and verified in a microkernel, would provide a practical and secure foundation for a variety of systems, while facilitating reasoning about security properties of systems built upon the microker-



a) Microkernel based system construction. Thick lines show the isolation boundaries.

b) Protection model formally connected t the kernel to facilitate reasoning.

Figure 1.1: Microkernel-based system construction and the use of formal protection models to facilitate reasoning.

nel.

1.1 Microkernel-Based Systems

A pictorial representation of a system constructed using the microkernel-based approach is shown in part (a) of Figure 1.1. As mentioned, the kernel is reduced to a bare minimum: only the essential functionality is kept inside the kernel [Lie95]. Traditional OS services such as virtual memory, interrupt handling, device drivers, file systems etc; are provided outside of the kernel by means of de-privileged (i.e. *user-level*) components. This enables modular-systems construction—the system is composed of smaller and hence tractable components. They are robust as faults are isolated within components and once detected a faulting components can be restarted [HBG+07, HBG+06, Hil92]. They are flexible and extensible as user-level components can be removed, replaced or added [Lie96].

In between the microkernel and these user-level components is a small, domain-specific *supervisory* OS personality (or *supervisor*), which is responsible for bootstrapping and enforcing a suitable, domain-specific *system-level policy* over the execution of the components. For some (arguably most) classes of systems, the desired policy is isolation, so that a component failure is isolated from the rest. If components are real-time, then system-level isolation policy should extend to both spatial and temporal domains. Or, in another extreme, where higher performance is the important metric, system-level policy might trade isolation between a few or all components in favour of performance.

In addition to the system-level policy, components themselves may enforce a *subsystem-level* policy over the components they manage. An illustrative example is when the microkernel is used as a virtualisation platform. The guest OS will have it's own, fine-tuned policy that it would enforce over the applications (sub-components) it manages. In general, a system can have a number of such enforcement layers, with each layer enforcing a policy over the components it manages. The requirements of each layer on the kernel to enforce the policy is similar to that of the supervisor. Thus in the following discussion I only focus on the supervisor and the enforcement of system-level policy.

1.2 A Formally Verified Microkernel

The success of microkernel-based systems depends, to a great extent, on the system's ability to enforce these domain-specific, diverse policies. A key component in achieving this is the microkernel — it must be equipped with sufficient API mechanisms such that the supervisor can enforce and affirm the enforcement of the required policy. For example, if the desired system-level policy is to isolate components, then there must be a sequence of API operations that the supervisor can use to bootstrap the components into isolated compartments and *assure* that these compartments are not breached in any future state, irrespective of component execution. Ideally, assurance here must be in the form of a formal proof based on an abstract *protection model* (or security model) of the kernel and a formal connection between the protection model and the kernel's implementation (see part (b) of Figure 1.1). Here the protection model facilitates reasoning about the policy enforcement, and the formal connection (or the *refinement proof*) links the reasoning to the deployed kernel.

A related project called *L4.Verified* [CKS08, Boy09, KEH+09] is working on a refinement proof from a formal model of the kernel to its implementation. Hereafter, I use the term *verified kernel* to refer to a kernel that is connected to its abstract model via a refinement proof. Suppose the L4.Verified project verifies the kernel to adhere to a specific formal model. Then the design of the kernel and therefore it's formal model should enable the construction of secure and reliable systems and moreover, the formal model should facilitate reasoning about whether the systems built on top have the properties of interest.

Formal models and their properties are well studied in the literature (for example [Lam71, HR75, LS77, San92a]). To date, the published models, focus *only* on the application level (or the *API* level), with the tacit assumption that the kernel's internal behaviour, when providing services to user-level components does not undermine the properties enforced at the API level. A clear relationship between the model used for reasoning and the low-level management of physical memory in the kernel is either non-existent, insufficient or ad hoc. Ideally, there should be a clear, systematic (potentially formal) relationship between the kernel's physical memory management and the formalism used for reasoning, so that the same policy enforced at API level is adhered to within the kernel.

This thesis focuses on a kernel design (a model) that unifies an access control based protection model and the management of in-kernel memory. So that the reasoning done using the protection model is adhered to within the kernel model.

However, the thesis does not attempt a grand unified design and a model to facilitate reasoning about all properties—for example, temporal properties and information flow are beyond the scope. Nor does it attempt to formally connect the model with the kernel implementation. As mentioned earlier, connecting the model with the kernel implementation is the aim of the related L4.verified project.

Unifying the protection model with the low-level management of kernel memory requires tackling two considerations. First, the (unified) protection model needs to facilitate reasoning—by looking at an initial state, one should be able to reason about the future states and in particular assure the enforcement of the desired policy.

Second, both the kernel implementation and the protection model should be usable across a broad range of system configurations without any modification. Even a small change or an extension to the kernel or the protection model, invalidates the refinement proof and depending on the modification, reestablishing refinement can take a significant amount of effort.

1.3 Managing Kernel Memory

The microkernel provides primitive abstractions to all user-level components — the very same components over which the system is attempting to enforce the desired, domain specific policy. The exact type and nature of the abstractions provided by the microkernel varies depending on the kernel API, but generally these include threads, inter-process communication (IPC), address spaces and so on. In providing these abstract services the kernel consumes physical memory. For example, a thread requires memory within the kernel to store its state and associated metadata. An address space requires page tables for storing the associated mappings, and depending on the semantics of the abstraction, additional kernel memory is needed for bookkeeping. This physical memory consumed by the kernel in implementing its abstractions is referred to as *kernel memory* or kernel *metadata*.

The policy used by the kernel to manage its memory should match the policy enforced over user-level components. For example, if the system requirements are such that two components need to be isolated, then the same isolation boundaries must be reflected within the kernel with regards to kernel memory management. If the system warrants a cache-colouring scheme, either to improve the real-time predictability [LHH97] or to improve performance in a multi-core system [CJ06, ZDS09], then the kernel should adhere to the same scheme when allocating kernel memory for those components. If not, a malicious component may use this mismatch to undermine the user-level enforcement.

The rationale here is that, if a policy is to be enforced faithfully, then the same policy should be applied, without exceptions, to *all* executions paths of the component — including the paths it may take through the kernel when obtaining kernel services. If there is a mismatch between the kernel's internal memory management policy and the policy warranted by the application domain, then we have a situation where different policies are applied to the same component. A malicious component can leverage this difference and undermine the enforcement. For example, (some versions of) the *L4* microkernel [Lie95] used a first-come-first-sever (FCFS) policy to allocate its kernel memory from a fixed-size memory pool setup at boot-time. The L4 kernel provides address-space mechanisms that can be used to partition the regions of physical memory a user-level component can access through virtual memory. But, since the kernel uses a different policy when providing its services to these components, the kernel cannot be used to enforce partitioning faithfully [BA03] — a malicious component can launch a denial-of-service (*DoS*) attack against the system by exhausting the kernel memory pool. A similar FCFS policy is used by the *Asbestos* [EKV+05] kernel, making it vulnerable to DoS attacks by kernel resource exhaustion. The defect was noted and fixed in [ZBWKM06] by changing the kernel's memory management scheme to a *caching* [CD94] scheme. Caching schemes guards against DoS attacks, but are not suitable for real-time systems — caching schemes make temporal guarantees difficult, if not impossible.

One possible solution is to modify the kernel's memory management policy to match the requirements of the application domain. However, modifying the low-level kernel code to modify its memory allocation policy is not ideal for any kernel, and in particular would require a significant amount of effort in the case of a formally verified kernel. This is because changing the code invalidates the refinement proof and as I explain in Chapter 3, changing the lowest level of kernel functionality (such as its memory management) requires a significant effort in reestablishing refinement.

Existing approaches to principled kernel memory management can be broadly categorised into two classes: those that seek to impose a limit on memory consumption and

those where the kernel memory is treated as a cache of system state. Both approaches are sufficient and even preferable in some circumstances but, quota-based limits provide little control over resources once allocated and are generally inefficient in dynamic systems (overall efficiency can be improved significantly by re-assigning resources to where they can be utilised [Wal02]), and as mentioned earlier, caching-based schemes are not suitable for the real-time domain.

1.4 Protection Models

A number of existing OSes provide protection models for reasoning and enforcing different policies, with varying degrees of assurance on the enforcement. The level of assurance may vary from an informal argument to a rigorous formal protection model of the system [SSF99, VEK+07, ZBWKM06] and a formal proof of the enforcement [SW00]. However, these protection (or security) models only capture, and hence the reasoning and the enforcement is limited to, the API level. The connection between the protection model and the kernel's internal resource management is either non-existent, insufficient or unclear. The kernel implementation is almost always presented as a "black box", with the assumption that its internal behaviour does not undermine the policy (say isolation) enforced at the application level.

1.5 Aim and Scope of Research

The ideal kernel memory management model for a formally verified kernel should be capable of enforcing different policies, as warranted by the application domain, including domains where temporal predictability is a central requirement, and this should be achievable without modifying the verified kernel code base. It must be possible to support diverse, co-existing policies in the case where the system is composed of heterogeneous applications. The model should facilitate fine-grained revocation and reallocation of memory as and when the need arises. At the same time, the kernel memory management model needs to be amenable to poorly scaling formal methods, so that one can reason about and formally affirm the enforcement of a particular policy.

The aim of this research is to design, implement and evaluate, both the formal and empirical characteristics of a unified protection and kernel memory management model. Unifying memory management with the protection model enables one to design and reason about systems constructed on top of the kernel within a single model that encompasses memory usage and access control. If the (unified) protection model has a formal decision procedure, then one can make formal statements about the ability of the model to enforce desired policies such as isolation. The memory management model should be general enough to support a range of complex applications such as virtualisation and applications with temporal demands and should have comparable performance to other systems to promote wide spread usage.
The design requirements for the model, therefore, are as follows:

- there should be a clear and precise relationship between the kernel's memory management scheme and the externally-visible protection model;
- the protection model should have a formal decision procedure that evaluates the ability (or the inability) to enforce different policies;

- it should be possible to enforce diverse policies concurrently over the management of kernel memory without modifying the verified kernel and
- the performance of the model should be comparable with other kernel memory management schemes.

The main barrier to providing a precise relationship between the protection model and the kernel's internal memory management scheme is the *implicit* memory allocations that take place within the kernel. Implicit allocations here refer to the memory allocated by the kernel as a side-effect of providing a service. The occurrence and the size of such allocations are not directly related to the authority to obtain the service but rather to the implementation of the service. For instance, the authority to a frame of physical memory has no direct relationship to the amount of memory the kernel may allocate in terms of page tables for mapping the frame, which depends on the implementation of virtual memory.

In general, protection models capture the authority distribution of the system and its dynamics (how the distribution changes over time). A component receiving the authority to a memory frame is captured by the protection model, but owing to implicit allocation, this authority has a loose relation to the amount of kernel memory the component may consume. Ideally, it should be possible to reason about the in-kernel memory consumption of a component based on the authority it possesses.

The challenge in establishing an analysable relationship between the authority and the kernel memory a component may consume, is to eliminate *all* implicit memory allocations from the kernel, and make all allocations explicit via the authority possessed by the component. This reduces the issue of in-kernel memory management to that of managing the dissemination of authority.

Capability-based authorisation schemes and their properties on controlling authority dissemination are well studied in literature. When coupled with a suitable transfer mechanism such as *take-grant* [LS77], these systems yield a decidable protection model. As mentioned earlier, decidable models facilitate reasoning about the amount of authority an application may obtain in the future by analysing the initial authority distribution.

While the primary concern of capability-based protection systems studied in literature is controlling access to user-level resources, there is a natural synergy between their formal properties and what is desired for controlling the explicit "allocation" authority. This work leverages the above similarity via a novel proposal of extending the take-grant capability protection to control the dissemination of explicit "allocation" authority.

An important consideration when extending any protection model is the effect the extension has on the decidability. It has long been recognised that protection models are generally undecidable [HRU76]. Only a small subset of models are decidable, and for some of the decidable ones, the decision procedure is NP-hard [San92a, ZLN05].

The classic take-grant model falls in the decidable category. Moreover, the decision procedure has a linear complexity [LS77]. This thesis demonstrates that these desirable properties of the take-grant model are preserved with the proposed extension.

Another important consideration here is the relationship between the protection model used in the analysis and the actual kernel code. Formally speaking, the results of the analysis is valid for the protection model, but not for the actual kernel unless the kernel and the protection model are formally *connected* to one another. However, as mentioned earlier, establishing such a relationship is beyond the scope of this thesis.

In the published literature, to date, specification and analysis of protection models has been done manually, using pen and paper. Yet, there is a large amount of research effort in

developing machine-assisted theorem provers. One of the goals of this research is therefore, to validate the feasibility of specifying and analysing a protection model in a machine-assisted theorem proving environment. However, there are a few past and ongoing research projects that either use, used or hinted at how to use machine-assisted theorem provers for safety analysis. In particular *PSOS* (Provably Secure Operating System) [FN79, NF03], *VFiasco* [HT05b] and *Coyotos* [Coy] (the successor of EROS [SSF99]) projects are directly related to the formal work presented here, since their primary security mechanism is capabilities.

While there are a number of policies one would like to enforce over the execution of user-level components, the analysis in this thesis is limited to two commonly used policies — partitioning [Rus99] and isolation.

Finally, this research explores the performance characteristics of a kernel with no implicit memory allocations and its effect on the kernel-provided services in both, a micro- and macro-level. Performance measurements were carried out on a prototype microkernel called *seL4::Pistachio* that implements the proposed memory management scheme and the associated protection model. Macro-level performance is evaluated by using seL4::Pistachio as a virtualisation platform for hosting a para-virtualised Linux instance.

1.6 Contributions

The primary contributions of this work are:

- a design of a dynamic, general purpose memory management model which supports high-end hardware features such as MMU with no implicit memory allocations within the kernel;
- a study of the feasibility of using capabilities to confer authority over all kernel memory allocations and its effect on the formal protection models;
- the design of a unified, decidable protection model based on take-grant, that facilitates reasoning about in-kernel memory consumption of a component;
- a machine checked specification of the above developed protection model and a machine checked formal proof of the model's ability to enforce spatial partitioning and isolation;
- a machine-checked proof of spatial partitioning and a machine-checked proof of isolation based on the protection model, together with a discussion on how to enforce the proved policies in the concrete system;
- a prototype implementation of the proposed design based on L4;
- an evaluation of performance, both at micro- and macro-level, of a kernel without any implicitly allocated kernel memory and
- a comparison of the cost associated with enforcing spatial isolation by centralised authority vs. controlled distribution.

1.7 Overview of this thesis

The remainder of the thesis is as follows. Chapter 2 examines the literature on kernel memory management and protection models, with the aim of identifying the merits and demerits of existing methods. Chapter 3 motivates the proposed design in the context of a formally verified, general-purpose microkernel targeted towards security applications.

A high-level design of the memory management scheme and the associated capability protection mechanism (hereafter called the *seL4 model*) is presented in Chapter 4.

Chapter 5 demonstrates how the proposed memory management scheme and its security mechanism can be modelled as an abstract specification (protection model) within a formal framework. This specification is developed using the machine assisted theorem prover Isabelle/HOL [NPW02]. It also discusses informally, how the protection model relates to the kernel implementation.

Then Chapter 6 demonstrates that the protection model developed in the previous chapter is decidable and through formal proofs show how it can be used to reason about the ability to enforce two different policies. The example policies considered are partitioning and isolation. The chapter provides a formal, machine-checked, proof that these mechanisms are sufficient to enforce the above two policies and demonstrates through a set of formal examples how to bootstrap a system so that it guarantees the policy enforcement.

Next, Chapter 7 introduces the *seL4::Pistachio* microkernel — a prototype realisation of the proposed model on modern hardware — and evaluates the performance of kernel-provided primitive operations. Chapter 8 further analyses the performance of seL4::Pistachio as a virtualisation platform. Finally, conclusions drawn from this work are presented in Chapter 9.

Chapter 2

Related Work

In this chapter, the literature on different aspects related to the work presented in this thesis is reviewed and discussed.

I present this discussion in two sections. First, Section 2.1 reviews research related to kernel memory management. Second, Section 2.2 examines the research in the area of security control models, including those used in capability-based systems, and the related formalisms used for reasoning about the system behaviour.

The ideal kernel memory management model for a formally verified kernel should be capable of enforcing different policies, as warranted by the application domain, including domains where temporal predictability is a central requirement, and this should be achievable without modifying the verified kernel code base. It must be possible to support diverse, co-existing policies in the case where the system is composed of heterogeneous applications. The model should facilitate fine-grained revocation and reallocation of memory as and when the need arises. At the same time, the kernel memory management model needs to be amenable to poorly scaling formal methods, so that one can reason about and formally affirm the enforcement of a particular policy. In Section 2.1, I critically analyse the existing work with regards to the above mentioned requirements.

There is a large amount of literature on protection models (a.k.a. security models) and their formal properties. Unfortunately, the published work, to date, has not considered the application of security models to control or reason about the in-kernel memory allocations. However, some models, in particular the ones used in capability-based systems and the associated reasoning techniques show evidence that they can be modified or extended for this purpose. Section 2.2 reviews these models in terms of their applicability as a model for reasoning about kernel memory.

2.1 Kernel Memory Management

Existing approaches to in-kernel memory management can be broadly categorised into three main classes: (a) those where memory management is governed by an in-kernel policy, (b) those where the kernel's memory is a cache of the system state and (c) those that seek to export some sort of control of in-kernel memory management to user-level applications.

The above categorisation is not strict. It simply aids the clarity of presentation. In the following sections, I examine the work done on each of the categories.

2.1.1 In-Kernel Policy

In this category, the kernel memory is governed by a built-in policy that resides within the kernel. This built-in policy can either be static (i.e. cannot be modified without modifying the kernel code) or it can be extended or modified by uploading code into the kernel.

The Linux (2.4) memory manager [vR01] manages multiple pools of frames called *caches*. These caches include the buffer cache, the inode cache and the slab cache. The kernel uses a buddy system to allocate frames for these pools. Free frames are dynamically reassigned from one cache to another, thus making the caches grow and shrink on demand. Resources from these caches are allocated on a first-come-first-server (FCFS) basis and when there are no free frames left in the system, the kernel randomly terminates tasks to free memory.

The FreeBSD [Fre06] kernel, similar to Linux, uses a buddy system to manage its frame-level allocations. However, unlike the slab allocator [Bon94] used in Linux, FreeBSD uses a *zone* allocator. The zone allocator runs on top of the buddy-based frame allocation layer. Kernel modules register with the zone allocator, specifying the range of memory sizes (or zones) they are likely to request during run-time. Based on this specification, the zone allocator grabs frames from the frame-allocation layer and carves them into smaller "ready-to-go" chunks. Most run-time memory requests are satisfied from these pre-allocated chunks. A garbage collector running in the background, searches for free frames in the zone and returns them to the frame allocation layer, which can then allocate the freed frames to a different zone. Kernel modules have the flexibility to use either the zone allocator or the frame-allocation layer directly. A comparison between the Linux and FreeBSD memory management subsystems can be found in [Dub98].

Early versions of the L4 kernel [Lie95] allocates metadata on a FCFS basis from a global, fixed-size memory pool created at system start-up by *sigma_1* — the kernel pager. Once the pool is exhausted, no new metadata is allocated and respective system calls fail with an error code. While there is no published information to date on the subject, browsing the source reveals that a similar scheme is used in Asbestos [VEK+07] (the observation is also made in [ZBWKM06]). FCFS is a simple and hence easy to implement and verify policy, that works well if all applications are of the same level of trust or are of equal importance to the system. However, in addition to opening a covert communication channel, the scheme is vulnerable to denial-of-service (DoS) attacks from applications monopolising kernel memory and therefore cannot enforce strong spatial partitioning [BA03] or be used in a system designed to execute untrusted and potentially malicious, downloaded web contents [LIJ97].

Later versions of the L4 API [NIC05] rectified the situation by making kernel services that require the allocation of kernel memory privileged — any kernel service that may require the allocation of kernel metadata must be made through, and therefore monitored by the privileged server. As such, the privileged server is in a position to monitor and limit the kernel memory consumption of an application. This makes it possible to enforce different kernel memory allocation policies by modifying the privileged server rather than the verified kernel. However, the privileged server has no control over the regions of memory used by the kernel. Thus, the scheme only provides mechanisms for limiting the kernel memory consumed by an application rather than controlling.

The ability to limit the kernel memory consumption alone is sufficient in some circumstances, but not in some others. For example, authors in [LHH97] studied the use of CPU cache-colouring techniques for improving the temporal predictability of real-time systems. Verghese et al. [VDGR96] report significant performance improvements by localising data

to the local memory of a cache-coherent non-uniform memory architecture (CC-NUMA) machine. The Solaris *Memory Placement Optimisation* (MPO) [MPO08] project applies similar techniques to Solaris to improve performance on NUMA machines. A detailed assessment of the importance of memory placement in NUMA machines can be found in [AJR06]. Enforcing such schemes requires more control than limiting — the privileged server should have the capacity to select the placement of kernel memory depending on the requirements of the applications.

The concept of using a privileged server for governing allocations can be found in several other systems. The *System Domain* of *Nemesis* [Nem00], *Domain0* in *Xen* [BDF+03] and *Supervisory VM* in *Denali* [WSG02] serve similar purposes. However, note that the Nemesis System Domain is a much more powerful concept than its L4 counterpart, in that it has control over where to allocate kernel memory and these allocated regions are used by the kernel without further checking — the kernel trusted the System Domain. In that sense the System Domain is part of the kernel, running in a different protection level of the processor.

Several other approaches introduced mechanisms to control resource consumption of an application directly, without mediation. *Scout* [SP99] accounts resources towards a special abstraction called *path*, and limits the resource usage. A path represents a stream of data flowing through the system via several subsystems. Banga et al. [BDM99] introduced *Resource Containers* for accounting and controlling the consumption of resources. *Virtual Services* [RMSK00] implements fine-grained resource accounting even in the presence of shared services by intercepting system calls and using classification mechanisms to determine the correct resource principal to whom the resources are accounted to. The Solaris resource manager [Inc] introduces limit nodes, or *lnodes*, which support user- or group-based resource control. These approaches only support resource limits for principals.

The Solaris *Zones* [PT04, TC04] technology introduced *resource pools* for partitioning system resources. Resource pools allow a collection of resources to be exclusively used by some set of processes. The current implementation of resource pools only include CPUs [PT04] (on a SMP), with a plan of extending the concept to partition other resources, including memory. The *Linux-VServer* [dL05] project uses virtualisation technology to isolate applications. Published literature however, does not provide sufficient details on how the scheme handles memory, in particular when used within the kernel. These approaches support, to various extents, isolation of components.

The *SPIN* system [BSP+95] allows uploading of code at run time to the kernel and thereby can change the system policy. This is achieved by installing a *spindle* — application-specific code written in a type-safe language. Even though spindles execute with kernel privileges, they cannot interfere with the rest of the system. The SPIN system achieves this property through a combination of run-time checks and compiler techniques. A spindle can only introduce new functionality based on the *core services* — a part of a framework for managing memory and other resources, implemented by the kernel itself. A spindle can, therefore only "fine-tune" the policy rather than modify it completely. For example, important features of memory management such as placement of memory, revocation and replacement cannot be influenced by the uploaded spindle. Moreover, the memory management policy is global, which may not suite all applications on a heterogeneous system.

The *VINO* kernel [SESS96] facilitates an application to customise the kernel resource management by *grafting* an extension into the kernel. Similar to SPIN, the amount of influence a VINO graft has on the underlying kernel memory allocator is limited. In contrast to SPIN, VINO grafts are not written in a type-safe language. The kernel uses run-time

checking to protect against misbehaving modules. These run-time checks are expensive compared to the actual extension [SESS96, HHL+97].

Both VINO and SPIN achieved extensibility by inserting modules into a monolithic operating system (*OS*). An alternative approach is the use of multiple OS serves [Lie95] in a microkernel based system. Each application, depending on its requirements, obtains services from a different OS server. OS servers themselves are protected from one another, thus removing the need for run-time checks required to protect the monolithic OS from misbehaving extension modules. Moreover, the sever approach yields better performance compared to the extension modules [LES+97, EH01].

The *K42* kernel [IBM02] takes advantage of C++ inheritance to control the behavior of the underlying memory allocator. Thus, by recompiling the kernel, its memory allocator can be specialised to meet particular application requirement.

Analysis

In this class of systems, kernel memory management is governed by a policy integrated into the kernel. This built-in policy can either be static in the sense that the policy cannot be modified without modifying the kernel, or it can be extended by uploading code. Uploaded code however, has limited capacity to modify the kernel's memory management policy. Thus, in this class of systems, the policy governing the management of in-kernel memory is fundamentally fixed and can only be changed by modifying the kernel.

A policy is designed either to suit the needs of a particular application scenario, or it's a trade-off between generality and meeting some specific requirements. For this discussion, I call the former type of policy *specific* and the latter *generic*. Specific policies, obviously work well for the domain it was designed for but not with another where requirements are different. Generic ones on the other hand, are suboptimal in any situation. In either case, a single policy, be it specific or generic, does not best suite all application domains of the kernel.

At one extreme, the requirements of the application domain can be strict partitioning and hence a strict partitioning of resources, including the kernel memory regions. At the other extreme the system can be a best-effort, where kernel memory allocation is on-demand and targeted to maximise through-put. A single memory allocation policy can serve either one of these situations, but not both as there are fundamental trade-offs between the two extremes. For example, the FCFS policy employed by L4 and Linux is sufficient for the latter case, but not for the former [BA03]. The Solaris Zones [TC04] technology suits the needs of the former domain but is inefficient for the latter— efficiency can be improved significantly by reassigning memory to where it can be utilised [Wal02]. Another example is the variants of quota-based policy used by most systems attempting to limit the memory consumption — they can either provide predictability through reservations, or better utilisation, but not both.

It is long being established that the default resource management policy of an operating system often yields suboptimal application performance [AL91, Sto81]. Applications can benefit significantly by switching to a fine-tuned resource management policy that suites their particular needs [EGK95, Han99]. This stems from the fact that the default in-kernel policy is almost always generic, so that it is applicable to a wide range of applications, in contrast to an application-specific, fine-tuned policy.

In systems where the management scheme is integrated into the kernel, the only way to modify the kernel's in-kernel memory management policy is by modifying the kernel

itself. This option is not attractive for a verified kernel. This is because any modification to the kernel will nullify the refinement proof. Changing the lowest-level of behaviour; such as that of the heap, requires a significant redo of the proofs (I quantify this claim through a case study in Chapter 3).

Few systems provide the capability to extend or modify the memory management policy. However, the scope of such extensions is limited. Moreover, the ability to extend or modify the low-level functionality of the kernel (like its memory management subsystem) comes at the cost of increasing the kernel complexity and hence the degree of difficulty involved in formal verification. For example, the VINO kernel uses run-time checks to protect against misbehaving grafts. To verify the kernel one needs to model all these run-time checks, prove the sufficiency of these checks to protect the kernel and show refinement from the abstract model to the code implementing these checks. This process adds significantly more complexity to the poorly scaling and labour intensive verification process. Moreover, given the limited scope these modules have on effecting the kernel memory management policy this complexity is unjustified.

2.1.2 Kernel Memory as a Cache

In this approach the kernel's internal data structures are viewed as a cache of the system state which is generally managed outside of the kernel. When the cache is full the kernel evicts an object to make room for another. Variations of this core idea are used in several kernels.

In the *V++ Cache kernel* [CD94] the management of data structures required for implementing the core kernel abstractions is done outside of the kernel by *application kernels*. The core abstractions of the cache kernel include threads and address spaces. An application kernel is responsible for managing and loading the objects into the kernel's cache when they are required. When its cache is full, the kernel evicts objects from the cache and writes them back to user-level. A small number of objects, however, can be *locked* in the cache. These locked objects ensure the execution of fault handling code (in particular code that loads objects) without incurring further nested faults.

The *KeyKOS* [BFF+92] nanokernel employed a similar mechanism for managing its kernel memory. However, unlike the cache kernel, evicted objects are written back to protected disk blocks.

The *EROS* [SSF99] kernel — the successor of KeyKOS — implements an abstract virtual-machine based on type-safe capabilities. Kernel metadata such as page tables, process descriptors, are constructed from the capabilities stored in type-safe user-level memory [SFS96]. The kernel improves performance by avoiding the need to validate these user-level data structures on every access by caching the results. When the kernel's cache is full, the object is written back to the user-level data structures and the contents are evicted from the kernel's cache.

HiStar [ZBWKM06] is a persistent system with a single-level object store. It manages kernel memory as a cache of objects in its single-level storage.

Aegis Exokernel [EGK95] uses a notionally similar technique for implementing virtual memory (VM). The kernel runs on the MIPS architecture which has a software-loaded TLB. Upon receiving a page-fault the kernel propagates it to the *application-level* VM (AVM) handler which maintains page tables for the address space. The AVM generates the appropriate TLB entry and calls the kernel which validates and installs the entry. To absorb the capacity misses, Aegis overlays the hardware TLB with a software TLB (STLB)

[BKW94].

In [RF97] Reed et al. leverage the single address space structure of Nemesis to propose a conceptually similar scheme for managing virtual memory. The kernel maintains a single page-table structure for the entire system. Each domain in the system has a default privilege (read) for each page, which is set up by this structure. When a domain takes a page-fault, the handler checks the current protection domain and if the domain is allowed additional access to that page it modifies the page-table structure and puts the page into a list of altered pages. When a protection domain switch occurs, the altered pages list is scanned and the modified entries are fixed to reflect the default access rights.

Analysis

In this class of systems, kernel metadata is stored and maintained at user-level where it is subjected to user-specific management policy or in secondary storage. Hence, except for the cache replacement policy, no separate policy is required for managing the kernel metadata.

The drawback of this approach is that applications compete for space in the kernel cache. As such, guaranteeing predictable execution time for real-time applications become difficult, if not impossible. A malicious application can easily thrash the kernel cache and thereby degrade the performance of others. Even though there is the possibility to incorporate the thrashing of kernel cache into the execution time analysis of the real-time application, result would be too pessimistic to be useful.

2.1.3 User-Level Management

In these systems, the actual memory consumed by the kernel is managed by user-level applications. One can modify the in-kernel memory management scheme by modifying the user-level application rather than the verified kernel.

The *system domain* of Nemesis manages all memory allocations, including kernel memory. For example, memory required for page tables is allocated by the *high-level translation* system [Han99], which is a part of the system domain. User-level applications request *stretches* — a contiguous region of virtual memory—from the system domain. To create a stretch, the system domain allocates and initialises the required page tables which are then used by the in-kernel or the low-level translation system.

Liedtke et al. [LIJ97] proposed an extension to the fixed-size memory pool used in the original L4 microkernel. In this scheme the kernel still allocates metadata from a central in-kernel pool until it is exhausted. Similar to the original model, once the pool is exhausted subsequent system calls fail. However, user-level memory managers can *donate* some of their own memory to the kernel memory pool and thereby rectify the situation. Before donating a page to the kernel, the donor must remove the donated page from all user address spaces and in doing so, it loses the control over the donated page. Thus, the user-level memory manager cannot reclaim donated memory. These donated pages are then used by the kernel to allocate metadata for the donor.

The *Calypso* translation layer [Szm01] — a virtual memory subsystem for the L4 microkernel— extends the above donor model in two aspects. First, it uses per-task kernel memory pools as oppose to a central one. Second, it does not fail system calls upon the exhaustion of a kernel memory pool. Instead, the requester is suspended and the fault is propagated to a pager which can then resolve the fault by donating memory to the kernel.

In both the above schemes, the pager loses control over the donated page, thus cannot reclaim the memory. Haeberlen and Elphinstone [HE03] further extended the scheme by making the "donated" pages reclaimable. When the kernel runs out of memory for a thread, it propagates the fault to the corresponding *kpager*. The kpager can then map any page it possesses to the kernel, and later preempt the mapping. However, the kpager is not aware of, and cannot control, the type of data that will be placed in each page and thus can not make an informed decision about which page to revoke when the need arises to reclaim memory to use in a different context.

The *L4.sec* kernel [Kau05a] divides kernel objects into first-class (addressable via capabilities) and second-class objects. Both classes requires a *kernel memory object* — a region of physical memory out of which other objects can be allocated, to provide the memory required for creation of objects. L4.sec does not allow direct manipulation of second-class objects such as page tables or capability tables (CNodes). As such, dynamically reallocating page table memory from an idle task to another task is not possible without destroying the former.

The virtual machine monitor *Xen* [BDF+03] uses a similar mechanism for manging its page tables. Before using a frame as a page-table the guest OS needs to notify Xen of this fact, after which the guest OS loses its "write" privilege over the frame. All subsequent updates to the page table must be made through and therefore validated by Xen. Obviously, if a frame is registered as a page table, it should not appear as a page frame in a different context. Xen enforces this by tracking the number of references made to a frame in a frame table — a frame can change its type only when its reference count is zero.

The *Fluke* kernel exports the state of kernel data structures to user-level applications [TLFH96]. However, the kernel memory consumed by these structures itself cannot be controlled — primary focus of Fluke is exporting the state of kernel data structures to facilitate user-level check-pointing rather than controlling in-kernel memory consumed by these data structures.

Analysis

Exporting the control of kernel memory management to user-level applications via a secure interface is attractive for several reasons. First, it enables user-level applications to use their own fine-tuned, hence efficient, management policy. Second, it facilitate the co-existence of diverse management policies by means of different user-level managers.

However, existing approaches do not fully satisfy the requirements we are looking for. In some cases, they do not facilitate revocation at all (in the case of "donate" model and Calypso). The ones that support revocation either do it at a coarse granularity (in the case of L4.sec) or do not provide sufficient information to the user-level manager to make an informed decision on what memory to reuse (in the case of kpager model). Moreover, the performance of these approaches is also open for discussion. I explore their performance later.

2.1.4 Summary

Table 2.1 summarises the main points of our discussion thus far. It shows general properties of each existing kernel memory management scheme. In-kernel policies cannot be modified without modifying the kernel and hence invalidating the refinement proofs. State caching schemes resolve the issue to a certain extent, but not entirely. The issue is resolved

Scheme	Domain specific tuning	Co-existing policies	Fine-grained revocation	Formal reasoning
In-kernel policy	by kernel modifications	no	yes (depends on policy)	no
State caching	yes (not suited for real-time)	yes (to some extent)	no	no
User-level management	yes (kernel unchanged)	yes	no	no

Table 2.1: Summary of the properties of existing kernel memory management schemes.

by placing the management of kernel data structures in user-level applications and thereby facilitating policy modifications via modifying user-level code rather than the verified kernel. Diverse policies can co-exist in the same system by means of different user-level code managing-kernel data structures. It is not entirely satisfactory because the kernel uses an in-built policy that views its metadata as a cache of these user-level structure. This in-built policy works well for a best-effort system but not ideal for systems requiring predictable execution times.

Out of the existing approaches, user-level management satisfies most of our requirements. One can modify the policy by modifying the user-level manager rather than the kernel. It supports diverse policies simultaneously via different managers. Currently, none of the existing user-level management schemes support fine-grained revocation and reuse of memory. This, however, is a limitation in the design/implementation rather than a conceptual one — there is no fundamental reason to avoid fine-grained revocation and reuse of memory within this scheme. As I show in this thesis, one can achieve fine-grained revocation and memory reuse without a significant loss of performance.

None of the existing user-level management schemes facilitate formal reasoning. Ideally one should be able to affirm that a particular policy is enforced — it should be possible to provide compelling evidence that a given user-level resource manager cannot violate the system-level policy (or a subsystem-level policy for that matter) in any system state. The policy here can be, for example, spatial partitioning [Rus99], and by compelling evidence I mean a formal proof based on a formal model of the system, that the given resource manager cannot violate the spatial partitioning policy in the current or in any derived system state.

User-level management schemes have an attractive property that may make it possible to bridge this gap. In this scheme, since kernel memory is managed by user-level applications (in contrast to opaquely, within the kernel), it naturally becomes a first-class resource of the system. In other words, one can view kernel memory as a system resource a user-level application can "access" in much the same way as having access to a file or a frame of memory. The main difference between the kernel memory resource and the traditional resources (say the file) would be the meaning of "access" — for the file it can mean read or write permission and for the kernel memory it can be the permission to "consume" or "release" kernel memory. This is simply a matter of how one interprets the permission.

In this manner, we can treat regions of kernel memory as a first-class resource governed by the access control mechanism of the system. Then, we can convert the question of policy enforcement to that of access control. For example, the question of whether or not the partitioning policy holds comes down to the ability of a partition to obtain access to

a region of kernel memory it did not have initially from another. This type of questions, termed *safety questions* [HRU76], is generally answered using a formal *protection model* (a.k.a. security model) of the system. The protection model of a system is an abstract, formal model that captures the authority distribution of the system together with the rules used to control their dissemination.

Published work in the area of protection models has not investigated the possibility of using them to reason about kernel memory. However, some models are general enough and in particular the decision procedure used in the analysis is such that they can be extended for this task. In the next section, I investigate formal security models to understand their suitability to use as a framework for reasoning about policy enforcement over kernel memory.

Though it is not central to the topic of this theses, it is worth noting that not all policies can be enforced via execution monitoring [Sch00]. To be enforceable using access control the policy needs to be a *safety property*. Without going into formalisms, a safety property has an exact point at which the system goes into an unsafe state. The partitioning policy we looked at above, for example, is a safety property, because these is clear point at which partitioning is violated (when one partition gets an access right it did not have from another), and hence the system is in an unsafe state. Availability, if taken to mean that resource access will be granted eventually, is not a safety property. Because of the term "eventually" one cannot determine when the system is in an unsafe state. The usual solution in such a scenario is to use an approximation of the original policy which itself is a safety property. For example, Giligor [Gli84] defined availability as resource access must be granted with a (some) fixed time.

2.2 Formal Modelling

A formal framework for reasoning provides some clear benefits. As an illustrative example, one should be able to answer questions such as "can application *A* access a given region of kernel memory?" and "can *A* gain access to more kernel memory than what it has now?" and if so "from whom?" and so on. These type of questions are called *safety questions* and formulating an answer to a safety question is called *safety analysis*. Safety analysis is done using a formal *protection model* (a.k.a. security model)— an abstract formalism capturing the *protection state* of the system and the rules governing it's mutations. The protection state essentially capture who has access to what resources and the mutations define how the protection state can be modified. I define the above terms in Section 2.2.1. Note that the above definitions are just to facilitate our discussion.

As mentioned earlier, one may treat kernel memory as a resource much the same way as a file or a frame. The protection state of the system may authorise an application to read or write to a frame and the application may use this authority in a manner it sees fit. A region of kernel memory may be viewed in much the same way — authority in this case would be to "consume" or "release" the resource. The application may exercise this authority in any way it chooses. Since the protection state can evolve, one may change the amount of kernel memory a subject may use by adjusting its authority. Moreover, one *may* perform safety analysis to identify the feasibility of an application obtaining "consume" authority over a region of kernel memory.

Unfortunately, formal security models do not capture kernel memory consumption. Thus, they cannot be used to reason about nor can they be used to control the kernel memory consumption of an application. The primary concern of these formal protection models

is controlling the access to user-level resources and as such have no notion of kernel memory.

From a safety analysis point of view, there is a difference between the authority to "consume" kernel memory and the authority to a user-level frame. This difference stems from the ability to perform *create* operations. An application possessing "consume" authority to a region of kernel memory can exercise it's authority and create a new entry in the protection state (say a thread), as opposed to reading or writing to the frame. These newly added entries complicate the safety analysis for the following reason: the safety analysis, by looking at the initial state, tries to determine whether a particular access right can be leaked from one application to another. Create operations complicate this process because they add new entries that we did not know of in the initial state, that might affect the final outcome. Most safety analysis techniques in the published literature make (very) strong assumptions on how and in particular when a create operation can be done. For example, a common technique is to assume that all create operations take place first and not after that [San88, SS92]. Such an analysis is not suitable for a security model attempting to control the ability to create in a dynamic fashion — the ability to create is based on the authority distribution which evolves.

However, by understanding the existing models, the limitations and in particular analysis techniques and how the create operation is handled, I aim to extend the models and the analysis to suit our needs.

The remainder of this section is organised in the following manner. Section 2.2.1 provides the background on access-control models. Then in Section 2.2.2, I critically analyse the existing security models.

2.2.1 Background

Access control is concerned with the question of who can access what resources in the system. The same resource, say a file, may be accessible to different users in different ways. Some users may have read and write access to the file, while some others may only read, and so on. Strictly speaking, users do not access system resources — these resources are accessed by programs running on behalf of a user. In the classic sense of access control, a program running is called the *subject* and system resources a called *objects*. So, access control is concerned with enforcing authorised access of subjects to objects. This basic idea was introduced by Lampson [Lam71]. He proposed an *access control* matrix, a subject $\times$ object matrix. Each cell in this matrix contains the access rights the corresponding subject has over the object. In essence, the access control matrix represents the protection state of the system — what access a subject is allowed on an object. As the system evolves, so does its protection state. For example new subjects or objects might be created, access rights might be removed from an existing subject or the subject itself might be deleted and so on. The *access control model* defines a set of rules or *commands* by which the protection state of the system can be changed.

The challenging aspect of access control is its dynamics. Starting from some protection state and particular access control model can one reason about the future protection states? In particular, "can some subject X obtain some access right α to resource (object) Y?". These type of questions are typically called *safety* questions [HR75]. Note the generality of the safety question — depending on the context, one can select a subject and a particular resource. For instance, in the context of kernel memory management, the resource of interest would be a region of kernel memory and the access right would be the subjects

authority to consume it.

The protection state and the access-control model is collectively called the *security model* or the *protection model* of the system. It describes the current access rights distribution of the system and the set of rules that can be used to manipulate the distribution. The security model of the system is used for the *safety analysis*; i.e. the process of formulating an answer to the safety question. If there exists an algorithm for safety analysis then the security models is said to be *decidable*. Note that decidability does not necessarily mean that safety analysis is feasible. It simply means there exists an algorithm — which itself can be intractable.

2.2.2 Classical Access-Control Models

The use of abstract formulations of the security system for safety analysis has a long history. In 1976, Harrison et al. [HRU76], in a model known as *HRU*, first formulated the safety question and proved that in the general case, safety is undecidable. However, the safety question is decidable for *mono-operational* systems — systems in which every command does a single operation. In later work, the same authors [HR78] showed that safety is decidable for *monotonic* (i.e. contains no *delete* or *destroy* operation), and *mono-conditional* (i.e. all commands have at most, one clause), systems. In the presence of *bi-conditional* (i.e. has two clauses), commands even a monotonic system becomes undecidable [HR78]. These results from the HRU model are disappointing in that the protection system either does not facilitate reasoning (i.e. undecidable), or in the case where it does it is too restrictive to be useful.

Since the introduction of HRU, a number of access control-models for which the safety is decidable have been proposed — the *take-grant* model [LS77], *Schematic Protection Model (SPM)* [San88], *Type Access Matrix Model (TAM)* [San92a], *Dynamically Typed Access Control (DTAC)* and *non-monotonic transformation model* [SS92] are decidable protection models.

The *take-grant* (TG) model initially proposed by Jones et al. [LS77], later enhanced by Snyder [Sny81] and many others, falls outside the known decidable cases of HRU (mono-operational), however there exists a linear-time algorithm for deciding the safety. While there are number of variants to the classical model, in general, the model represents the system state as a directed graph where vertices are either subjects or objects and the outgoing, labelled, edges denote the authority the source of the edge has over the destination. The access control model is defined as graph rewriting rules. In general there are four graph rewriting rules — the *take rule* allows a subject to acquire access rights from another, the *grant rule* allows a subject to propagate access rights to another, the *remove rule* facilitates removing access rights and the *create rule* allows *every* subject to create new nodes in the graph. Conceptually, creation of a new node in the abstract model corresponds to allocating physical memory within the kernel. Unfortunately, under the classical TG model, the semantics of the create rule is such that every subject possess the ability to create without any restriction. As such, it is not directly amenable to controlling memory allocations within the kernel.

Even though safety under take-grant protection is decidable in linear time, the analysis requires knowledge of the entire system — it does not facilitate endogenous reasoning. In [LM82] Lockman and Minsky, showed that by removing the grant rule from the take-grant model, flow of authority can be made unidirectional. All authority transfers are authorised by the authority possessed by the receiver rather than the sender, thus providing

endogenous control — the ability to receive a capability is determined solely by the authority within the subject receiving it. In a similar manner, the *diminished-take* [SW00] model proposes filters on the take operation to enforce transitive read-only paths by the authority on the receiver-side. While desirable, endogenous control is orthogonal to our interest — a protection model for controlling the allocations within the kernel, which both the models do not provide.

Many authors have proposed extensions to the take-grant model. Bishop and Snyder [BS79] enhanced the take-grant model by introducing de-facto rules — rules that derive feasible information flow paths given the distribution of authority. The take-grant model captures the direct authority (*de-jure*) distribution and the de-facto rules compute the implied authority based on that distribution. For example, A can have a direct authority to write to a file and B direct authority to read from the same file. This distribution implies a de-facto write authority from A to B (through the file). Later work by the same authors [Sny81, Bis96] further enhanced the initial proposal. Wu [Wu90] and later Bishop [Bis81] applied take-grant control for a hierarchical protection system. Dacier [Dac93] formulated the classical take-grant model in terms of a *petri net* [DJ01] and proposed an algorithm to determine all the access rights a subject may acquire with the help of a given number of conspirators as opposed to the classical model where every subject is a conspirator. Frank et al. [FB96] extended the take-grant model by adding a notion of a cost of information or right flows and found the most likely path in terms of the costs. All these models extend the take-grant model and propose variants for safety analysis. However, none of these extensions have a notion of kernel resources and the analysis assumes all subjects have the ability to create without affecting another, thus they do not facilitate reasoning about the kernel memory consumption of a subject in particular as create works on a global, limited resource.

Even though take-grant is a purely theoretical model it has being successfully used to analyse practical systems. In [Bis84] Bishop demonstrated that with the proper extensions, the take-grant protection model can be used to examine realistic systems for security flaws. The diminished-take model was used to capture the operational semantics of the EROS kernel [Sha03] and to verify its *confinement* [Lam73] mechanism [SW00]. In addition to its use as an access-control model, more recent work has explored the possibility of using take-grant for analysing network vulnerabilities [SSJ$^+$05]. The wide-spread applicability of the model and consequently the extensive literature on different analysis techniques and various extensions to the basic model, make the take-grant model the ideal candidate for exploring the possibility of using it to control the allocations within the kernel.

Closely related to take-grant is the *send/receive* [Min84] transport mechanism. Similar to take-grant, the safety for this model is decidable, however it is more restrictive than take-grant in the sense that in addition to "Take" (receive) or "Grant" (send) channels, it is also possible to enforce additional restrictions on the type of privileges a subject may transport over these channel. In the analysis, authors showed that creation of new subjects does not increase the possibility of an access right leak between two existing subjects, as long as all subjects possess the ability to send and receive authority to/from themselves (such a state was called *uniform*). The analysis ignores the create operation assuming the initial state is uniform. The motivation behind removing create operation is to fix the number of subjects in the system. Excluding the create operation from the analysis limits its scope to the behaviour of subjects that are already in existence. However, the result is encouraging — since the creation of new subjects does not increase the possibility of an access right leak, controlling the create operation should not, at least in theory, break the decidability

property of the model (which is based on access-right leakages).

In general, the create operation is treated as the most complex operation in safety analysis. This complexity stems from the fact that it introduces new subjects that we do not know of in the initial state, yielding a potentially unbounded system [San88, AS91]. As a workaround, the general norm is to assume all possible create operations to occur first, which makes any subsequent create redundant and therefore can be ignored from the analysis.

Such a technique was used in the analysis of the Schematic Protection Model [San88] (or *SPM*). In SPM, subjects are associated with a static security type. Each type is allowed to create other types as defined by the *can-create* relationship. The model is only decidable for *acyclic* creates [San92b] — that is, if subjects of type a are allowed to directly or indirectly create subjects of type b, then it should not possible for subjects of type b to directly or indirectly create subjects of type a. The can-create relation is static, in that the types of subjects that can be created by another type, do not change as the system evolves. The static nature of the can-create relation and the acyclic creates are exploited in the analysis of SPM. All create operations are assumed to occur first. Each subject creates subjects of all possible type, and so do the newly created subjects. Once this state is computed, any subsequent create is redundant, thus the analysis focuses on *copy* (or grant) operations. In later work, Ammann et al. [AS90, AS91] proposed an enhancement to SPM called the *extended* SPM or *ESPM*, which yields a model that is formally equivalent to the monotonic HRU [HR78] model. Similar to the analysis of SPM, ESPM also made the assumption that all creates occur first in the system.

Sandhu introduced strong typing into HRU. Each subject or object is created to be a particular type which thereafter does not change (static typing). This new model is the Typed Access Matrix Model (TAM) [San92a]. Moreover, TAM allows to check for the absence of rights. An extension to TAM, called the *Augmented Typed Access Matrix (ATAM)* [AS92] allows checking for the absence of rights in the command. The notion of strong typing in TAM is analogous (not identical though) to *strong tranquillity* in *Bell-LaPadula* [BL76, BL73] style models. Strong tranquillity means that security levels associated with subjects and objects do not change during the lifetime [Bis03]. The monotonic TAM (MTAM) is decidable, but NP-hard. A simplified version of MTAM called, *ternary* MTAM is decidable in polynomial complexity. Similar to the analysis technique for SPM, the analysis of these models assumes all creates to occur first.

Strong tranquility is somewhat restrictive. Yet, allowing subjects to change their type in an unconstrained manner has adverse effects on the system security [McL85, Den76]. There is however, a point in the middle where subjects are allowed to change their type, but in a manner that does not violate the security policy; known as the *weak tranquility*. For instance, the protection system of *Asbestos* [EKV$^+$05] allows subjects to change their security level, how ever in a contained manner. Such systems are flexible when compared with rigid strong-tranquility systems.

In a similar manner, the Dynamically Typed Access Control (*DTAC*) [TP98] model employed a dynamic type system. Similar to TAM, DTAC introduces typing into HRU. In recent work, the authors introduced a graphical way of representation and a constraint specification language [TP01]. As a result of DTAC's dynamic nature, it requires more run-time checks, analogous to type checking for programming languages.

Note that all the above work is done by using "pen-and-paper". In other words, they are not machine-assisted or machine-checked proofs.

2.2.3 Summary

The security model of a verified kernel should facilitate reasoning, in other words the model should be decidable. Out of the few decidable security models that exist, the take-grant model is the most promising candidate. It has being applied for a wide range of domains.

The model does not have a notion of kernel memory or the general notion of a limited resource — every subject can create nodes in the graph, without effecting another, and the analysis is closely related to this assumption. This means we cannot directly use the model to reason about kernel memory. However, related work indicates the possibility of improving — there is evidence to suggest that it is possible to extend the model, and thereby make it feasible to reason about kernel memory consumption. Moreover, since the model, as it is, can be used to reason about overt information flow (both de-jure and de-facto) the extended model should ideally facilitate both.

In Chapter 5, I explain how take-grant can be extended to facilitate reasoning about the kernel memory consumption. Chapter 6 shows how the extended model can be used to analyse kernel memory as well as overt information flows.

Chapter 3

Managing Kernel Memory

In this chapter, I argue that existing approaches to in-kernel memory management, while attractive in many circumstances, are inadequate for a general purpose, formally verified kernel and then motivate and rationalise the proposed memory management scheme.

The chapter is organised in the following manner. Section 3.1 investigates the work carried out by the *L4.verified* team to formally verify the seL4 kernel and discusses the additional requirements placed on the kernel design and in particular its memory management model. Then Section 3.2 analyses the existing approaches against these requirements. Finally, Section 3.3 introduces the proposed memory management model and the challenges in realising it.

3.1 L4.verified – Formal Verification

The term "formal verification" is highly overloaded — depending on the context, it may refer to model checking, safety analysis, or formal refinement. In the following sections, I outline the formal verification approach adopted by the related *L4.verified* [EKK06, EKD+07, KEH+09, Kle09] project and discuss its implications on the kernel design.

The end goal of the L4.verified project is to formally prove that the kernel's implementation is correct with respect to a high-level, abstract model of its behaviour. This abstract model is then used to reason about the ability to enforce a particular policy. The policy here can be, for example, spatial partitioning [Rus99], isolation, confinement [Lam73] and so on. The formal proof of correctness connects the reasoning at the abstract level with the deployed kernel.

3.1.1 Overview of the Methodology

A pictorial representation of the steps involved in verifying the implementation correctness of the kernel is shown in Figure 3.1. At the very top of the verification hierarchy is a *protection model* (or a security model), which provides a high-level abstract view of the security-critical functionality of the kernel. In particular it models the access-control mechanism of the kernel.

The main objective of the protection model is to facilitate formal reasoning — it acts as a framework for analysing the ability (or the inability) to enforce a given policy. Using the protection model, one can prove formal theorems about the policy enforcement. I call such an analysis a *security analysis*.

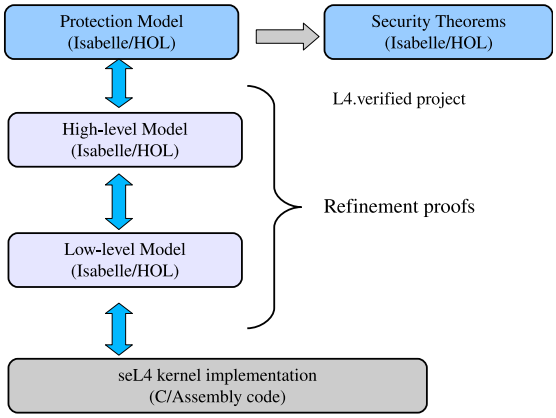


Figure 3.1: L4.verified approach to kernel verification.

Suppose the outcome of such a security analysis is favourable. The question then is "what does it mean for the deployed system?".

Formally speaking, the security analysis demonstrates that the protection model can enforce the given policy, but not necessarily the deployed kernel — the bottom-most layer of the verification hierarchy in Figure 3.1.

The answer to the above question comes from a *refinement* [dRE98] proof (see Figure 3.1). The aim of the refinement proof is to formally connect the protection model with the seL4 kernel implementation, such that the results of the security analysis is formally valid for the deployed kernel.

For the sake of completeness, it is worth mentioning that refinement is done in several, hierarchical stages (see Figure 3.1). As one moves down the hierarchy, models become less abstract with the final stage being the actual kernel. Each model in the hierarchy is formally connected to the model immediately above it via a refinement proof(s).

This formal refinement is beyond the scope of this thesis. The thesis does not claim any formal connection between the abstract model presented in Chapter 5 and the seL4 kernel. However, the desire to do so places additional requirements on the kernel's memory management model. I articulate these requirements by investigating, at a very high-level, the refinement work done in L4.verified project.

3.1.2 Proof Effort: Case Study

In this section, I quantify the effort exerted by the L4.verified project in proving a refinement relation between the kernel and its memory management model. I use the number of lines of (Isabelle) proof scripts as an indication for measuring the effort.

The refinement example used for this analysis is from the high-level model of the kernel to the low-level model. Pictorially, it corresponds to the middle double-sided arrow in Figure 3.1.

The L4.verified project uses the interactive theorem prover Isabelle/HOL [NPW02] for its refinement proofs. In this system, theorems are organised in a hierarchy of *theory* files (*.thy), with each file containing definitions, lemmas and theorems, generally about a particular part of the seL4 kernel. The number of lines in each file is counted using the *SLOCCount* [Whe01] tool.

File Name	Lines of script in the file	Lines of script related to kernel memory	Percentage of total (%)
KernelInit_R.thy	13,997	6203	5.9
CSpace_R.thy	13,025	2384	2.3
CNode_R.thy	12,122	961	0.9
Untyped_R.thy	5331	5331	5.1
Retype_R.thy	5251	5251	5.0
Ipc_R.thy	5142	0	0
Wellformed.thy	4714	0	0
Finalise_R.thy	4442	603	0.6
Detype_R.thy	1840	1840	1.8
Others	39,079	0	0
Total	104,943	22,573	21.5

Table 3.1: Break-down of lines of Isabelle proof scripts related to kernel memory management.

Table 3.1 shows a break-down of lines of Isabelle scripts in different theory files. In this table, the first column provides the name of the theory file. The second column of the table shows the number of lines of Isabelle script in each file and the third column is the number of lines of script that deals with the kernel memory management model. The final column shows the number of lines related to the kernel memory management model as a percentage of the total.

For clarity of presentation, I have grouped all the small theory files which do not deal with kernel memory into a single group called *others*.

There are three main theory files refining the kernel memory management model — *Untyped_R.thy*, *Retype_R.thy* and *Detype_R.thy*. The Untyped_R.thy file contains refinement theories on *Untyped* object. An Untyped object is a seL4 abstraction of a region of physical memory. I explain these abstraction later, in Chapter 4. The *Retype* and *Detype* operations allocate and de-allocate kernel objects respectively and the refinement theories on these operations are mostly in the two theory files Retype_R.thy and Detype_R.thy respectively.

In addition to the three main files, other files also contain theories about the kernel memory management model. Out of these, the most obvious is the *KernelInit_R.thy*, which contains refinement theories on kernel initialisation including theories on how the kernel memory pool is initialised and how memory is allocated the for initial system objects.

As I explain in Chapter 4, the seL4 kernel uses a *capability derivation tree* (CDT) to record how capabilities are derived from one another. The information stored in the CDT is essential for safe allocation of kernel objects (see Section 4.3.3). The CDT information is maintained by linking capabilities. These capabilities are stored in a capability address space or a *CSpace*, which is constructed by mapping *CNode* objects to one another. As such, the corresponding refinement theory files contain theories related to the memory management model.

In summary, the refinement proof consists of 105k lines of proof script, out of which 21.5% deals with the kernel memory management model. In terms of time, the entire proof took an estimated 8 person-years (py) to complete. Thus, we can *very roughly* estimate 1.7 py to complete the proofs related to kernel memory. However, as I explain below, the

1.7 py is an underestimation due to the cross-cutting nature of memory management.

Obviously, the proof effort depends on the complexity of the memory management model. The above estimation is for the seL4 memory management scheme described in Chapter 4. I envisage that the complexity of the seL4 memory management scheme is in par with those schemes used in high-end operating system kernels. Thus, the above estimation is a valid indicator for other high-end operating systems.

Modifications are Costly

For any kernel, irrespective of whether it is verified or not, modifying kernel code, in particular the low-level functionality of the kernel, is generally time consuming and hence costly. Such modifications usually involve low-level coding, testing, debugging and optimising.

The cost of modifying the kernel becomes even more prominent in the context of a formally verified kernel. Firstly, it breaks refinement theorems — refinement proofs demonstrate that code implementing the kernel is a refinement of a higher-level model. Any non-trivial modification (and most of the trivial ones) to the kernel means that proofs are no longer true. Hence the refinement will break. Depending on the nature of the modification, reestablishing refinement requires a significant effort to both reprove the invalidated invariants, and to prove newly required ones.

If the modification is a simple optimisation to the code — say, changing the code layout to improve the cache behaviour — then the effort required to fix the refinement proof would be low. Such modifications do not change the fundamental behaviour of the kernel and therefore, are not visible at the high-levels of abstraction (see Figure 3.1).

However, any large modification to the kernel which has cross-cutting concerns in other parts of the kernel requires significant effort. For example, the L4.verified team reports an effort of 1.5-2 py to re-verify the kernel after introducing just a new API call with cross-cutting features and the kernel data structures required for the realisation [KEH+09]. The bulk of this cost is for proving new invariants for the new code which have to be preserved over the whole kernel — not just in the new code.

Suppose we modify the kernel memory management model (because it does not suite the system requirements), say from a partitioning scheme to a best-effort allocator. In the *best case*, this modification will nullify 21% of the total refinement proofs which took approximately (based on the percentage) 1.7 py complete.

In practise, however, the 1.7 py is an underestimation. This is because of the cross-cutting nature of kernel memory management — almost all parts of the kernel depend on the memory manager for their functionality. For example, the thread subsystem relies on the kernel memory allocator for allocating thread control block, the virtual memory subsystem for allocating page tables and so on. Any modification to the kernel memory allocator, in particular a conceptual modification, is visible to all these kernel subsystems. These subsystems now have to be modified so that they co-operate correctly with the new allocator and the verification needs to prove that this co-operation is correct and sufficient for the expected functionality of whole kernel — triggering additional proof obligations in all parts of the kernel. Thus increasing the actual cost of re-verification.

Moreover, a modification to the kernel's externally visible behaviour require modifications to the higher-level formal models to establish refinement. Roughly speaking, refinement shows that kernel behaves according to its higher-level model. Thus, modifying the visible behaviours of the kernel model may require modifying its protection model so that one can prove refinement between the two. Modifying the protection model would break

the security theorems, which are based on the model (see Figure 3.1). Thus, one needs to reprove these theorems for the new model.

In summary, changing the low-level functionality of a formally verified kernel — such as how the in-kernel memory is managed — is highly expensive and therefore should ideally be avoided. Furthermore, a verified kernel should be capable of catering the needs of many diverse application domains so that the relatively-high cost of verification gets amortised.

Based on these, we can conclude that a verified kernel should ideally be capable of catering the needs of many different application domains without any modification to the kernel code base.

3.2 Summary of Existing Approaches

As discussed in Chapter 2, existing approaches to in-kernel memory management can be broadly categorised into three main classes: (a) those where memory management is governed by an in-kernel policy, (b) those where the kernel's memory is a cache of the system state and (c) those that seek to export the control of in-kernel memory management to user-level applications.

In the first class of systems, kernel memory is managed by a policy integrated into the kernel. While there is some flexibility to "fine-tune" the kernel policy, it can only be changed by modifying the kernel — a fundamental change to the kernel policy requires modification to the kernel's code. In the context of a verified kernel, such a modification means that the refinement proofs are no longer valid — nullifying the assurance provided by formal verification. Moreover, as discussed previously, re-verification can be costly, depending on the nature of the modification.

On the other hand, if we refrain from modifying the kernel integrated policy to preserve the assurance, then the verified kernel can only be deployed in a system where there is a natural synergy between the kernel's policy and the domain's requirements. For example, a kernel using a first-come-first-serve memory allocation policy cannot be deployed in a domain that warrants strict partitioning [BA03] and a kernel enforcing partitioning is inefficient for a best-effort system [Wal02]— efficiency can be significantly improved by reassigning memory to where it can be used. Ideally, a verified kernel should be capable of catering to the needs of a number of application domains, so that the high cost associated with verification gets amortised.

Kernels that treat their in-kernel memory as a cache of the system state can be viewed as a special case of kernel-integrated policy where the policy is caching. A single policy, as I discussed previously is not suited for all application domains. In this particular case, the kernel's caching scheme is not appropriate for a system with temporal requirements.

In the final class of systems, the in-kernel memory is managed by user-level *resource managers*. Through an API, the kernel exports its in-kernel memory management to user-level managers. In-kernel memory management policy is implemented outside of the kernel. Thus, as I demonstrate through this thesis, one can change the policy to suit the application domain by changing the user-level resource manager, rather than the verified kernel code base. The scheme supports diversity by means of different user-level resource managers— each resource manager implements its own policy over the resources it manages. Through the same mechanism we can even support co-existing, diverse resource management policies.

User-level management provides most of the features claimed by the kernel memory management model of a general purpose, verified kernel. The only feature missing is the ability to reason about the system behaviour. As an illustrative example, consider a user-level resource manager attempting to partition kernel memory. There are two questions the resource manager needs to answer; (a) can the kernel API be used for enforcing this policy, and (b) if so, how to configure the applications in such a manner to guarantee the enforcement. As this thesis demonstrates, by unifying the memory management model with the formal protection model (see Figure 3.1) one can facilitate this type of reasoning — it should be possible to answer the above questions, formally using the protection model.

3.2.1 Reasoning About Memory

A number of systems employ abstract models to facilitate formal reasoning of application behaviour. Such models have been examined in KeyKOS [Har85], EROS [SSF99, Sha03], Asbestos [VEK+07], and Hi-Star [ZBWKM06] including a formal proof of enforcing confinement in the case of EROS [SW00]. The sHype architecture [SJV+05] examined the application of mandatory access control to communication and shared resources to achieve isolation.

These models, however, focus only on the application level (at API level), with the tacit assumption that the kernel's internal behaviour, when providing services to these applications does not undermine the properties enforced at the API level. A clear relationship between the high-level model and the low-level management of physical memory in the kernel is either non-existent, insufficient or ad hoc. Ideally, there should be a clear, systematic (potentially formal) relationship between the kernel's physical memory management and the API model of the system, so that the same policy enforced at API level is adhered to within the kernel — isolation at API level for example, should imply isolation of physical memory inside the kernel.

3.3 seL4 Approach: Rationale

A kernel should provide an analysable relationship between a model of the system and the kernel's physical memory consumption. Thereby making it feasible to reason about and control the amount of resources used by the kernel in providing services to application software. Moreover, the kernel should minimise the allocation policy in the kernel, and maximise the control application software has over the management of in-kernel physical memory in an analysable manner.

The main issue in providing an analysable relationship between the model of the system and the kernel's physical memory management is *implicit* allocation. By implicit allocation I mean the memory allocated as a side-effect of providing a kernel service. There is loose relationship between these allocations and the authority to receive a kernel service. For example, the authority to install a frame in a virtual address space has only a lower bound (a frame) and has no connection to the actual amount of physical memory consumed by the kernel, which is determined by the layout, the number of page tables and the number of mappings in the virtual address space. These implicit allocations creates a gap between the model and kernel's internal behaviour.

Moreover, the need for an in-kernel memory management policy stems from these implicit allocations. The kernel needs a scheme to manage implicitly allocated metadata and hence an in-kernel memory management policy.

Motivated by these observations, we propose a memory allocation model that has two main properties. First, it eliminates all implicit memory allocations from the kernel, and makes all in-kernel memory allocations explicit via authorised requests from user-level resource managers. Second, all authority, including the authority to allocate kernel memory, is precisely captured and the effects of usage are modelled by a protection model.

The former property creates a direct relationship between authority and kernel memory usage. I show that this allows us to implement different kernel memory allocation policies by modifying the authority distribution and the way in which a user-level resource manager uses this authority.

The latter property facilitates reasoning — provided the protection model is amenable for doing so. We can enforce and reason about the enforcement, of a different policies as warranted by the application domain, based on the protection model.

The proposed scheme bears all the desired characteristics we discussed above. One can use the memory management policy best suited for a particular application domain without modifying the verified kernel code and hence without invalidating the refinement relation. The protection model facilitates reason about the policy enforcement and the refinement proof connects that reasoning with the deployed kernel.

However, the scheme presents several interesting challenges. First, there may be a performance penalty involved with exporting policy decisions on kernel memory allocation to user-level resource managers. Instead of the kernel deciding on when and where to allocate the memory it requires, it exports the decisions to user-level resource managers. This has performance implications that need to be addressed, quantified and reduced to a bare minimum. Second, we need a method to connect the protection model with the memory management model. As mentioned in Section 2.2, existing formal models do not capture the notion of a limited resource and hence cannot be used to model the semantics of kernel memory. However, there is the possibility of extending them to develop a new protection model that captures the behaviour of kernel memory. Finally, the protection model, while capturing the behaviour of kernel memory, should still be *decidable* with respect to useful properties (e.g. safety). Unfortunately, protection models are in general undecidable [HRU76] (with regards to safety analysis), hence do not facilitate reasoning. The model must be carefully crafted to avoid creating an undecidable model.

In the following chapters I discuss the above mentioned facets: I start by introducing the kernel model, in particular its memory management scheme in Chapter 4. Chapter 5 shows how to incorporate the authority to allocate kernel memory into a formal protection model. Then, Chapter 6 shows how this model can be used to formally reason about and enforce policies on memory usage of an application. Chapter 7 analyses the micro-level performance of a prototype kernel that implements the above model. The macro-level performance characteristics of the kernel model when used as a hypervisor is discussed in Chapter 8. Chapter 9 concludes.

Chapter 4

Conceptual Model

In this chapter, I present a high-level view of the proposed memory management model.

As discussed in Chapter 3, the main barrier to providing a principled scheme for managing the in-kernel memory is the *implicit* allocations that take place within the kernel. I define implicit allocations as the memory that is allocated by the kernel as a side-effect of obtaining a kernel service.

Almost all kernel provided services require metadata within the kernel for either directly supporting the service (e.g. page tables for implementing a virtual address space), or for storing additional bookkeeping (e.g. bookkeeping required for reclaiming resources on release). This metadata is managed implicitly by the kernel — when providing services, the kernel allocates metadata as and when required. For some kernel services, these implicit allocations of memory within the kernel make it hard, if not impossible, to reason about the memory consumption of a process given its authority to use kernel services. Due to the implicit nature of memory management within the kernel, one cannot make a direct connection between the authority of an application to obtain a service and the amount of physical memory consumed in providing the service.

The thesis propose a memory model that resolves the issue by promoting *all* dynamically allocated kernel memory (be it the memory required for directly supporting the service or the memory required for storing in-kernel bookkeeping) into first-class, explicitly allocated kernel objects. These kernel objects are allocated *only* upon receiving authorised requests from user-level resource managers, thus creating a direct relationship between authority possessed by an application and the amount of kernel memory it may consume. Consequently, one can reason about and enforce different policies on kernel memory management by reasoning about and controlling the flow of authority. I defer the discussion on controlling authority flow until Chapter 5.

Achieving a direct relationship between the authority possessed by an application and the amount of kernel memory it may consume is challenging for some kernel provided services. For example, there is little relation between the authority to use an address space and the amount of kernel memory consumed in providing it. Following sections, discuss these challenges, how they can be addressed, and the impact of doing so on the kernel provided abstractions.

The chapter is organised in the following manner: Section 4.1 provides the background required for the reminder of the chapter. Section 4.2 summarises the memory allocation model. Then the following three sections discuss the memory allocation model in depth — object allocation, recycling of objects and reuse of memory to implement a different object type is discussed in Section 4.3, Section 4.4 and Section 4.5 respectively. Then Section 4.6

discusses the impact of the allocation scheme on kernel abstractions. Section 4.7 reports practical issues in realising the model and finally Section 4.8 summarises.

4.1 Overview of seL4

4.1.1 Basic Kernel Model

The seL4 microkernel API is based on *kernel objects* and capabilities [DVH66] to control access to these kernel objects.

Each kernel object implements a particular abstraction and supports one or more methods related to the particular abstraction it provides. For example, a *Thread Control Block* (*TCB*) object implements a thread abstraction, and thread related services (thread related system calls) are the methods supported by this type of object.

The seL4 kernel uses partitioned (or segregated) capabilities [AW88] for access control. That is to say that capabilities themselves are stored within kernel-protected objects making them tamper-proof. All seL4 objects are named and accessed via capabilities. The possession of a capability with the required authority is sufficient to obtain a particular service from the object pointed to by the capability. Processes have no intrinsic authority beyond what they possess as capabilities.

System calls are implemented as object method invocations by invoking a capability that points to the kernel object with a method name. As such, all system calls, without exception, require at least one capability to authorise the operation.

An application can delegate all or part of its authority to another application, provided the it is authorised to do so. Moreover, the delegated authority can be removed by performing a *revoke* operation. The kernel maintains a *Capability Derivation Tree* (CDT) to store the bookkeeping required for implementing the revoke operation. When a revoke operation is performed on a capability (say X), the kernel locates all the CDT descendants of X and removes them from the system.

The kernel provides two operations for propagating a capability: the *mint* and *imitate* operations. A capability propagated via the mint operation has equal or lesser authority than the source and it is added to the CDT as a child of the source capability. On the other hand, a capability copy made via the imitate operation has the same authority as the source and it is inserted as a CDT sibling of the source.

4.1.2 System Structure

The overall structure of a system based on the seL4 microkernel is shown in Figure 4.1. The microkernel runs directly above the hardware with full hardware privileges and provides basic services to all other user-level components. Once the microkernel bootstraps itself, it creates the *initial resource manager* — a domain specific OS personality responsible for bootstrapping the remainder of the system.

The general structure of the system is a client-server configuration. Clients rely on *resource managers* (servers) for OS services such as virtual memory, threads, IO functionality etc. These resource managers implement services using a suitable management policy that they determine.

At system start-up, the kernel creates an initial resource manager and all the resources remaining for it to function. The two subsequent resource managers shown in Figure 4.1 (*RM1* and *RM2*) are created by the initial resource manager, and in doing so, the initial

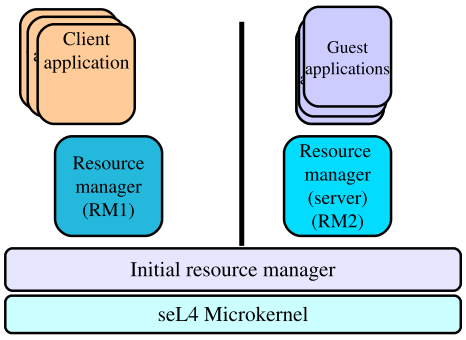


Figure 4.1: Example system configuration.

resource manager delegates part of its resources (by delegating capabilities) to the newly created managers. In this manner, the system supports hierarchical resource management with each resource manager in the hierarchy creating *sub-resource managers* by delegating a part of the resources it has. Moreover, the system supports coexisting resource managers implementing diverse policies. For example, *RM1* and *RM2* in Figure 4.1 can implement different policies over the resources they received from the initial resource manager.

4.1.3 seL4 Memory Management

The seL4 model eliminates all implicit allocations within the kernel by promoting *all* dynamically allocated kernel memory — be it the memory required for directly supporting the service or the memory required for storing in-kernel bookkeeping — into first-class, explicitly allocated kernel objects. These kernel objects are allocated *only* upon receiving authorised requests from user-level resource managers. Thus creating a direct relationship between authority possessed by an application and the amount of kernel memory it may consume. Consequently, one can reason about and enforce different policies on kernel memory management by reasoning about and controlling the flow of authority.

For regulating the in-kernel memory allocations, seL4 introduces a novel concept called *Untyped memory* objects or *UM* objects. An UM object represents a power-of-two sized, size-aligned region of physical memory. Possession of a capability that points to a UM object (an *UM capability*) is sufficient authority to allocate kernel objects within the corresponding region of memory. By calling the *retype* method implemented by UM objects user-level resource managers request the kernel to subdivide that region into kernel objects (including smaller or equal-sized UM objects). Details of this object allocation mechanism is discussed in Section 4.3.

With the exception of explicit allocations of objects via the retype method, the seL4 kernel *never* allocates metadata, thus, establishing a direct relationship between the authority of a process and the amount of kernel memory it may consume. As a result, controlling the dissemination of authority yields precise control of the amount of physical memory a process may consume.

As mentioned, seL4 promotes all dynamically allocated kernel metadata into first-class, explicitly allocated objects. Moreover, *all* the memory required for implementing the object and its functionality is pre-allocated, at the time of creation — once created, an object can serve its purpose without requiring any further memory resources. The obvious prereq-

uisite here is that the amount of memory required for an object's functionality is invariant. In the next section, I show that it is possible to define objects that adhere to this invariant and discuss the implications of doing so later in Section 4.6.

4.1.4 Kernel Objects

Objects implemented by the kernel can be divided into architecture independent and architecture dependent types. Architecture-dependent data structures such as page table nodes are encapsulated in objects visible to applications via the API when the application possesses the appropriate capability.

Architecture Independent Objects

The following object types are independent of the underlying hardware architecture.

Untyped encapsulates a region of power-of-2, fixed-sized, size-aligned, continuous region of physical memory that can be retyped to allocate new kernel objects.

TCB contains the state of a thread, including four capabilities to associate the thread with its (a) capability address space (*CSpace*), (b) virtual-memory address space (*VSpace*), (c) exception handler *endpoint*, and (d) the *reply* endpoint. The two endpoint capabilities in the TCB are used for handling exceptions — the former for notifying a handler and the latter to notify the client, once the exception is resolved. The size of a TCB depends on the architecture (because of the thread state), but for a given architecture the size is fixed.

Endpoint is a synchronous interprocess communication rendezvous point. It contains storage for endpoint state and a queue head for threads blocked on the endpoint. The thread queue itself is implemented by linking the associated TCBs in a doubly-linked list. Thus its size is fixed.

AsyncEndpoint is an object supporting an asynchronous interprocess notification mechanism. It contains the storage required for storing *a message* and a queue head for threads blocked waiting for a notification. Similar to the previous case, blocked thread queue is implemented via linking associated TCBs. If more than one message is written to the object, the kernel combines the messages — for example, by XORing — to produce a single message. Hence, the object consumes only a fixed amount of memory.

CNode is a power-of-2, fixed-sized storage container for capabilities, which can be combined (by mapping one CNode to another) to form a graph-like data structure representing a thread's CSpace. A CNodes contains a power-of-2, fixed number of slots (specified at the time of creation) for storing capabilities and associated *CDT* information. I introduce the CDT later in Section 4.3.3. For now it is sufficient to note that the space required for a CDT entry is fixed (2 machine words). Moreover, a capability requires another 2 machine words; making a total fixed per-slot storage of 4 machine words. As such, the kernel can compute the total storage required for a CNode at the time of creation.

Architecture-Dependent Objects

Architecture-dependent objects are used to promote hardware-defined data structures into first-class objects in the kernel's API. For the purposes of this thesis, the underlying archi-

Operation	Capability invoked	Restrictions
retype	parent UM capability	leaf in the CDT
recycle object	master capability	no imitated copies
reuse memory	parent UM capability	-

Table 4.1: Summary of the kernel memory management API.

tecture is that of the ARM11 [ARM05].

PageDirectory is an object representing a specific virtual address space and the right to manipulate it. It contains the root page table (page directory) of the two-level page-table structure of ARM. Note that the size of a PageDirectory is fixed.

Frame is a frame of physical memory that is installable in a virtual address space for providing backing storage for virtual memory. The size of a Frame is specified at the time of creation and is invariant.

PageTable is the leaf node of the two-level page table structure of ARM architecture. On ARM, the size of a leaf node is fixed at 4KB.

InterruptController is an object that stores the current interrupt associations of the kernel. seL4 features user-level drivers, and delivers interrupts to user level via the AsyncEndpoint notification mechanism. The size of this object is fixed and determined by the number of hardware interrupts.

InterruptHandler is used by user-level interrupt handlers to acknowledge the interrupt. This is a *dummy* object (which consumes no memory) for facilitating interrupt acknowledgement.

seL4 ASID Table object stores the association of a VSpace with a hardware address space identifier and PageDirectory object implementing the VSpace (see Section 4.6.3 for more details). The size of the seL4 ASID table determines the number of possible concurrent address spaces and it is fixed at kernel compile time.

4.2 Overview of Memory Management API

Table 4.1 summarises the kernel memory management API. For each operation, the table shows the capability that needs to be invoked to perform the operation and the restrictions on the invoked capability.

In the following sections, I discuss the memory management API in depth. The retype operation — the method used for object allocation — is analysed in Section 4.3. The mechanism for recycling an existing object is discussed in Section 4.4 and how memory is reused to implement a different object type is discussed in Section 4.5.

4.3 Kernel Object Allocation

Having introduced a rough sketch of the model, now I describe the object allocation scheme in depth.

Once the kernel has bootstrapped itself, the authority over the remaining memory (not used by the kernel) is conferred to the initial resource manager in the form of UM capabilities to distinct, non-overlapping UM objects.

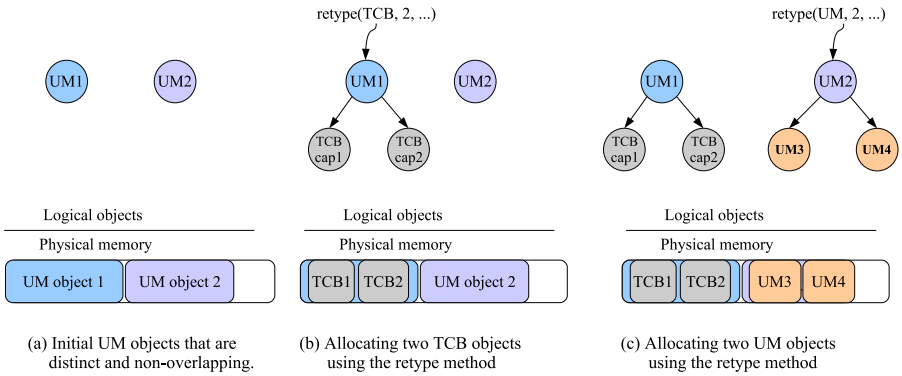


Figure 4.2: Allocation of different kernel object types using the retype method. Circles represent capabilities and rounded boxes correspond to kernel objects.

Shown in part (a) of Figure 4.2 are two such UM capabilities and the regions of physical memory they represent. In this diagram, circles and rounded boxes represent capabilities and kernel objects, respectively.

The initial resource manager can use the UM capabilities in its possession to allocate other kernel objects. This is achieved by calling the retype method on an UM capability, specifying the type and the number of kernel objects that need to be created (see part (b) of Figure 4.2).

Note that the retype method can allocate more than one kernel object at a time, but all allocated objects are of the same type. In theory, a single retype operation can allocate objects of different types. However, we did not find compelling, practical examples where such an operation is useful.

When the retype method creates new kernel objects, it returns capabilities with full authority to the newly-created objects — one capability for each new object. For example, as shown in part (b) of Figure 4.2, upon receiving a request to retype *UM1* into two TCBs, the kernel creates two new TCBs in the corresponding memory region, and returns two capabilities, with the full authority to each TCB object. These new objects are called *children*, and the UM object used for the retype operation the *parent*. When creating new objects, the retype operation enforces the following invariants:

1. The child object(s) must be wholly contained within the original parent UM object,
2. the child objects must be non-overlapping (not even partially), and
3. the child objects must be distinct.

The retype operation consumes kernel memory in two ways. First, the newly created objects require backing storage. This storage is provided by the parent UM object and the first invariant above guarantees that we do not exceed the provision provided by the parent. Second, placing the newly created capabilities consumes memory, in terms of *capability slots*—storage containers within the kernel for capabilities. The application performing the retype operation provides the storage space required for placing the new capabilities by supplying (as an argument to retype) a capability to a *CNode*—an array of capability slots—and an offset into this array. The kernel places the newly-created capabilities in these slots, given that the capability slots are initially empty. Thus, creating the child capabilities does not consume any kernel metadata, beyond what is already supplied.

I discuss the need for the second and the third invariants later in Section 4.3.1, and Section 4.3.3, respectively.

The initial resource manager may delegate all or part of the authority it received from the retype operation to one or more of its clients, using either imitate or mint operations, respectively. Thereby, the initial resource manager allows a client to obtain kernel services by invoking the object.

The retype operation is capable of creating several types of objects, including smaller UM objects. As such, the available storage can be divided into smaller regions, each of which may be retyped individually or delegated to another user-level resource manager.

As mentioned, all the physical memory required to implement a kernel object is preallocated to the object at the time of its creation. Moreover, this amount does not exceed the memory resources of the parent UM it was derived from. This eliminates the need for the kernel to dynamically allocate memory to satisfy requests from user-level tasks.

4.3.1 Type and Memory Safety of Kernel Objects

In order to ensure the integrity of kernel objects, the kernel needs to ensure that a region of memory implements a single object at any given time.

To make this guarantee, the retype operation must guarantee that no existing object is using any of the memory being consumed by retype. Thus, before starting the retype operation, the kernel checks whether there are any existing objects within the region of memory represented by the UM object.

Returning to the above invariants on retype (see Section 4.3), the first two restrictions, together with the fact that initial UM objects are distinct and non-overlapping, guarantees that two UM objects can only overlap if they have a parent-child relationship. For example, the newly-created UM objects, *UM3* in part (c) of Figure 4.2, overlaps with and is a child of *UM2*. Similarly, any *typed object* — any object type other than UM, that resides in the memory region encapsulated within an UM is a child of the parent UM — the two TCB objects that consume memory from *UM1* are children of *UM1*. So, the kernel can make the above guarantee by ensuring that the parent capability used for the retype operation has no children (typed or untyped children). I defer discussing the mechanism used to track parent-child relationship and the need for the third invariant until Section 4.3.3.

4.3.2 Preventing User-Level Access to Kernel Data

For security reasons, kernel data structures must be protected from user-level access. I now explain how this is achieved in the seL4 model.

The kernel prevents user-level access by using the following mechanisms. First, the allocation policy guarantees that two typed objects never overlap. Two objects may overlap in physical memory *only* if at least one object is an UM object. Moreover, only typed objects contain any data.

Finally, the kernel ensures that each physical frame mapped by the MMU at a user-accessible address is of the *Frame* object type. These Frame objects contain no kernel data and since they cannot overlap with other typed objects, direct user access to kernel data is not possible.

4.3.3 The Capability Derivation Tree

For type and memory safety of kernel objects, the kernel needs to guarantee that the memory being used by the retype operation is not currently used by any existing kernel object.

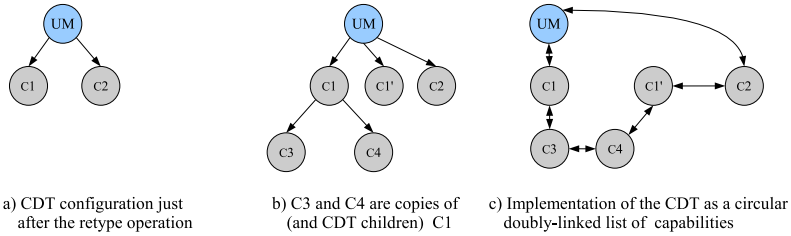


Figure 4.3: The capability derivation tree (CDT). Part (a) shows the CDT configuration soon after the retype operation and Part (b) demonstrates the configuration of the CDT after granting copies of C1 to other clients and given in Part (c) is the internal representation of the corresponding logical tree.

Since, the retype operation only consumes memory from a single UM object the kernel only needs to guarantee that the parent UM object used for retype has no existing children (including smaller UM objects). The *capability derivation tree* (*CDT*), is the kernel's internal bookkeeping mechanism to track the parent-child relationship between kernel objects.

An obvious design questions here is how to manage the storage required for the CDT?. In seL4, all existing objects have *at least* one capability pointing to them. This property provides an obvious location for including the CDT information — capabilities.

Whenever a new capability is created, that is, when a UM object is retyped, or when rights to a typed object are delegated by copying its capability, the kernel records the relationship between the new capability and its parent in the CDT. For instance, the two TCB capabilities generated via the example retype operation in Figure 4.2, are inserted into the CDT as children of the parent UM capability. The configuration of the CDT soon after the retype operation is shown in part (a) of Figure 4.3. In this figure, *C1* and *C2* denote the two capabilities generated by the example retype operation.

Once an object is created, the resource manager may delegate its authority over the new typed object to one or more of its clients by granting each client a copy of the capability. seL4 provides two mechanisms for propagating capabilities for typed objects (or typed capabilities)—the *mint* operation and the *imitate* operation. A capability propagated via the mint operation has equal or lesser authority than the source and it is added as a CDT child of the source. In Figure 4.3, for example, *C3* and *C4* are copies of *C1* made via the mint operation. On the other hand, a copy made via imitate has the same authority as the source capability and it is inserted as a CDT sibling of the source. The capability *C1'* in Figure 4.3, for example, is a copy of *C1* created via imitate. Note that these capability copies as well as the capabilities generated via the retype method are descendants of the parent UM capability — all capabilities that point to objects within the region covered by the UM object are CDT descendants of the parent UM capability.

The kernel avoids dynamic allocation of storage for CDT nodes by implementing it as a circular, doubly-linked list stored within the capabilities themselves. Shown in part (c) of Figure 4.3 is the linked list representation of the logical tree from part (b) of the same figure. This list is equivalent to the pre-order traversal of the logical tree. To reconstruct the tree from this list, each node in the list is tagged with a depth field — the depth of the node in the logical tree.

The number of bits used to encode the depth imposes a limit on the CDT depth and hence a limit on the number of mint operations. For example, the prototype kernel presented in this thesis uses 7bits to encode the CDT depth and hence the tree is limited to 128 levels. It is possible to remove this limit by implementing the CDT as a tree, rather

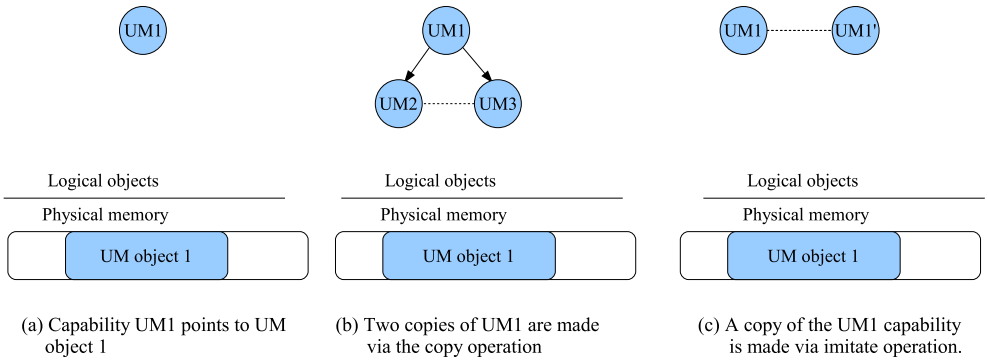


Figure 4.4: Propagating untyped capabilities using mint and imitate operations. Part (a) shows the initial UM capability. The CDT after making two copies of *UM1* using mint and a single copy using imitate operation is shown in part (b) and (c) respectively. Arrows are used to point to children and dotted lines connect siblings.

than the pre-order traversal of it. However, this option was not considered as it requires additional memory provision within capabilities. Moreover, experience with seL4 based systems suggests that limited depth is not an issue in practise.

The CDT provides a simple mechanism to test whether it is safe to allocate objects using a parent UM — for safe allocation the parent UM object must be a leaf in the CDT. Moreover, given that the pre-order traversal of the logical tree is readily available, this means that the capability to the right in the list of the parent UM capability must be either a sibling (equal depth) or an ancestor (lesser depth). However, this simple test is sufficient only in conjunction with another restriction on untyped capabilities, which I examine below.

Untyped Capabilities

Before focusing on the restriction, I introduce the problematic scenario. Consider the untyped capability *UM1*, in Figure 4.4. Suppose we allow capability copy operations — mint and imitate — on this UM capability. The CDT configuration after creating two copies; *UM2* and *UM3*, of capability *UM1* via the mint operation is shown in part (b) of the figure. Since these copies are made via mint, they will be installed into the CDT as children of the capability *UM1*. In the figure, out-bound arrows point to child capabilities and dotted lines connect siblings. In the CDT configuration in part (b), even though *UM2* has no children, the kernel cannot safely allocate new objects in the memory region it points to, without knowing the status of *UM3*. Establishing the status of *UM3* would require a traversal up the CDT tree, until a common parent is found, and back down on all possible branches — which can potentially be a long running operation. Part (c) of the figure shows the CDT configuration after making a copy of *UM1* through the imitate operation. In this case, as above, the kernel cannot allocate objects safely using *UM1* without knowing the status of *UM1'*.

The issue arises from the fact that we now have two capabilities that are siblings in the CDT, but are pointing to the same UM object. The kernel prevents this problematic situation from occurring by using two techniques. First, the third restriction on retype (see Section 4.3) — newly-created child objects must be distinct — ensures that new UM capabilities created via retype point to distinct (different) UM objects. Recall the semantics

of retype: after allocating objects it returns a set of capabilities — one per newly-created object. The capabilities in this set are siblings in the CDT — all of these capabilities are inserted into the CDT as children of the parent UM capability. Since the newly-created objects are distinct, none of the siblings in this set would point to the same UM object.

Second, the kernel disallows copy operations (both mint and imitate) on UM capabilities. The *only* way to propagate a UM capability is by retyping — retype the parent UM into another UM object of the same size. However, note that, as an optimisation, we can allow mint operations on UM capabilities — conceptually, mint has the same effect as retyping into an object of the same size.

These two invariants, together with the semantics of retype ensures that if a capability points to an object; be it a typed object or an untyped object, that consumes memory from a region covered by an UM object, then the capability *must be* a CDT descendent of the UM capability. As such, if a parent UM capability does not have any CDT children, then there are no (currently existing) objects within the region covered by the UM object. So, performing the above mentioned simple test is sufficient to guarantee that it is safe to perform the retype operation.

4.3.4 Summary of the Retype Operation

Before proceeding, I summarise the operational semantics of retype:

Operation: Creates a set of kernel objects and returns a set of capabilities to these newly-created objects with the full authority to the newly-created object.

Arguments: User-level resource managers provide the following arguments to the retype operation.

- The parent UM capability — the retype method is called by invoking the parent UM capability.
- A set of empty capability slots for placing the result capabilities.

Restrictions: The following preconditions must be satisfied by the parent capability:

- Must be a UM capability, and
- must be a leaf in CDT.

Invariants: The retype operation enforces following invariants:

- Child objects must be wholly contained within the parent,
- child objects must be non-overlapping, and
- child objects must be distinct.

CDT relationship: The set of output capabilities to newly-created objects are:

- CDT children of the parent, and
- any two output capabilities are CDT siblings.

4.4 Recycling Memory

Once the retype operation creates kernel objects, the resource manager can use these objects to provide services to its clients. For example, the resource manager may allocate a

TCB object and use it to provide a thread abstraction for client *A*. At some point in time (when client *A* no longer needs the thread), the resource manager may decide to reuse the same TCB object to implement a thread for a different client *B*.

The seL4 memory management model facilitates the reuse of existing objects in a different context through the *recycle* operation. The recycle operation, revokes all the outstanding capabilities and reconfigures the object to its initial state such that it can be reused in another context.

When the retype operation creates new kernel objects, it returns capabilities — one capability per newly created object. These capabilities, generated by the retype operation are called *master* capabilities.

Possession of the master capability to an object is sufficient authority to recycle the object. By calling the recycle method on the master capability, a resource manager can instruct the kernel to revoke all the outstanding capabilities and reconfigure the object to its initial state such that it can be used in a different context without any residual state from the previous.

The kernel disallows recycling an object if there exists an imitated copy of the master capability. Recall that imitate creates a capability with equal authority to the source and the copy is inserted as a CDT sibling of the source. In that sense, an imitated master capability means two master capabilities to the same object. The processes holding these two master capabilities need to coordinate to recycle the object — one process needs to give up its authority for the other to recycle the object. However, a resource manager can avoid the need for such coordination by refraining from imitating the master capability.

4.5 Reusing Memory

The model described thus far is sufficient for safe allocation of kernel objects and reusing the allocated objects from one context to another.

This model alone is sufficient for a simple static system configuration and even a dynamic system where it is not necessary to change the balance between the object types. However, for a highly dynamic system, the kernel needs to support safe reuse of memory. Reuse here refers to using a region of physical memory to implement a different kernel object type, rather than reusing an already allocated kernel object in a different context.

Before reusing a memory region, the kernel needs to guarantee that there are no outstanding references to the objects that previously occupied that memory. These references include capabilities and direct pointers kept within other kernel data structures.

Possession of the parent UM capability that was used to allocate kernel objects is sufficient authority to delete those objects. By calling a *revoke* operation on the parent capability, users can remove all of the CDT descendants, including capabilities that are not directly accessible to the resource manager that possesses the parent capability.

Recall the structure of the CDT — any capability that has a reference to an object within the region of memory covered by the UM object is a CDT descendent of the parent UM object capability. Thus, when performed on the parent UM capability, the revoke operation removes all capabilities that contain references to objects within the memory region covered by the parent.

Removing the last capability to a kernel object destroys the only explicit record of the object's type and location information. Before this happens, the kernel deactivates the object and removes any internal references to it from other kernel data structures. For example, removing the last capability to a TCB halts the thread and removes it from the

scheduling queues and any such structure that may contain a reference to it. This cleanup work is done by the internal *destroy* operation, which is triggered just before the last capability to an object is deleted.

Detecting the last capability reference to an object is done via the CDT. A capability is the last reference to an object when:

- The capability has no CDT children or *identical* siblings (siblings that point to the same object irrespective of access rights), and
- the CDT parent of the capability is untyped.

In other words, a capability is the last reference to an object when no other capability is pointing to the same object and the parent is untyped.

By enforcing an additional invariant on the CDT structure, the above test can be performed efficiently using the pre-order traversal of the CDT tree. The invariant here is that *identical* siblings in the CDT — siblings pointing to the same kernel object, irrespective of the access rights – must always be adjacent. This is initially a consequence of the fact that retype returns capabilities to distinct objects and is maintained by the CDT insert operation.

With this invariant in place, the kernel can detect the last capability to an object by considering just the two adjacent entries in the pre-order list of the CDT: A capability is the last reference to an object when both adjacent entries refer to other kernel objects. The ease of detecting the last reference via the CDT avoids reference counting, which is an issue for objects without space to store a reference count (e.g. page tables), in particular in a system without any dynamically allocated metadata.

In summary, when performed on a parent UM capability, the revoke operation identifies *all* the capabilities that refer to objects within the memory region of the parent UM and removes them from the system. Before removing the last capability to an object the kernel breaks all the internal dependencies between that object and other kernel objects. Therefore, upon the completion of revoke, there will be no outstanding references to the objects that previously occupied that memory.

Once the revoke operation on the UM capability is complete, the memory region can be reused to allocate other kernel objects, by calling the retype method on the parent UM capability.

4.6 Managing Address Spaces

Having described the memory allocation model of seL4, I now discuss the implications of the allocation model on seL4 abstractions. In particular, as a result of the allocation model, seL4 address spaces are slightly different to what is provided by a more traditional microkernel.

A seL4 thread is associated with two address spaces — a capability address space (or CSpace) and a virtual memory address space (or VSpace). The need for two types of address spaces, mainly stems from hardware defined data structure formats used for implementing the VSpace.

Address spaces provided by seL4 (VSpace and CSpace) have slightly different semantics to what is provided by most small kernels. To facilitate the discussion, I define two terms. I use the term *visible objects* to refer to those objects that are mapped into an user-level address space. For example, memory pages mapped into a virtual address space are visible to the application and hence are visible objects. In addition to visible objects, an

address space requires *meta objects*, either to directly support the abstraction (e.g. page tables) or to provide the bookkeeping for supporting the operational semantics of the abstraction (e.g. bookkeeping for revoking a page mapping).

In many small-kernel-based systems, including Mach [YTR+87], Chorus [ARG89], Grasshopper [RDH+96] and Sawmill [APJ+01], the semantics of visible objects (memory objects) is implemented outside the kernel by user-level pagers. However, the management of meta objects required for implementing the address space is, traditionally, done within the kernel.

The seL4 model exports the control of both visible objects and meta objects to user-level resource managers through the capability interface. Meta objects required for constructing an address space are encapsulated as kernel objects, protected via capabilities and allocated and managed using the techniques discussed in Section 4.3.

Constructing a seL4 address space requires allocating the required meta objects, installing mappings into these meta objects to construct the required (meta-object) layout of the address space and finally, installing mappings for visible objects.

Exporting the management of meta objects to user-level resource managers has two main implications. First, there is a performance penalty involved in exporting the management, traditionally, done within the kernel, to a user-level manager. Second, the management interface needs careful designing to avoid a malicious resource manager leaving the meta objects (used for an address space) in an inconsistent state. In this section, I discuss the design and how the kernel is guarded against misbehaving resource managers. I defer the performance analysis until Chapter 7.

In most cases, the management of a VSpace is architecture-dependent, because the meta objects required for implementing a VSpace are defined by the architecture. For this discussion, the underlying architecture is that of the ARM11 [ARM05]. The design, however, can be generalised for any architecture with a multilevel page-table format.

In the remainder of this section, I discuss the address space exception handling mechanism (see Section 4.6.1) and the semantics of CSpace (see Section 4.6.2) and VSpace (see Section 4.6.3).

4.6.1 Exception Model

Exceptions are handled via the kernel generating and delivering an *exception IPC* to an exception-handler Endpoint on behalf of the thread generating the exception [YTR+87]. A capability in the TCB of a thread associates the thread with an exception handler (see the contents of a TCB given in Section 4.1.4).

The payload of the exception IPC contains sufficient information about the meta objects required for resolving the exception. For example, an exception IPC notifying the handler of a page fault indicates whether or not a second-level page-table object is required for resolving the exception.

In a traditional setup, the kernel notifies user-level managers only about exceptions on visible objects. In contrast, seL4 generates exceptions for both visible objects and meta objects and then delivers to user-level managers. SeL4 user-level managers are responsible for handling exceptions on visible objects as well as meta objects.

4.6.2 Capability Address Space

The capability address space (*CSpace*) is a logical address space of capabilities. It forms a name space for capabilities — similar to a virtual address space translating virtual address into physical, a CSpace translates *capability indexes* into capabilities. A capability index is a 32bit (or 64bit in a 64bit machine) quantity used to name a capability within a CSpace.

The structure of the CSpace and the operations provided by the kernel to manipulate it are important for security, maintainability and the overall system efficiency (both temporal and spacial efficiency). Since the majority of kernel's metadata is stored within CSpaces — as capabilities and their CDT relationships to one another — the integrity of CSpaces is crucial for system security. Furthermore, since the CSpace itself is maintained by potentially malicious, user-level resource managers, the kernel operations to manipulate the CSpace must be carefully crafted to avoid security breaches, while providing the required functionality to manage the space efficiently. The layout of the CSpace impacts the efficiency because every seL4 system call requires at least one capability address translation to locate the capability that authorises the operation. In the following sections I introduce a CSpace design and discuss how the design addresses security and efficiency requirements.

Structure of CSpace

CSpaces are formed by connecting CNodes to form a directed graph by placing the capability to one CNode within a capability slot of another.

Each CNode contains a number of capability slots. The number of slots contained within a CNode may vary, but is always a power of two specified by the user-level manager at the time of creating the CNode object and not changing afterwards.

Address Translation

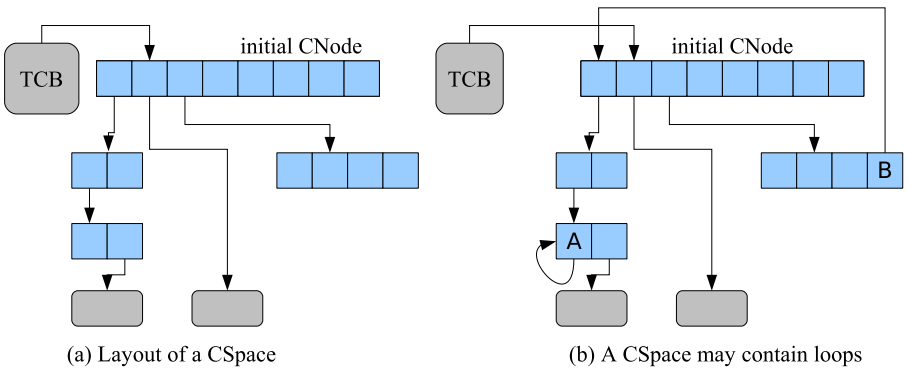
The decoding of a capability index is done by using the most significant bits to index an initial CNode (registered with the TCB of the thread) identifying a capability slot. If the slot contains a capability to another CNode, the remaining bits are used to repeat the indexing process in the next CNode. The process repeats until there are no remaining address bits (32bits or 64bits, depending on the architecture) to decode. The process can also terminate early successfully if a slot contains a non-CNode capability, or raise an exception in the case of an empty slot.

Additionally, the decoding method uses a guarded-page-table (*GPT*) [LE96] like shortcut mechanism to avoid decoding all bits via many levels of CNodes. Similar to a GPT, one can skip intermediate levels required to complete the decoding by comparing the most significant bits of the capability index with a *guard* value specified in the CNode capability. If the guard is a prefix of the capability index, then a number of bits equal to the size of the guard are deemed decoded and the remaining bits are decoded using the same process.

Naming Objects in a CSpace

In addition to managing the visible objects mapped into an address space, a seL4 resource manager needs to manage meta objects implementing the address space.

As an illustrative example, consider the CSpace layout shown in part (a) of Figure 4.5. In this figure, boxes represent CNodes and smaller boxes within them show the capability slots. Rounded boxes denote other (than CNode) kernel objects, and arrows represent



(a) Layout of a CSpace (b) A CSpace may contain loops

Figure 4.5: An example CSpace layout. Rectangular boxes represent CNode objects that constitute the address space and rounded boxes represent other kernel objects. Part (a) represents a typical CSpace layout of a thread and part (b) shows a CSpace which contains loops.

capabilities. The TCB objects shown in the figure denote the client thread(s) associated with this CSpace. Note that a capability slot also contains CDT information which I have omitted from the figure for clarity. A resource manager managing this CSpace needs to access individual CNodes that constitute the space in addition to the visible objects that are mapped into the space. This consumes a significant portion of the resource managers capability address space, given that a single resource manager may manage the CSpace of may client processes.

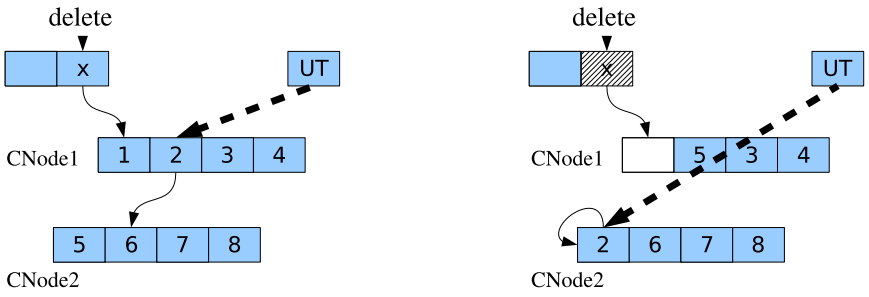
In seL4, the possession of a capability to a CNode with suitable authority is sufficient to manage the entire CSpace rooted at that CNode. For example, the capability in slot *B* (see in part (b) of Figure 4.5), authorises the management of the entire CSpace, provided the capability contains sufficient authority. Individual CNodes constituting the space are named by providing a capability index and a *depth* — the number of bits that must be translated to reach the node — relative to that CSpace. Thereby, seL4 eliminates the need for having a number of capabilities, within the CSpace of a resource manager to address individual CNodes of a client CSpace.

Malformed CSpaces

Because the layout of a CSpace is managed outside the kernel by potentially, malicious resource managers, the kernel can no longer rely on its structure when using it for decoding addresses.

A malformed CSpace, may contain *loops* — for example, capability slot *A* in part (b) of Figure 4.5, contains a capability to the object that contains capability slot *A*. In the presence of loops, the translation may never terminate.

The seL4 kernel does not attempt to prevent or even detect loops. Instead, seL4 uses the following two techniques to guarantee the termination of address lookup, even in the presence of loops. First, the translation process terminates once a fixed number of bits is translated. Second, at the time of retype, the kernel disallows the creation of CNode objects with a single capability slot. This guarantees that every CNode translates *at least* one bit from the address by means of indexing (regardless of the guard). Thus, every address lookup is guaranteed to terminate.



a) Request to delete the capability X

b) Rearrange the contents of the two CNodes

Figure 4.6: Kernel rearranges the contents of the deleted CNode. In the figure, small boxes represent CNode slots and numbers refer to capabilities they contain and hashed boxes denote that the capability is a zombie. Continuous lines denote the last capability to the CNode pointed to by the arrow and outbound dotted lines point to the CDT child of a capability.

Deleting a CNode

Recall from Section 4.5, just before the last capability to a kernel object is destroyed, the kernel destroys the object pointed to by that capability. Consequently, the destruction of a CNode is potentially recursive — the destroyed CNode might contain the last capability to another CNode, which in turn contains the last capability to another, and so on.

To avoid potentially unbounded recursion (and hence unbounded kernel stack), seL4 uses the algorithm described below to delete CNodes. This algorithm is motivated from the following observations:

- When a CNode is being destroyed, its last capability is marked as a *zombie*, preventing any access to slots within the CNode. This means that the actual content within the deleted CNode is opaque outside the kernel.
- If a deleted CNode contains the last capability to another CNode, then the second CNode can only be accessed via the CDT when performing a revocation.

Based on these observations, note that the kernel can rearrange the contents of a CNodes undergoing deletion arbitrarily as ‘long as the CDT links are preserved.

Now I explain the algorithm using the diagram shown in Figure 4.6. In this figure, boxes represent the slots in a CNode, numbers inside the boxes refer to individual capabilities, the outbound arrows with continuous lines indicate that it is the last capability to the CNode pointed to by the arrow and outbound arrows with dotted lines denote that the capability pointed to by the arrow is a CDT child. If a box is hatched, it indicates that the capability is a zombie. Note that, all capabilities in the figure have a CDT relationship to another, but I have omitted them for clarity.

Part (a) of the figure shows the initial configuration. Now, suppose a request comes in to delete the last capability (*X*) to *CNode1*. While deleting the individual slots of *CNode1*, the kernel detects that capability *2* is the last capability to another CNode (*CNode2*).

The seL4 kernel avoids recursive calls by deferring the deletion of *CNode2* — instead of deleting *CNode2* the kernel swaps capability *2* with the capability in the first slot of *CNode2* (capability *5*). The configuration after rearranging the contents is shown in part

(b) of Figure 4.6. Now the kernel can proceed deleting *CNode1*, given capability *5* is not the last capability to another CNode. If capability *5* is the last capability to another *CNode*, then the kernel repeats the swap operation. This swapping is done until the kernel finds a capability that is not the last capability pointing to a *CNode* and then proceeds with the deletion of *CNode1*.

As for *CNode2* (in Part (b) of Figure 4.6) the only possibility of reaching it, is though a revoke operation performed on the *UT* capability — the Untyped capability used to allocate the *CNode2* — at that point the kernel deletes *CNode2*.

4.6.3 Virtual Memory Address Space

One of the challenging problems encountered during the development of seL4 was the design of a safe and efficient mechanism for manging the virtual memory address space (*VSpace*) — in particular, on architectures with hardware-defined page table structures. Since all dynamically allocated kernel memory is managed outside the kernel, the kernel needs to export the management of hardware-defined page tables to user-level in a secure and efficient manner.

The main challenge in the design is that the VSpace and its associated hardware units, such as the *Translation Lookaside Buffer* (TLB), contains entries that references kernel objects (e.g. Frame objects). These references are essentially capabilities stored in a hardware-defined format. The kernel must be able to reach these capabilities by walking the CDT, thereby allowing them to be kept consistent with the CSpace capabilities they were derived from. Recall from our previous discussions that all metadata required for safe management of kernel memory is kept within the CSpace and the CDT. Thus, the consistency between these references and the CSpace is crucial for safe memory management.

For safe management of memory, the kernel enforces the following invariant: a reference in a hardware defined structure is *usable* only if the capability from which it was derived exists. When the capability is revoked — say, to reuse the memory to implement a different object type — the kernel guarantees that references to that particular object kept in hardware defined structures are no longer usable.

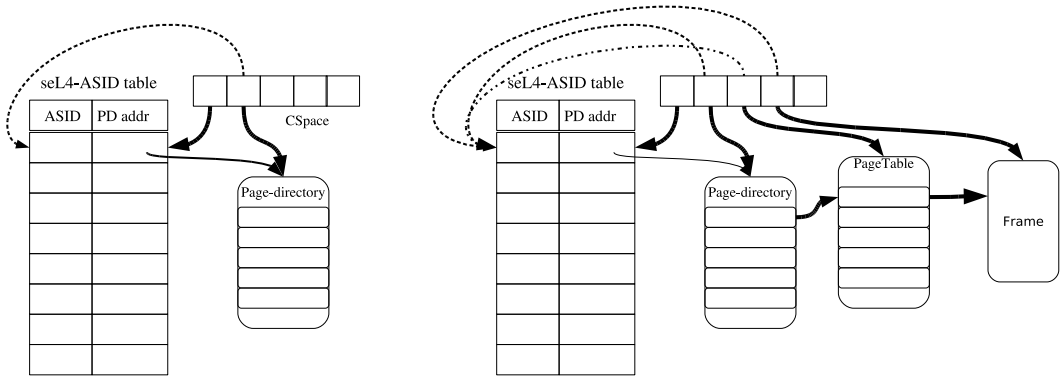
To enforce the above invariant, the kernel enforces some additional restrictions on object capabilities that constitute the VSpace. This section discusses the VSpace management API and the restrictions required to guarantee the invariant. The discussion is based on the VSpace implementation for the ARM11 [ARM05] (ARMv6 ISA) architecture. The concepts, however, can be generalised to any architecture with a multilevel-page-table structure.

Virtual Memory on ARM11

The *Memory Management Unit* (*MMU*) architecture of ARM11 consists of a two-level, hardware-walked page-table structure and a tagged TLB — tagged with a 8bit *Address Space Identifier (ASID)* — giving 256 hardware address spaces.

I first introduce the objects provided by seL4 for managing a VSpace from user-level and then discuss their usage through an example.
The seL4 API defines the following objects for VSpace management:

seL4 ASID Table: The seL4 ASID table is a global, fixed-sized table created at the time of bootstrapping the kernel. It is global in the sense that there is only one table in the system — the one created at boot time. The table is indexed by a *seL4 ASID*— thus,



a) PageDirectory associated with a seL4 ASID

b) Page mapping in a seL4 VSpace

Figure 4.7: Creating a virtual address space. The square box represents the seL4 ASID table and the rounded boxes represent kernel objects required for implementing a VSpace. The dark arrows represent capabilities and the dotted arrows show the weak-links.

its size determines the maximum number of concurrent virtual address spaces in the system. Each seL4 ASID is associated with a hardware ASID.

To prevent denial-of-service attacks based on exhausting seL4 ASIDs, the seL4 ASID table slots are exported to user-level as capability protected objects.

PageDirectory: A PageDirectory object defines the root page table of the two-level, hardware defined page table structure. A suitably authorised capability that points to a PageDirectory gives the holder the right to manipulate the particular VSpace.

PageTable: The PageTable object implements the leaf node of the ARM11, two-level page table structure.

Frame: A Frame object encapsulates a frame of physical memory that is installable in a VSpace for backing virtual memory. As mentioned in Section 4.3, Frame objects contain no kernel data.

For simplicity, the seL4 API only supports hardware-defined frame sizes. Note that the model does not impose a limit on supporting other page sizes, which are a multiple of the base size. Not supporting these page sizes is a design decision motivated by keeping the interface simple. Performance implications of this decision are discussed in Section 7.3.3.

Creating a VSpace

Shown in Figure 4.7 is an illustrative example of a VSpace, created by using seL4 kernel objects. In the figure, the square box represents the seL4 ASID table and rounded boxes represent different kernel objects required to install a page mapping into an virtual address space. Dark arrows denote capabilities and the dotted ones denote *weak links*, which I explain below.

Using this example, I explain how kernel objects are used to implement a virtual address space and the restrictions enforced by the kernel to guarantee the invariant.

The first step in creating a VSpace is to allocate a PageDirectory object using the retype method. This newly created PageDirectory object, however, is not yet usable as a VSpace.

To make it usable, the resource manager needs to initialise the PageDirectory with a seL4 ASID.

The initialisation is done by invoking the PageDirectory object and passing a capability that authorises the use of a slot in the seL4 ASID table. As depicted in part (a) Figure 4.7, the initialisation operation updates the seL4 ASID table, by copying the memory address of the PageDirectory into the provided slot and updates the PageDirectory capability by storing the seL4 ASID (i.e. table index) in the capability (to reduce the cost of indirection through the seL4 ASID table on an address-space switch, the kernel additionally stores the hardware ASID in the capability). The capability provides the storage required for storing ASID information. Moreover, once a seL4 ASID is used to initialise an address space, the same seL4 ASID cannot be used again unless that original address space is deleted or recycled.

Once a PageDirectory is initialised, it can be used as the VSpace of a thread. Moreover, any number of threads can share the same VSpace.

Other objects required to implement a VSpace can be installed into an initialised PageDirectory: By invoking the PageDirectory and passing a PageTable capability and a virtual address, a resource manager can install the PageTable into that address space. As shown in part (b) of Figure 4.7, after installing the PageTable the kernel updates the PageTable capability by storing the seL4 ASID (of the PageDirectory) and the virtual address to which it is mapped. Similarly, a Frame object can be installed into an address space by invoking the PageDirectory and passing a Frame capability together with a virtual address. After installing the Frame the kernel updates the Frame capability with the seL4 ASID and the virtual address used for the mapping.

seL4 ASIDs provide two main services. First, they act as kernel's internal naming mechanism for identifying an address space to remove a mapping from. Second, they provide a *weak link* between VSpace mappings and the capabilities from which they were derived. I explain the motivation behind weak links through an example.

Consider the address space shown in part (b) of Figure 4.7. This address space consists of a PageTable mapping and a Frame mapping installed within the PageTable. Suppose the resource manager wants to reclaim all the memory from this address space. For this, ideally, the resource manager should first revoke all the Frames, then the PageTables and finally the PageDirectory.

When the resource manager calls revoke on the Frame capability, the seL4 ASID acts as a link to locate the PageTable slot containing the mapping — kernel retrieves the seL4 ASID and the virtual address of the mapping from the Frame capability, locates the PageDirectory by using the seL4 ASID to index the seL4 ASID table, locate the PageTable slot by walking the page-table structure using the virtual address and removes the mapping from the PageTable and from the TLB. Using the same process, the kernel can locate and remove the PageTable mapping.

However, when deleting an address space, a resource manager may not follow this ideal sequence. Instead, the resource manager may delete the PageDirectory, before removing the Frame or the PageTable mappings. The hardware-defined PageDirectory structure does not provide bookkeeping provision to back-track the capabilities from which the mappings are derived from, and update the bookkeeping stored in those capabilities.

One possible solution here is to allocate shadow tables to store the bookkeeping required for back-tracking the capabilities. This option was traded in favour of reducing the memory foot-print of an address space.

Instead of back-tracking the capabilities, seL4 uses weak links through the seL4 ASID

table. When a PageDirectory is revoked (using the bookkeeping stored in the PageDirectory capability) the kernel updates the seL4 ASID table by setting the corresponding PageDirectory address (called *PD addr* in the Figure 4.7) to *NULL* (and does a selective, TLB flush using the hardware ASID). A subsequent attempt to remove a PageTable or a Frame is easily detected and can be ignored — the kernel has already flushed the ASID from the TLB and since the PageDirectory itself is deleted there can be no more TLB entries from the mappings that existed in the address space.

Because of the fact that a PageDirectory can be deleted without updating the capability links, these links are called weak links.

A PageTable can only be used once in the construction of an address space. The restrictions required for efficient enforcement of this condition are discussed further below. Among others, this restriction guarantees that a PageTable used in the deleted address space cannot be installed into another. Hence, the Frame mappings contained within the PageTable — which can now be potentially dangling, due to the removal of the Frame capabilities from which they were generated — are not usable. To reuse a PageTable in another space, the resource manager either has to perform a recycle operation or reclaim the memory used by the PageTable via a revoke operation (on the parent Untyped capability) and reallocate PageTables via retype. Both these operations reinitialise the PageTable.

When a mapping is removed from an address space, the kernel uses the seL4 ASID to flush the TLB. To be exact, the kernel uses the hardware ASID assigned to the seL4 ASID, however since this assignment is fixed we can consider them equal for this discussion. As such, the kernel needs to guarantee that the same PageDirectory is not initialised with two different seL4 ASIDs.

The kernel provides the above guarantee by enforcing the following three restrictions on PageDirectory capabilities: (a) once a PageDirectory capability is initialised with a seL4 ASID it is immutable — that is, any copy made of the original will contain the same (valid) seL4 ASID, (b) the initialised PageDirectory needs to be an uninitialised (i.e. no seL4 ASID) CDT leaf, and (c) kernel disallows imitate operations on uninitialised PageDirectory capabilities.

To explain these restriction, the first one guarantees that none of the CDT ancestors of the PageDirectory capability were initialised — if they were, then the capability should have a valid seL4 ASID. The second invariant guarantees that none of the ancestors can be initialised. This is because these ancestors will not be leafs in the CDT. The final invariant guarantees that any capability copy made from an uninitialised PageDirectory is always a CDT descendent (not a sibling) of the source.

Checking if a PageDirectory capability is initialised (or not) is straight-forward and testing for a CDT leaf can be done in constant time using the pre-order traversal of the logical tree (see Section 4.3.3).

The above restrictions avoid the need to traverse the CDT up to the Untyped capability and back down in all possible CDT branches to establish the status of all capabilities that refer to the same PageDirectory. Depending on the number of capability copies made, such a traversal can be long running.

Frame objects can be shared. For each shared mapping, the resource manager needs to create a copy of the Frame capability and use the copy to install the new mapping. Recall that when a Frame object is installed into a VSpace, the kernel stores bookkeeping information inside the Frame capability to facilitate revocation. By creating a copy of the Frame capability the resource manager provides the storage for bookkeeping the shared mapping. When the Frame object is deleted, the kernel can locate all capability copies

though the CDT and by using the bookkeeping in those capabilities remove all the shared mappings.

Sharing of PageTables, however, is somewhat problematic. We can allow sharing of PageTables using the same technique as Frames — a new capability copy for each shared mapping. Thus, when deleting the PageTable, the kernel can locate and revoke all the shared mappings from PageDirectorys and the TLB. However, the problem arises when un-mapping a Frame installed within a shared PageTable — when PageTable is shared mappings contained within it may appear in the TLB, either with different ASID tags or with different virtual address (or both). There is not enough bookkeeping in the hardware defined data structure to trace the PageTable capability to determine whether the PageTable is shared (see Figure 4.7). Consequently, the kernel needs to flush the entire TLB on every Frame un-map, pessimistically assuming that the PageTable is shared. Flushing the entire TLB is unattractive for performance reasons and hence seL4 disallows the sharing of PageTable.

Disallowing PageTable sharing is a design trade-off to facilitate the use of selective TLB flush of the ARM11 architecture. If need be, the model can accommodate shared PageTables. This option is attractive in particular for an architecture that does not support selective TLB flushes.

The kernel guarantees that PageTables are not shared by enforcing the same three restrictions as with the PageDirectory: (a) once a PageTable capability is installed into a PageDirectory, the capability is updated with an immutable *used* tag, (b) an unused PageTable capability can be installed into a PageDirectory only if it is a CDT leaf, and (c) the imitate operation is disallowed on unused PageTable capabilities. The sufficiency of these restrictions can be explained using the same reasoning given above for PageDirectory capabilities.

4.7 Implementation Details

In this section, I brief on the main implementation details of realising the model on modern hardware. The discussion here is based on seL4::pistachio (see Chapter 7), a prototype realisation of the model for ARM11.

4.7.1 Untyped Memory Abstraction

I have already discussed how untyped memory can be used to allocate any kernel object type. In most cases, the kernel needs to access these objects — for example to examine and modify the contents of a TCB object.

The seL4 kernel accesses all its memory virtually. That is to say, kernel-generated addresses are translated via the MMU. Though it is feasible to configure the kernel to use physical addresses directly, this would imply, among other things, switching the MMU on and off on every kernel entry and exit, causing a significant overhead in handling system calls. To prevent such an overhead, the memory region covered by all the Untyped objects must be accessible to the kernel via a virtual address mappings.

A region of virtual memory, usually allocated towards the high-end of the address space above the user accessible virtual memory region, called the *kernel window*, contains mappings to all the Untyped memory objects. These mappings are privileged — in that only the kernel can access them and are never modified.

Since all Untyped objects and therefore other kernel objects derived from them are mapped into the kernel window, the kernel window size imposes a maximum limit on the kernel objects. For the experimental prototype, this did not cause any concerns — the hardware consists of only 128MB of RAM which fits within the (576MB) kernel window.

Other platforms, however, may have more physical memory than the kernel window. There are three possible solutions for this. First is to increase the size of the kernel window. This is acceptable for most cases and in particular, is an attractive solution for 64bit machines where the virtual memory address space is significantly larger.

However, increasing the kernel window, and therefore reducing the user-accessible VM region is not ideal for virtualisation. Guest OSes, typically have their own algorithms for deciding the memory layout of a process — for example, the guest decides where to place the process stack, heap, shared libraries and so on. These decisions are made by higher-layers of the guest kernel. If seL4's kernel window is large enough to interfere with these policy decisions, then porting the guest OS to seL4 becomes (at best) hard — now we need to make changes to policy layers of the guest. If there is no such interference, then we simply need to keep the page tables used within the guest consistent with those used within seL4. That said, a small increase in the kernel window does not cause serious concerns, but may not be sufficient to map all the available physical memory. As such, this option was not further investigated.

The second possibility is to keep the size of the virtual window fixed, but establish mappings on demand. Investigating this option is beyond the scope of this work.

The seL4 model opts for a third solution. In the process of bootstrapping, the kernel creates "one-to-one" (i.e. with a fixed offset) mappings of the kernel window to a region of free physical memory. If these mappings are insufficient to cover all the available physical memory, then, the remainder is used to allocate Frame objects. These Frame objects are never accessed by the kernel, thus eliminating the need for virtual memory mappings for them within the kernel.

Memory Mapped IO

Most platforms' physical memory contains memory mapped IO regions. For obvious reasons, such a region of memory should not be used to allocate kernel objects.

The seL4 model exports the control of these IO regions to the initial resource manager by encapsulating them as Frame objects. Similar to a normal Frame object, the initial resource manager can either install these Frames into its or some others' virtual address space and thereby gain or delegate access to the device. Moreover, since Frame objects cannot be used to allocate any kernel object, kernel objects are guaranteed not to overlap with any device memory.

Cache Aliasing

An Untyped memory object (parent) can be used to allocate a Frame object (or a number of Frame objects, depending on the size of the Untyped object). A Frame object can be installed as the backing storage for a page within the user-accessible VM region. Moreover, after performing a revoke operation on the parent Untyped capability, the memory region can be reused to implement a different kernel object — say a TCB object — which is accessed by the kernel through a different address within the kernel window. This may lead to cache aliasing — the same physical memory region is accessed using different

virtual addresses. Such aliasing is avoided by flushing the cache when a Frame object, is unmapped during a revoke operation.

Similar aliasing may take place when kernel objects (accessed via the kernel window) are revoked and the memory is used to implement Frame objects. Again, this is avoided by a cache flush. However, the responsibility is delegated to the user-level managers.

4.8 Summary

The seL4 memory management model removes all implicit memory allocations from the kernel — all memory allocations in the kernel are explicit via authorised requests from user-level resource managers. Thus, the allocation scheme makes a direct connection between the authority a user-level resource manager possesses and the amount of kernel memory it may consume.

Since there is a direct relationship between the authority — conferred via capabilities — and the ability to consume a particular region of memory, reasoning about memory consumption boils down to reasoning about capability dissemination.

In the next Chapter (Chapter 5) I discuss the model used for controlling the dissemination of capabilities.

Chapter 5

Formal Model of Authority

Having described the memory allocation model of the seL4 kernel in Chapter 4, I now introduce a formal model governing the dissemination of authority and memory allocation. Recall from our previous discussions that seL4 unifies memory management with the capability based authority model. I call this machine-checked specification the *abstract protection model* or simply the *protection model*.

As I have discussed in Section 2.2, a protection model is an abstract, formal, specification that captures: (a) the authority distribution of the system (*protection state*), and (b) the rules governing the mutations of the above authority distribution (*access-control model*) [Lan81].

In other words, the protection model is an abstract, formal model that captures the distribution of authority and how this distribution evolves over time. The purpose of such a model is to facilitate reasoning — it provides a framework for analysing the feasibility of enforcing different *policies* related to protection state.

In this chapter, I present the protection model and in Chapter 6 I analyse its formal properties. In particular, the analysis shows that the model is capable of enforcing (at least) two useful policies relating to access control — *spatial partitioning* and *isolation*. I introduce these policies later in Chapter 6.

The protection model is based on the take-grant model [LS77]. It deviates from the original take-grant model (and any such model based on take-grant) in two important aspects. First, the protection model captures the behaviour of untyped memory objects — making it feasible to reason about physical memory usage of an application, which is not possible with the original model. Second, for reasons I discuss latter, the model does not employ the take rule.

The formalisation of the model and its analysis in the next chapter (see Chapter 6) are developed entirely in the interactive theorem prover *Isabelle/HOL* [NPW02]. Thus making the intuitive graph diagram notation that is commonly used for this type of specification and analysis fully precise.

Moreover, parts of this chapter and some parts of the following chapter formed the basis for the following publications [EKE08, EKE07], which are coauthored with Dr. Gerwin Klein and Dr. Kevin Elphinstone.

The remainder of this chapter is structured as follows. In Section 5.1, I introduce the classical take-grant access control model and the extensions proposed. Section 5.2 briefly outlines the Isabelle/HOL system and the notation used in the formal specification and the formal proofs in the following chapter (see Chapter 6). The seL4 protection model is described in Section 5.3 which is followed by an informal description of how the protection

model relates to the concrete kernel (see Section 5.4).

5.1 The Take-Grant Model

Protection models provide a formalism and a framework for specifying, analysing and implementing security policies. Such a model consists of: (a) finite set of access rights, and (b) a finite set of rules for modifying the distribution of these access rights. The *safety* analysis of a model then determines, for the given set of rules in the model and an initial distribution of access rights, whether or not it is possible to reach a state in which a particular access right α is acquired by a subject that did not possess it initially. If there exists an algorithm that decides safety, then the model is said to be *decidable*.

The classical access model used in capability systems is the take-grant model, originally proposed by Lipton and Snyder [LS77] and later enhanced by many authors [BS79, Sny81, Bis81, Bis96, SW00]. The model is decidable in linear time [LS77] — the complexity of the decision algorithm varies linearly.

Suppose we want to know the possibility of subject S_1 (i.e. an application) accessing a particular object O_1 (i.e. a resource) in some way; say for writing. This falls into a safety question — can the S_1 acquire write authority over O_1 — and the answer can be formulated by a safety analysis on the take-grant model. Bishop et. al. [BS79] later extended the analysis to cover *de facto* rights — access rights implied by a given distribution of authority. As an illustrative example, assume that there is a subject S_1 with write authority over subject S_2. Moreover, S_2 has write authority to subject S_3. Then, S_1 possesses a de facto write authority to subject S_3; because data that S_1 writes to S_2 can then be transfered to S_3 via S_2. Though S_1 does not possess direct (or *de jure*) write authority to S_3, it can still write to S_3 via S_2. The above example is an instance of the *find* rule defined in [BS79].

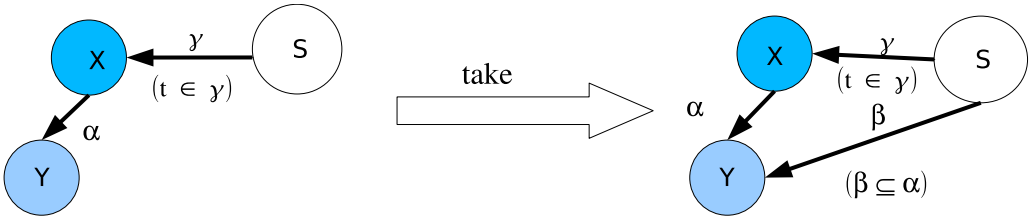
The take-grant model represents the protection state of the system as a directed graph; where nodes represent subjects or objects in the system, and labelled directed arcs represent authority (i.e. capabilities) possessed by a subject over the object pointed to by the arc. An outgoing arc from X to Y with label α means X possesses α authority over the object Y (see Figure 5.1).

If the set of of access rights of the system is R, then any label α on an arc is a nonempty subset of R ($\alpha \subseteq R$). While R might vary, there are two rights that deals with the propagation of authority — the take right (t) and the grant right (g). They play a special role in dissemination of authority.

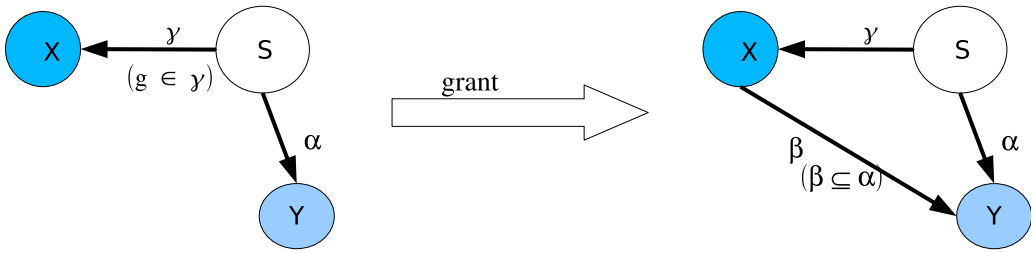
The system operations that modify the authority distribution are modelled as graph mutation rules. There are number of formulations of the basic model, with different graph mutation rules in the literature. The specific rules that I discuss here are [LS77]: *take, grant, create* and *remove*. Take and grant rules are used to propagate existing authority from one node to another. The create rule adds a node to the graph and an arc connecting the new node to an existing one. The remove rule takes away part of the access rights from an arc or removes the arc entirely.

A detailed description of the take-grant model and its rules can be found elsewhere; including in [Har84, Bis02]. Following is a detailed description of the graph rewriting rules based on the definitions used by Lipton and Snyder [LS77]:

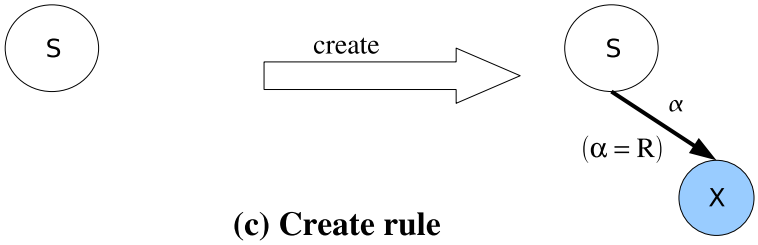
- **take rule:** Let S, X, Y be three distinct vertices in the protection graph. Let there be an arc from X to Y labelled α, and from S to X with a label γ such that $t \in \gamma$.



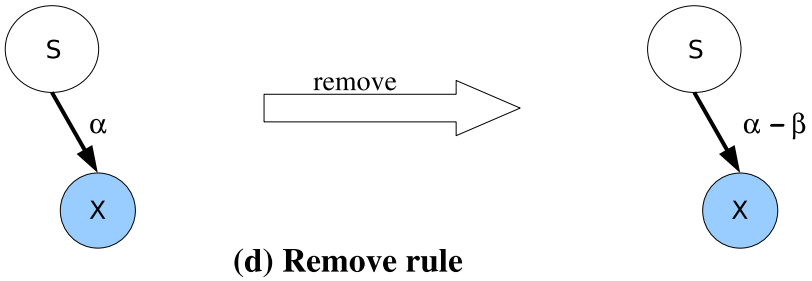
(a) Take rule



(b) Grant rule



(c) Create rule



(d) Remove rule

Figure 5.1: Take-grant authority distribution rules

Operation	Description	Take-grant operation
SysCreate	Create new subjects/objects	modified create rule
SysRemove	Remove a capability	modified remove rule
SysRevoke	Remove a set of capabilities	-
SysGrant	Propagate a capability	grant rule
SysNoOP	NULL operation	-
SysRead	Reading information	-
SysWrite	Writing information	-

Table 5.1: Summary of the operations in the seL4 protection model.

Then the take rule defines a new graph by adding an edge from S to Y with the label $\beta \subseteq \alpha$. Part (a) of Figure 5.1 represents an application of the take rule.

- **grant rule:** Let S, X, Y be three distinct vertices in the protection graph. Let there be an arc from S to Y labelled α, and from S to X with a label γ such that $g \in \gamma$. Then the grant rule defines a new graph by adding an edge from X to Y with the label $\beta \subseteq \alpha$. Part (b) in Figure 5.1 is a graphical illustration of this rule.

- **create rule:** Let S be a vertex in the protection graph. Then the create rule defines a new graph by adding an new node X and a arc from S to X with a label α. Part (c) of Figure 5.1 represents an application of this rule.

- **remove rule:** Let S, X be vertices in the protection graph. Let there be an arc from S to X with a label α. Then the remove rule defines a new graph by deleting β labels from α. If $\alpha - \beta = \{\}$ then the arc itself is removed. The operation is shown in part (d) of Figure 5.1.

These graph rewriting rules model different operations that takes place in a system. The take and grant rules model the propagation of access rights. The grant rule models a situation where a subject propagates all or part of the authority it possesses to another. The take rule also models a propagation of access right(s). However, unlike the grant rule, where the dissemination of authority is controlled by the sender, the take rule models a situation where the dissemination of authority is governed by the receiver side — the take rule allows a subject to acquire authority possessed by another. The creation of a new subject (e.g. a process) or an object (e.g. a file) is modelled by the create rule. Finally, the remove rule models the revocation of access rights.

5.1.1 The seL4 protection model — Informal Introduction

The list of the operations in the seL4 protection model is given in Table 5.1. The first column of this table gives the names of the protection model operations. For each operations, the second column provides a brief description and the third column shows the relationship of the operation to the take-grant rules specified above.

The operations in the seL4 protection model modify the classic take-grant rules in several aspects. The most significant of these modifications is the create rule, or the *SysCreate* operation as it is called in the seL4 protection model. Adding a new node to the protection graph corresponds to allocating a new object in the concrete kernel. In classic take-grant, there is no restriction on adding new nodes to the protection graph: each entity has the

potential to add new nodes to the graph without affecting the potential of another to add another node to the graph. Consequently, the take-grant model is not expressive enough to capture the semantics of Untyped memory: the mechanism used by seL4 to control the allocation of physical memory. In the seL4 protection model, the create rule is only applicable if there is an outgoing arc with *Create* authority.

The second modification is that seL4 protection model uses a simpler remove rule (or a *SysRemove* operation) than that of take-grant. The protection model removes the capability, or the whole arc, instead of removing part of its label. In the protection model the only way to diminish authority (i.e. remove part of the label) is by removing the capability and granting a new one with diminished authority.

In addition to removing a single capability, the seL4 protection model provides an operation called *SysRevoke* which removes a set of capabilities from the protection state. The main purpose of SysRevoke is to mimic the revoke operation of the seL4 kernel. A detailed discussion of kernel's revoke operation is provided elsewhere in Section 4.5. None of the graph rewriting rules in take-grant represent the revoke operation. However, it can be simply be thought of as a set of multiple applications of remove.

Furthermore, the seL4 protection model does not employ the take rule. This has the advantage of giving each subject control over the distribution of its authority at the source. All authority distributions in the protection model are governed by the sender using the *SysGrant* operation; which is similar to grant rule in take-grant.

However, the protection model, can be easily extended to support the take rule or the effect of take on authority propagation can be modelled with an appropriate grant right [Sha99].

In addition to the above rules, which modify the authority distribution, the seL4 protection model defines three more operations that do not mutate the protection state: *SysNoOP*, *SysRead* and *SysWrite*. Out of these, the last two operations model accessing data for reading and writing, respectively. Even though these two operations do not mutate the protection state, they have preconditions that are visible to the protection model. As the name implies, SysNoOP does not perform any function. The motivation behind its inclusion is discussed later in Section 5.4.

5.2 Isabelle/HOL system

This section gives a short introduction to Isabelle/HOL — the interactive theorem prover used for this work. It is by no means comprehensive, but it introduces the Isabelle/HOL notation that is used in this thesis. For a complete treatment I recommend [NPW02].

Isabelle is a generic system for implementing logical formalisms. It is generic in the sense that it can be instantiated with different object logics. Isabelle/HOL is the specialisation of Isabelle for HOL — which stands for *Higher-Order-Logic*. These formalisations are organised in an acyclic graph of *theories*. Each theory contains a set of declarations, definitions, lemmas and theorems. From a technical point of view, lemmas and theorems are the same — a theorem is a lemma with contextual significance. For the most part, the notations used in Isabelle/HOL coincides with that of standard mathematics.

HOL is a typed logic, whose type system is similar to that of functional programming languages like ML [MTH90] or Haskell [Bir98]. Typed variables are written `'a`, `'b`, etc. The notation `x::'a` means that HOL term `x` is of type `'a`. HOL provides a number of base types, in particular `bool`, the type of truth values, and `nat`, the type of natural numbers. Type constructors are supported as well: `nat list` is a list of natural numbers and `nat`

set denotes a set of natural numbers. The empty list is written *[]* and the list constructor is written with the infix *x·xs*.

New data types can be introduced in a number of ways. However, in this thesis, I am using only three ways to introduce new types. The command **datatype** defines a new algebraic data type. For example, the four primitive access rights of a capability are defined by:

datatype *rights* = *Read* | *Write* | *Grant* | *Create*

Expressions such as **types** *caps = capability set* are used for simple abbreviations. Finally, the **record** command introduces tuples with named components. In the example;

```
record point =
  x :: nat
  y :: nat
```

the new type *point* is a tuple of two *nat* components. If *p* is a *point* (i.e. *p::point*), the term *x p* stands for the *x*-component of *p*, and *y p* for its *y*-component. If the *x*-component of some variable *q* is *2* and *y*-component is *3*, then *q* is written $q \equiv (\!|x = 2, y = 3|\!)$. The update notation $q(\!|y := 4|\!)$ leads to $(\!|x = 2, y = 4|\!)$.

The space of total functions is denoted by $\Rightarrow$. *Function update* is written *f(x := y)*. When a function is updated, for example after performing *f(x := y)* operation, the updated function if called with *x* returns *y*. Sequential updates are written *f(x:= a, y := b)*. Applying a function *f* to a set *A* is written *f ‘ A* $\equiv$ *{y | ∃x∈A. y = f x}*.

The symbol $\Longrightarrow$ is used for implication when writing lemmas to separate antecedent and conclusion. $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A) \ldots)$.

Isabelle proofs can be augmented with LaTeX text. This presentation mechanism is used to generate the text for all of the specifications, definitions, lemmas and theorems in this thesis — they are directly from the machine-checked Isabelle sources.

5.3 The seL4 Protection model

In this section, I present the formal protection model of seL4 using the Isabelle/HOL system.

5.3.1 Semantic Entities

The protection state consists of a collection of *objects*. There is no distinction between active subjects and passive objects. Instead, they are collectively called *entities*. Each entity is identified by its memory address which is represented by a natural number. In the usual graph models found in the literature, entities would be nodes and their addresses the names or labels of these nodes.

types *entity_id* = *nat*

There are four primitive access rights in the model. As usual, each right represents authority to perform particular operation.

datatype *rights* = *Read* | *Write* | *Grant* | *Create*

Out of these access rights, *Read* and *Write* have the obvious meaning — they authorise reading and writing of information. Similar to the take-grant model [LS77], *Grant* is

sufficient authority to propagate a capability to another entity. The *Create* right confers the authority to create new entities.

Each capability has associated with it an *entity_id*, which identifies an entity and a set of access rights that defines the operations the holder of the capability is authorised to perform on that entity. Thus, a capability or *cap* is defined in the following manner;

record *cap* = *entity :: entity_id*
rights :: rights set

In the protection model, an entity only contains a set of capabilities. Thus an entity is defined as follows:

record *entity* = *caps :: cap set*

Entities have no additional authority beyond what they possess as capabilities.

5.3.2 Protection State

The protection state of the system consist of two fields:

record *state* = *heap :: entity_id ⇒ entity*
next_id :: entity_id

The component *heap* stores the entities of the protection model. It is modelled by a total function which returns an *entity* given the *entity_id*. If the provided *entity_id* does not corresponds to any existing entity, then the *heap* return *nullEntity*, which is defined in the following manner.

```
nullEntity:: entity
nullEntity ≡ (|caps = {}|)
```

Thus, we require a test to determine whether or not a given *entity_id* corresponds to an existing entity in that protection state. This test is facilitated via the *next_id* component in the state. The heap can be viewed as an array that contains entities from addresses *0* up to and excluding *next_id*. The *next_id* is the next free slot for placing an entity without overlapping with any existing one.[1] This setup allows a simple test to determine the existence of an *entity* for a given *entity_id*:

```
isEntityOf :: state ⇒ entity_id ⇒ bool
isEntityOf s e ≡ e < next_id s
```

In a well formed state all the existing capabilities should only point to existing entities: the entities stored in *heap* contain capabilities, which again contain references to other entities. In any run of the system, these references should only point to existing entities. We call such system states *sane*:

```
sane ::   state ⇒ bool
sane s ≡  (∀ c∈all_caps s. isEntityOf s (entity c)) ∧
          (∀ e. ¬ isEntityOf s e ⟶ caps_of s e = {})
```

where

[1] An alternative to this model is the use of a partial function for the heap. I found working with a total functions slightly more convenient in Isabelle.

```
caps_of :: state ⇒ entity_id ⇒ cap set
caps_of s r ≡ caps (heap s r)

all_caps :: state ⇒ cap set
all_caps s ≡ ⋃e caps_of s e
```

The term `caps_of s r` denotes the set of all capabilities contained in the entity with address `r` in state `s`, and `all_caps s` denotes all capabilities in the given state `s` — the union of the capabilities over all entities in the system.

Adding a new entity to the system state is done via the *SysCreate* operation. This operation uses the `next_id` of the current `state` as the `entity_id` of the newly created `entity`. Thus, the create operation will guarantee that for any `sane` state `s`:

- the new `entity` will not overlap with any of the existing ones, and
- no capability in the current `state` will be pointing to the heap location of the new entity.

5.3.3 Operational Semantics

Next, I formally introduce the operations of the seL4 protection model, captured in data type `sysOPs`:

```
datatype sysOPs = SysNoOP entity_id
                | SysRead entity_id cap
                | SysWrite entity_id cap
                | SysCreate entity_id cap cap
                | SysGrant entity_id cap cap rights set
                | SysRemove entity_id cap cap
                | SysRevoke entity_id cap
```

The rationale behind these abstract operations is to capture the seL4 kernel operations as closely as possible — even though the thesis does not provide or claim the feasibility of providing, a formal connection between the two. `SysRead` and `SysWrite` operations capture the reading and writing of information, from and to objects, respectively. These two operations have preconditions that are visible at the abstract level, but does not modify the authority distribution and hence does not mutate the protection state. The `SysNoOP` does not cause a change to the protection state either. It is included in the protection model to mimic seL4 kernel operations such as sending a non-blocking message to a thread not willing to accept — which results in a dropped massage.

The first argument of each operation indicates the `entity` initiating that operation. The second argument is the capability being invoked and the third argument depends on the operation being performed. The third argument for `SysCreate` is a capability that points to the destination entity for placing the newly created capability, for `SysGrant` it is the capability that is transported and for `SysRemove` it is the capability that is removed. The fourth argument to `SysGrant` is a mask for the access rights of the transported capability. The `diminish` function reduces access rights according to such a mask `R`;

```
diminish :: cap ⇒ rights set ⇒ cap
diminish c R ≡ c(|rights := rights c ∩ R|)
```

Through the diminish function, the entity initiating the `SysGrant` operation is at liberty to transport a subset of the authority it possesses to the receiver.

Any operation is allowed only under certain preconditions, encoded by `legal`.

```
legal :: sysOPs ⇒ state ⇒ bool
legal (SysNoOP e) s          = isEntityOf s e
legal (SysRead e c) s        = isEntityOf s e ∧ c ∈ caps_of s e ∧
                               Read ∈ rights c
legal (SysWrite e c) s       = isEntityOf s e ∧ c ∈ caps_of s e ∧
                               Write ∈ rights c
legal (SysCreate e c1 c2) s  = isEntityOf s e ∧ {c1, c2} ⊆ caps_of s e ∧
                               Grant ∈ rights c2 ∧ Create ∈ rights c1
legal (SysGrant e c1 c2 r) s = isEntityOf s e ∧ {c1, c2} ⊆ caps_of s e ∧
                               Grant ∈ rights c1
legal (SysRemove e c1 c2) s  = isEntityOf s e ∧ c1 ∈ caps_of s e
legal (SysRevoke e c) s      = isEntityOf s e ∧ c ∈ caps_of s e
```

The `legal` function models the authority checks that are performed before allowing an operation. Firstly, it checks if the entity initiating the operation exists in that state. Secondly, all the capabilities specified in the operation should be in the entity's possession at that state. Finally, the capabilities specified should have at least the appropriate permissions.

Each operation listed under `sysOPs` mutates the protection state in a particular way, provided that the operation is legal in the starting state. The mutation performed by each operation is defined using an Isabelle function. For example, the state modifications done by `SysCreate` operation are defined in a function called `createOperation` and the mutations performed by `SysGrant` is shown in `grantOperation` and so on. The relationship between `SysCreate` and `createOperation`, for example, is analogous to that of a system call number and the function implementing the system call — here `SysCreate` corresponds to the system call number and the `createOperation` is the function implementing that system call.

I now define how each of the operations mutates the protection state, assuming it is started in a legal state. The `SysCreate` and `SysGrant` operations add capabilities to entities.

```
createOperation e c1 c2 s ≡
let nullEntity = (|caps = {}|);
    newCap = (|entity = next_id s, rights = allRights|);
    newTarget = (|caps = {newCap} ∪ caps_of s (entity c2)|)
in (|heap = (heap s)(next_id s := nullEntity, entity c2 := newTarget),
     next_id = next_id s + 1|)
```

where

```
allRights ≡ {Read, Write, Grant, Create}
```

The `SysCreate` operation allocates a new entity in the system heap, creates a new capability to the new entity with full authority, and places this new capability in the entity pointed to by the c_2 capability. For ease of reference, I will call the entity which receives the new capability as the *parent* of the newly created entity.

The create operation consumes resources in terms of creating a new entity in the heap. So, the subject initiating this call is required to provide and invoke an untyped capability c_1. In the abstract level, an untyped capability is a capability with the `Create` right. This is why the `legal` function checks for `Create` among the set of access rights of the capability c_1. Moreover, since this operation is placing a capability in the parent entity, the capability which names the parent; i.e. c_2, should possess the `Grant` right among its set of access

rights (see the definition of legal for *SysGrant*).

Note that the newly created entity has no capabilities within it (see the definition of *SysCreate*). Since all authority is conferred via capabilities, this means that soon after the creation, the new entity has no authority. After the new entity is created, the parent can propagate all or part of its authority to the new entity by grant operation(s).

```
grantOperation e c1 c2 R s ≡
s(|heap := (heap s)(entity c1 := (|caps = {diminish c2 R} ∪
caps_of s (entity c1)|))|)
```

The *SysGrant* operation, similar to *SysCreate*, adds a capability to the entity pointed to by c_1. However, unlike with *SysCreate*, the new capability is a (potentially) diminished copy of the existing capability c_2. Note that the entity initiating this operation can only propagate a capability that is in its possession and should possess *Grant* right over the entity receiving the diminished copy.
The *SysRemove* and *SysRevoke* operation remove capabilities from system entities.

```
removeOperation e c1 c2 s ≡
s(|heap := (heap s)(entity c1 := (|caps = caps_of s
(entity c1) - {c2}|))|)
```

The *SysRemove* operation removes the specified capability c_2 from the entity denoted by c_1.

The *SysRevoke* operation is a repeated application of *SysRemove*. It is used to remove all the *CDT* children of a given capability. The implementation of the CDT and its main purpose is discussed in Chapter 4.3.3. In brief, the seL4 kernel internally tracks in the capability derivation tree (or the CDT) how capabilities are derived from one another with create and grant operations. The revoke operation is facilitated by this information in the CDT. The protection model however, does not model the CDT explicitly, instead the model uses an under-specified function *cdt* that returns, for the current system state and the capability to be revoked, a list that describes which capabilities are derived from the given one and therefore are to be removed and from which entities:

```
cdt :: state ⇒ cap ⇒ (cap × cap list) list
```

Each element in the above list has two components (*cap × cap list*); the capability and a list of capabilities. The capability identifies an entity in the protection model and the second component specifies a list of capabilities that are to be removed from that particular entity. Given one of these elements, *removeCaps* function removes the capabilities specified in the list by repeatedly calling *SysRemove*:

```
removeCaps e (c, cs) s ≡ foldr (removeOperation e c) cs s
foldr f [] a         = a
foldr f (x # xs) a   = f x (foldr f xs a)
```

The revoke operation is then just a repeated call of *removeCaps* for each element in the list returned from the *cdt* function.

```
revokeOperation e c s  ≡ foldr (removeCaps e) (cdt s c) s
```

Note that CDT is neither an axiom in the model nor an assumption in the proofs (presented in the next chapter). The model simply says there exists a function CDT which returns for a given capability a list of capabilities, but does not define how the list is computed. That is, the proofs will hold for any instance of the CDT.

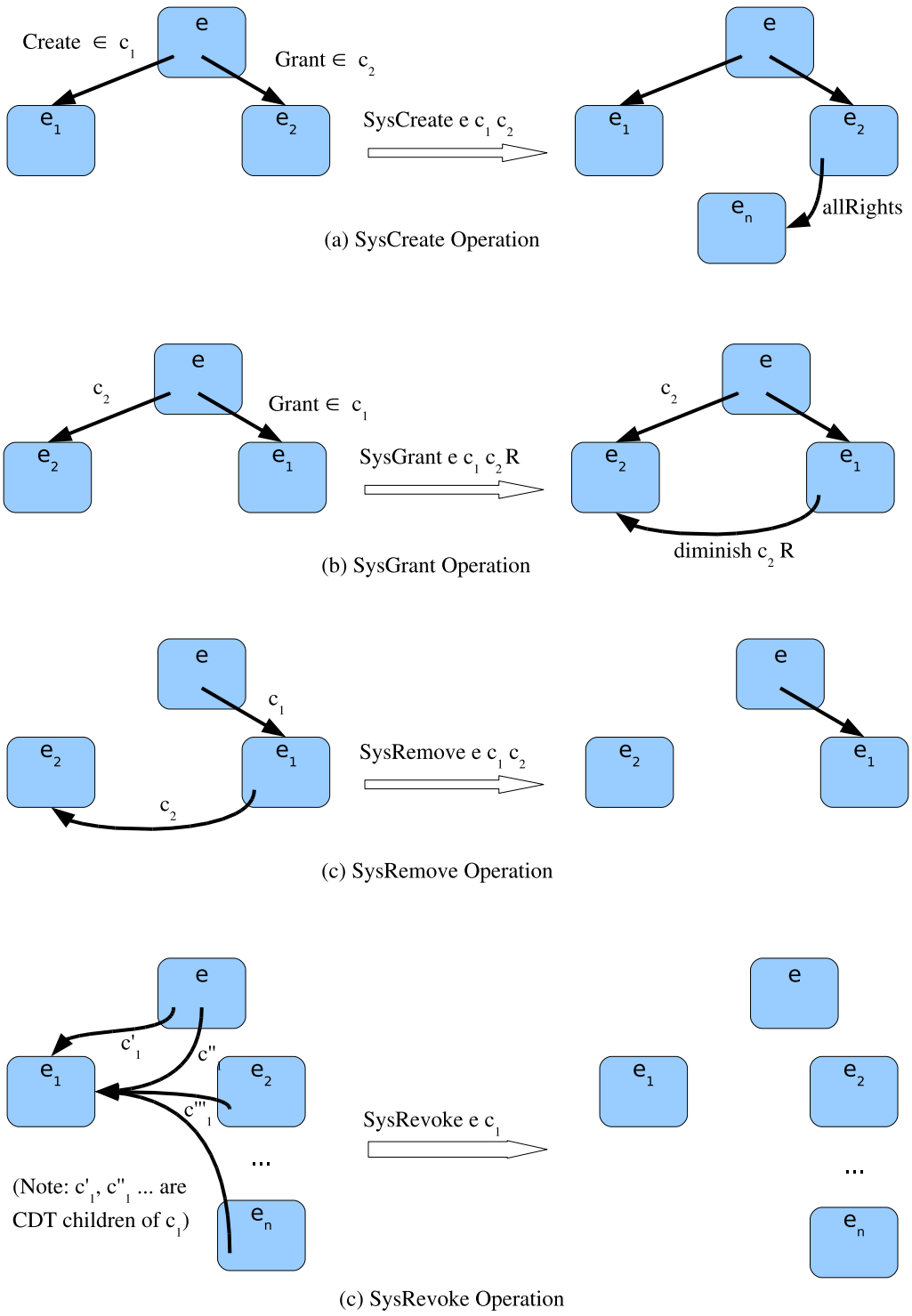


Figure 5.2: Graphical representation of seL4 authority distribution rules

The motivation behind under-specifying the CDT is to keep the model simple as possible — the model only captures the behaviour of operations upon which the proved properties rely on. On the other hand, due this under-specification it becomes infeasible to reason about some other, arguably interesting, properties. For example, one cannot reason about what entity might reclaim a resource from another. If this type of reasoning is required, then the model has to be augmented with an instance of the CDT which specifies how the list is computed. Further, note that the proofs (in the next chapter) will hold for any such instance of the CDT function.
Figure 5.2 shows the definitions of the above operations as graph rewriting rules.

Having explained the operations in the model, I now turn to executing a command. A single step of execution in the model is summarised by the function *step*. The first task of *step* is to check the legality of the command in that state. If the command is legal, then it is passed to *step'*, which decodes the command and calls the appropriate operation.

```
step' :: sysOPs ⇒ state ⇒ state
step' (SysNoOP e) s = s
step' (SysRead e c) s = s
step' (SysWrite e c) s = s
step' (SysCreate e c1 c2) s = createOperation e c1 c2 s
step' (SysGrant e c1 c2 R) s = grantOperation e c1 c2 R s
step' (SysRemove e c1 c2) s = removeOperation e c1 c2 s
step' (SysRevoke e c) s = revokeOperation e c s

step :: sysOPs ⇒ state ⇒ state
step cmd s ≡ if legal cmd s then step' cmd s else s
```

The *legal* function checks the preconditions required for each command. If a command does not satisfy the preconditions then it is not allowed to proceed and hence there is no modification to the protection state.
The state after a whole system run, i.e., executing a list of commands, is then just repetition of *step* (note that the list of commands are read from right to left):

```
execute :: sysOPs list ⇒ state ⇒ state
execute = foldr step
```

5.3.4 The Initial System State

Up to now, I have discussed the model, its operations and in particular, how a sequence of commands can be executed. Suppose we have a list of commands that we would like to execute. Now, the question is from what state should we start the execution?

The model defines an *initial protection state*, from which a given sequence of commands can be executed. I use the notation s_i to denote the initial protection state.

There is only one entity that exists in state s_i — a *initial resource manager*. The initial resource manager possesses full rights to all the system resources conferred via capabilities. In the protection model, these capabilities are folded into one.

$$s_i \equiv (\!|\mathit{heap} = [0 \mapsto \{\mathit{allCap}\ 0\}],\ \mathit{next\_id} = 1|\!)$$

where

```
allCap e    =(|entity = e, rights = allRights|)
allRights   ={Read, Write, Grant, Create}
```

The notation `[0 ↦ {allCap 0}]` stands for an *empty heap* where the position `0` is overwritten with an object that has `{allCap 0}` as its capability set. As one would expect the definition of empty heap is as follows:

$$\mathit{emptyHeap} \equiv \lambda x.\ \mathit{nullEntity}$$

Starting from this initial state, the protection state evolves by executing the operations defined by in the protection model.

5.3.5 Fundamental Invariant of the Model

A `sane` system state is a condition required for proving all most all of the security theorems in Chapter 6. A sane state guarantees two important properties of the model required for the these security theorems: (a) there are no dangling reference within capabilities, and (b) the newly added entity will not overlap with an existing one.

By investigating the definition of the initial state s_i — the state from which we start executing — we see that it is sane:

Lemma 1. *The initial state created for the resource manager (s_i) is* `sane`*. In Isabelle we write:*

$$\mathit{sane}\ s_i$$

Proof. By unfolding the definition of state s_i and `sane`. □

Moreover, sanity is preserved by `step`. In Isabelle, I proved the following lemma:

Lemma 2. *Single execution steps preserve sanity:*

$$\mathit{sane}\ s \Longrightarrow \mathit{sane}\ (\mathit{step}\ a\ s)$$

Proof. By case distinction on the command to be executed, unfolding definitions, and observing that no operation creates overlapping objects or references to non-existing objects. □

Next, I lift the above lemma to a command sequence by induction and prove:

Lemma 3. *Execution preserves sanity:*

$$\mathit{sane}\ s \Longrightarrow \mathit{sane}\ (\mathit{execute}\ \mathit{cmds}\ s)$$

Proof. By induction over the command sequence and Lemma 2. □

All possible states of the protection model, created by any run of the system are derived by executing some sequence of commands starting from the initial state s_i. Since, this initial state, s_i, is sane (from Lemma 1) and since execution preserves sanity (from Lemma 3), we see that all system states reached by any run of the system are sane.

5.4 Informal Correspondence to seL4

The formalism I presented in Section 5.3 provides a framework for reasoning about the ability to enforce access control policies. The analysis in Chapter 6 affirms that the protection model is capable of enforcing at least two useful policies. One important question, however, is: What does this proof means for the concrete kernel?.

Concrete Object Type	Representation	Possible Access Rights
Untyped	Entity with no capabilities	$Create$
TCB	Entity with capabilities	$Grant$, $Read$ and $Write$
Endpoint	-	-
AsyncEndpoint	-	-
CNode	Entity with capabilities	$Grant$
VSpace	Entity with capabilities	$Grant$
Frame	Entity with no capabilities	$Read$ and $Write$
InterruptController	Entity with capabilities	$Grant$
InterruptHandler	Entity with no capabilities	$Write$
seL4 ASID	Entity with no capabilities	$Write$

Table 5.2: Correspondence between concrete kernel objects and protection state entities.

A full formal treatment of this topic, is beyond the scope of this thesis. Note that the thesis does not either, claim to connect or claim the feasibility of connecting the protection model with the seL4 kernel. What is provided in this section is an informal description of the relationship between the protection model and the seL4 kernel.

The approach and the feasibility of connecting the protection model with a formal model of the concrete seL4 kernel — developed by the *L4.verified* team [EKK06,EKD+07, CKS08] — is discussed elsewhere [Boy09].

In the remainder of this section, first I discuss, informally, the connection between the protection state and the concrete kernel state. Then I focus on the informal relationship between the system calls of the concrete kernel and the operations of the protection model.

5.4.1 Concrete and Abstract Protection States

The concrete kernel state consists of objects. Roughly, there are two main types of kernel objects—objects that directly implement kernel operations and objects that support in the implementation of kernel operations. Thread Control Block (TCB) objects which implements threads, for example, fall into the former category while Endpoint objects which acts as rendezvous points for synchronous interprocess communication falls into the latter. Note that this distinction is not clear-cut, I introduced it to aid our discussion.

In the protection model, all object types that directly implement kernel services are folded into entities. The capabilities stored inside these kernel objects are modelled by the capability set of corresponding entity — which can be *empty* in a case where the concrete object does not store authority. In addition to authority, these objects have data (or meta data) stored in them. The abstract protection model captures only the operations that can be performed on this data. This is achieved by modelling the access rights. However, the model does not attempt to model the actual data stored in the object.

The two types of endpoint objects implemented by the concrete kernel support in establishing communication channels between threads. The functionality of these two object types is modelled implicitly by modelling the impact of the service; the protection model captures the communication channel between entities rather than the rendezvous point used in establishing the channel.

Table 5.2 shows a summary of how each concrete kernel object is modelled using entities. The first column of this table gives the concrete kernel object type, the second column

shows its representation at the abstract level and the final column indicates the possible access rights over the abstract entity.

Given below is a detailed discussion, highlighting the important points on how concrete objects are modelled at the abstract level and the limitations in doing so.

Discussion

Each thread in the concrete kernel is associated with a capability address space; which can be shared with another. The capability address space is essentially a collection of *CNode* objects—a set of slots for storing capabilities. At the abstract level, a thread is represented by an entity. Recall the definition of an entity: it contains a set of capabilities. These capabilities represent the capability address space the thread is associated with. If two threads share the same capability address space, then at the abstract level they become a single entity.

However, currently the model does not *directly* support partial sharing of address spaces. For the correspondence to hold it is assumed that CNodes are either, not shared between threads — even though there is room in the allocation model for shared CNodes — or the implication of such sharing is modelled indirectly by a method described below.

The main limitation that arises with the lack of partial sharing is when capabilities are either copied or deleted from the shared CNode. This capability modification is now visible to all entities sharing the space. In the current model, shared CNodes can still be modelled *indirectly* by duplicating the operation across the entities sharing the space — copying or deleting the capability into or from all the entities sharing the address space.

Similar to CNodes, Sharing of a page tables (second level tables) also denotes a partially shared address space. However, recall from our discussion in Chapter 4 the memory allocation scheme disallows the sharing of page tables.

Besides threads (or TCB objects) and CNodes, there are number of other concrete kernel objects: Frame objects for backing virtual memory, Untyped Memory objects, page-table objects (VSpace) and seL4 ASID for implementing virtual memory, the InterruptController object which describes the current interrupt association of the system and InterruptHandler objects for acknowledging interrupts. In brief, all these concrete object types appear as entities at the abstract level. The abstract protection model is concerned with the authority distribution and it's mutations. Therefore, most semantic differences between concrete objects becomes irrelevant at this level.

An abstract Frame object is modelled as an entity with no capabilities within it. Moreover, all the capabilities pointing to the entity, will have a subset of $\{Read,\ Write\}$ as their permissions. Similarly, an entity corresponding to an Untyped Memory object also has an *empty* capability set, but any capability pointing to it can only have the *Create* permission.

The InterruptController object stores, for each interrupt, a capability that should be invoked to deliver the interrupt. These capabilities are modelled by the capability set of the abstract entity corresponding to the InterruptController. Acknowledging an interrupt via the InterruptHandler object is modelled as a write operation to the corresponding entity.

Page table objects are restricted capability storage or CNodes. They are restricted in the sense that they can only store capabilities to Frame objects and their structure is dictated by the underlying hardware. These restrictions are not related to the protection model. As such page tables are modelled in the same way as CNodes. Each thread in the concrete kernel is associated with a collection of page tables—or a *VSpace*. The contents of the page tables of a thread in the concrete system is captured by the capability set in the corresponding

Capability Type	Concrete Kernel	protection model
Untyped	Retype	sequence of $SysCreate$
	Revoke	$SysRevoke$
TCB	ThreadControl	$SysNoOP$, $SysGrant$
	Exchange Registers	$SysWrite$ or $SysRead$
	Yield	$SysNoOP$
Synchronous IPC (Endpoint)	Send IPC	$SysWrite$ or $SysNoOP$
	Wait IPC	$SysRead$
	Grant IPC	$SysWrite$, $SysGrant$ or $SysNoOP$
Asynchronous IPC (AsyncEndpoint)	Send Event	$SysWrite$
	Wait Event	$SysRead$
CNode	imitate	$SysGrant$
	mint	$SysGrant$
	Remove	$SysRemove$
	Revoke	$SysRevoke$
	Move	$SysGrant$, $SysRemove$
	Recycle	$SysRevoke$, sequence of $SysRemove$
VSpace	Install Mapping	$SysGrant$
	Remove Mapping	$SysRemove$
	Remap	$SysRemove$, $SysGrant$
	initialise	$SysNoOP$
Frame	-	-
InterruptController	Register interrupt	$SysGrant$
	Unregister interrupt	$SysRemove$
InterruptHandler	Acknowledge interrupt	$SysWrite$
seL4 ASID Table	Associate VSpace	$SysNoOP$
	Disassociate VSpace	$SysNoOP$

Table 5.3: Relationship between the operations of the concrete kernel and those of the protection model.

entity at the abstract level.

Since the contents of the CSpace and the VSpace of a thread is abstracted by the capability set within the entity, the ability to add capabilities to the CSpace or the VSpace of a thread, in the concrete kernel, is modelled by a capability with *Grant* authority over that particular abstract entity.

5.4.2 Concrete and Abstract Operations

Each operation in the concrete kernel maps to one or more operations in the protection model.

Table 5.3 shows how the concrete kernel operations are modelled by those in the protection model. The first column of this table, indicates the object type which implements the operation. The second column provides a descriptive name for the concrete operation. The last column shows how the given operation is modelled in the abstract protection model.

Broadly speaking, the concrete kernel performs three main types of operations: (a) creation of new kernel objects, (b) adding or removing capabilities to or from kernel objects,

and (c) writing or reading information to or from kernel objects. The creation of a new concrete kernel object maps to $SysCreate$ at the abstract level. The addition of a capability maps to a $SysGrant$ operation and the removal of a capability or capabilities maps to $SysRemove$ or $SysRevoke$, respectively. Finally, reading and writing information at the concrete level maps to $SysRead$ and $SysWrite$ abstract operations.

Some of the concrete system calls (due to performance reasons) perform a combination of these main operations. Such system calls are modelled as a sequence of abstract operations.

Most operations of the concrete kernel have no effect on the authority distribution. Such operations have no visible effect on the protection state. At the abstract level, these operations are therefore, modelled either by $SysRead$ or $SysWrite$; in the case where the concrete operation modifies the user-visible data, or with $SysNoOP$; in all other cases. Given below is a detailed discussion of the summary presented in Table 5.3.

Discussion

The retype operation is the mechanism through which kernel objects are allocated. For performance reasons, the concrete retype operation supports the allocation of a number of objects (of the same type) in one system call. Thus, it maps to a sequence of $SysCreate$ operations in the protection model where sequence length is determined by the number of concrete objects created.

The *ThreadControl* operation provides an interface for controlling the properties of a thread. It facilitates a number of sub-operations, which are selected based on the system call arguments. These sub-operations include modifying thread parameters (like priority) and copying capabilities into the TCB for associating the thread with other kernel objects. In the abstract, the former operations are mapped to $SysNoOP$ and the latter to $SysGrant$. The *Exchange Registers* facilitates reading or writing user context from a TCB. Thus, this operation corresponds to $SysRead$ or $SysWrite$. The final operation supported by a TCB object is yield which has no visible effect in the abstract model and therefore modelled by $SysNoOP$.

Except for GrantIPC, the two Endpoint objects aid in implementing information flow channels. Thus, their operations corresponds, mainly to $SysRead$ or $SysWrite$; in cases where communication is successful, or $SysNoOP$ in case where there is no willing receiver to receive a non-blocking message. In addition to information, GrantIPC supports propagation of capabilities given an appropriate and a willing receiver. Such IPC operations are modelled by a $SysWrite$ followed with a $SysGrant$.

CNode related operations, in most cases, are straightforward. They either directly corresponds to an operation in the protection model or have an obvious combination. The only standout is the recycle operation. Recall the behaviour of recycle — it revokes all CDT children of the given capability (which is modelled by the $SysRevoke$), and then if the capability is the last remaining reference to the object after the revocation, it re-initialises the object. This reinitialisation requires removing all capabilities within the given object which is modelled by a sequence of $SysRemove$s.

Similar to CNode, VSpace operations are straightforward but simpler than CNode ones. Installing and removing mappings corresponds to $SysGrant$ and $SysRemove$ respectively and *Remap* removes the existing mapping and installs a new one. The last operation, *initialise*, associates an address space (PageDirectory) with a seL4 ASID, which has no visibility at the abstract level.

Frame objects do not implement any system call.They are used for installing VSpace mappings—as arguments in VSpace operations.

The *InterruptController* object stores the current interrupt association of the concrete kernel. For each interrupt line, the InterruptController object stores an AsyncEndpoint capability that is invoked to deliver the interrupt. These AsyncEndpoint capabilities are stored into the InterruptController object through *Register Interrupt* and are removed via the *Unregister Interrupt* operation. At the abstract level, these two operations maps to $SysGrant$ and $SysRemove$ respectively.

The seL4 ASID table associates a VSpace with a hardware address space identifiers (of the ARM11 architecture). The operations performed on this object does not modify the capability distribution and has no visibility to user-level applications. Therefore, all the concrete operations on this object map to $SysNoOP$ at the abstract level.

5.5 Summary

The take-grant model has been extended, and analysed by many authors (example [Sny81, Bis96, SW00]). Shapiro [Sha03] applied the *diminish-take* model—another variant of take-grant, to capture the operational semantics of the *EROS* [Har85, SSF99] system. All these formalisations and proofs however, are pen-and-paper only, mostly using graph diagram notation.

While graph diagram notation is intuitive for this domain, in this work I did find that graph diagrams can often be deceptively simple, glossing over subtle side conditions that the theorem prover enables us to track precisely. Examples of such side conditions are: new nodes added to the graph cannot overlap with the existing ones, and the graph cannot have dangling arcs; the two conditions guaranteed by sanity.

The main cause of this impreciseness is the human intuition — for instance, a "new node" intuitively will not overlap with an existing one. As such the specification simply ignores this possibility. Automated theorem provers on the other hand, lack this intuition, hence the specification needs to be precise about the what a "new node" is, and prove that all "new nodes", added in any state, adhere to that specification — recall the definition of *sane* and Lemma 3. Even though it is beyond the scope of this thesis, one can evaluate the kernel implementation against such a precise specification.

In this chapter, I have developed a precise specification of the model governing the dissemination of authority and the management of kernel memory by extending the take-grant model. The specification is entirely developed using the Isabelle/HOL interactive theorem prover. In the next chapter, I analyse the formal properties of this model.

Chapter 6

Formal Analysis of the Protection Model

Having described the seL4 protection model in the previous Chapter 5, I can now explore the question of whether the model is sufficient to enforce two useful authority distribution policies and if so, what preconditions are required to ensure the enforcement, and how to bootstrap a system satisfying these preconditions.

The first policy I explore is *spatial partitioning*. I explain the details of this policy later in Section 6.1. In brief, I show how to confine the collective authority of a set of entities (called a *subsystem*) and with that confine their access to physical resources. Moreover, the proof identifies the preconditions required for enforcing subsystems and Section 6.4.1 provides a formal example of an initial resource manager that bootstraps the system satisfying the identified preconditions.

The seL4 protection model, not only facilitates reasoning about physical memory consumption, but is also expressive enough to capture direct (*de-jure*) and indirect (*de-facto*) access to information, based on the distribution of authority. In Section 6.5, I show how the model can be used to reason about information access, by investigating the feasibility of enforcing complete, bidirectional *isolation* between entities. A formal definition of the isolation policy is provided later in Section 6.5.

All formal definitions, lemmas, theorems, and examples I present in this Chapter are machined-checked using the Isabelle/HOL system.

6.1 Informal Discussion of Requirements

Ideally a partitioned system can be viewed as a distributed system, with each machine in this idealised system, communicating with another via dedicated communication links. Rushby [Rus99] defined partitioning as:

> A partitioned system should provide fault containment equivalent to an idealised system in which each partition is allocated an independent processor and associated peripherals and all inter-partition communications are carried on dedicated lines.

In such a system, no application in a partition will be capable of accessing physical resources, from another in a different partition; be it physical memory or CPU cycles. As such, partitioned applications are isolated in both the spatial and temporal domains. The former is called *spatial partitioning* and the latter *temporal partitioning*.

Temporal partitioning, while important, is beyond the scope of this thesis. Spatial partitioning, on the other hand, is central to seL4.

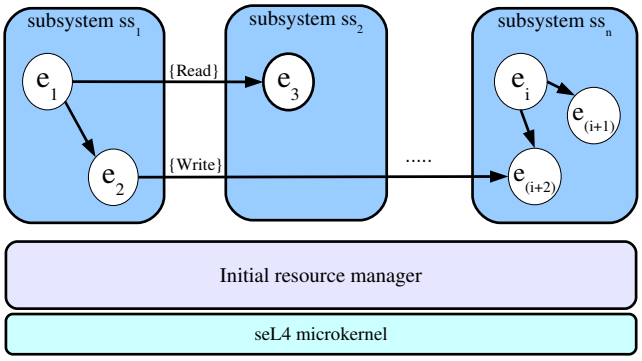


Figure 6.1: Example system configuration

According to Rushby's definition in [Rus99], spatial partitioning is concerned with software in one partition accessing the memory of another — isolation of the memory resources.

The seL4 protection model, by extending its access control to cover physical memory, provides a framework for analysing and enforcing a spatial partitioning policy over applications.

However, partitioning in the seL4 protection model, extends beyond physical memory, and therefore beyond simple spatial partitioning. Besides physical memory, the model captures the configuration of authority within a partition that allows communication. Thus, the model facilitates reasoning about the communication channels that a set of partitioned applications may acquire in a future state.

To make the distinction, I use the term *subsystems*. In an idealised system, each subsystem is allocated independent memory resources and all inter-subsystem communications are carried on dedicated lines.

As an illustrative, concrete, example, consider the system configuration in Figure 6.1. In this system there are *n* distinct subsystems, namely $ss_1, ss_2 \ldots ss_n$. Each subsystem is a collection of one or more applications or processes, or at an abstract level, a set of entities. The initial resource manager responsible for instantiating subsystems needs to guarantee that any given subsystem, say ss_i, cannot exceed the amount of physical memory given to it. Nor should ss_i be capable of *directly* communicating with another subsystem unless the resource manager has explicitly arranged for it.

6.1.1 Capability Based Protection

The initial resource manager (see Figure 6.1), created by the system soon after booting, is responsible for creating and disseminating the authority for each subsystem. Once the subsystems are setup, the initial resource manager can exits, leaving the subsystems which are now bounded by the authority they received from the initial resource manager and the protection model.

Recall the protection mechanism of seL4 — all memory allocation is explicit by user request via the invocation of an Untyped capability. This means that the amount of physical memory that an entity (or a process) can use is strictly controlled by the untyped capabilities in its possession. Similarly, all communication channels are named and accessed via capabilities.

So, if a subsystem ss_i can violate its boundary, then there should be at least one entity in ss_i that has received a capability from an entity in a different subsystem. In other words, subsystem boundaries are violated only if capabilities are allowed to flow from one subsystem to another. Thus, enforcing subsystems boils down to isolation of authority.

Given this scenario, the question is: Under which conditions can some entity, say e_x, *leak* a capability to some other entity e_i in the current state or at any point in the future? And, importantly, can such a future leak be foreseen and prevented?

Once the subsystems are created, they will execute system calls, and thereby modify the protection state. In order to make strong guarantees we need to prove that there is no sequence of commands with which to arrive at a future system state in which e_x can leak more access to e_i. Moreover, the interest here is in *mandatory* isolation of authority, that is, in showing that e_x *cannot* leak a capability to e_i, rather than that e_x *can* but does not.

A subsystem is then the set of entities such that all the entities that possess the ability to leak to a subsystem entity, in the current state or in some future state, is within the subsystem itself.

Unlike subsystems, analysing the access to information has another dimension — proxy or de-facto access. As an example, consider the configuration shown in Figure 6.1. If subsystems are enforced, then subsystem ss_2 will not come into possession of a capability to directly write (de-jure write) to subsystem ss_n; if it does it will be a violation of the subsystem policy. However, subsystem ss_2 can still indirectly write information to subsystem ss_n through subsystem ss_1. For this to happen, subsystem ss_1 reads information from ss_2 (using its read authority) and then writes that to ss_n (using its write authority), causing a de-facto write from ss_2 to ss_n via ss_1.

To analyse the direct and indirect access to information, I use another policy: bidirectional, mandatory, *isolation* of entities. I leave the formal definition of isolation till Section 6.5. In brief, a set of entities are isolated if all the entities that possess the ability to either directly or indirectly access information from an entity in the isolated set, in the current state or in some future state, is within that set itself. Note the similarity between subsystems and isolation. The two analysis are similar, but performed for different access rights. As such, one should be able derive one analysis from the other.

With this informal introduction I now turn to analysing the seL4's protection model. I show that it is feasible to enforce subsystems and isolate entities. The structure of the formal analysis is as follows: first in Section 6.2, I introduce some formal predicates related to the informal discussion above. Section 6.3 shows that it is feasible to decide if a leak can take place in any future state and identifies what restrictions should be in place to prevent such a leak. Then, these results are extended in Section 6.4 to show that it is feasible to enforce subsystems and provides an example of a resource manager enforcing subsystems (see Section 6.4.1). Section 6.5 shows how the model can be used to reason about direct and indirect access to information.

6.2 Predicates

Before proceeding to the analysis, a precise statement of what leak means is warranted. Suppose we have two entities, $\mathtt{e}_x$ and $\mathtt{e}_y$, in some state $\mathtt{s}$. Suppose $\mathtt{e}_x$ is authorised to add a capability to $\mathtt{e}_y$. Then we say $\mathtt{e}_x$ can leak to $\mathtt{e}_y$ in state $\mathtt{s}$.

In our model, there are two operations that add a capability to a system entity — `SysCreate` and `SysGrant`. These operations are `legal` only if the entity initiating the

operation has a capability that points to the entity under consideration and has (at least) *Grant* right (see definition of *legal* in Section 5.3.3). Such a capability I call a *grantCap*:

```
grantCap :: entity_id ⇒ cap
grantCap e ≡ (|entity = e, rights = {Grant}|)
```

Note that *grantCap e* is the capability that contains the minimum (least) authority that allows an addition of a capability to entity *e*. This notion of least authority is captured by the infix operator *:<*. It indicates that a set of capabilities (*C*) has at least as much authority as a given capability (*c*):

```
c :< C ≡ ∃c'∈C. entity c = entity c' ∧ rights c ⊆ rights c'
```

If there is a capability in the given set such that it points to the same object and has equal or more authority, then *:<* returns *true*, and *false* otherwise.

Now I can define the predicate *leak*. I write $s \vdash e_x \rightarrow e_y$ to indicate that in state *s*, entity e_x has the ability to add a capability to entity e_y. The definition of *leak* is as follows:

```
leak :: state ⇒ entity_id ⇒ entity_id ⇒ bool
s ⊢ e_x → e_y ≡ grantCap e_y :< caps_of s e_x
```

The rationale behind the *leak* predicate is the following observation:

Lemma 4. *Suppose there exists some entity z in state s. Suppose, none of the entities in subsystem* ss_1 *has a capability that points to z in state s. Suppose after executing a command on s, some entity in* ss_1 *gets a capability that points to z. Then, there exists an entity that can leak to an entity in* ss_1 *and already has a capability that points to z in state s.*

$$\begin{aligned}&[\![\mathit{isEntityOf}\ s\ z;\ \forall c \in \bigcup_{e \in ss_1} \mathit{caps\_of}\ s\ e.\ \mathit{entity}\ c \neq z;\\ &\ \exists c \in \bigcup_{e \in ss_1} \mathit{caps\_of}\ (\mathit{step}\ \mathit{cmd}\ s)\ e.\ \mathit{entity}\ c = z]\!]\\ &\Longrightarrow \exists e_x\ e_y\ c'.\ c' \in \mathit{caps\_of}\ s\ e_x \wedge \mathit{entity}\ c' = z \wedge e_y \in ss_1 \wedge s \vdash e_x \rightarrow e_y\end{aligned}$$

Proof. By unfolding the definitions and observing that only the grant operation can leak a capability to an existing entity. □

As an illustrative example consider the system in example Figure 6.1. Just for explanation let us assume that the number of entities in a system does not grow. The two entities in subsystem ss_1 have no capability that points to e_i. Suppose, after executing a command one of these entities gets a capability that points to e_i. This is at odds with the enforcement of subsystem. For this to happen, from Lemma 4, we see that there should exists some entity (say e_x) that has a capability to e_i and has the ability to leak to an entity in ss_1 in the current state. Moreover, e_x cannot be one of the two entities in ss_1, since they did not have a capability that points to e_i in state *s*. So to enforce subsystem policy, what we need is a mechanism to prevent entities outside a subsystem from leaking capabilities into entities within a subsystem.

Similar to *SysGrant*, *SysCreate* also adds a capability to an entity. However, unlike *SysGrant*, the capability added by the *SysCreate* will not point to any of the existing entities (such as e_i in Figure 6.1).

Preventing a leak in the initial state (s_0) is trivial — the initial resource manager creates this state, and therefore the capabilities possessed by any entity are directly under its control. More interesting are leaks that might occur in some later state:

$$leakInFuture :: state \Rightarrow entity\_id \Rightarrow entity\_id \Rightarrow bool$$
$$\Diamond\ s \vdash e_x \rightarrow e_y \equiv \exists cmds.\ execute\ cmds\ s \vdash e_x \rightarrow e_y$$

That means, there is a sequence of commands, that if executed will result in a state in which entity e_x can leak to e_y.

The purpose of the analysis in the next section is twofold. Firstly, I show that given the initial state s_0, it is feasible to decide whether $\Diamond\ s_0 \vdash e_x \rightarrow e_i$ is true. Secondly, identify a restriction R that the resource manager can enforce on s_0 such that $R\ s \Longrightarrow \neg\ \Diamond\ s_0 \vdash e_x \rightarrow e_i$.

6.3 Necessary Conditions for Propagation of Authority

This part of the analysis identifies the necessary conditions that must be satisfied in the initial state for one entity to leak a capability to another in some future state. To be precise, I show that given a state s, and two entities e_x and e_y in s, we can give a tight and safe approximation of the value of the predicate $\Diamond\ s \vdash e_x \rightarrow e_y$. I show formally that the approximation is safe and argue informally that it is tight (i.e. close enough).

In fact, the literature usually does not call this predicate an approximation, but just makes stronger assumptions on the system such that the approximation is precise. Here I leave the model itself more general, and recognise that the decision procedure is, indeed a conservative approximation only.

Using this approximation, I show that if the initial resource manager adheres to certain restrictions on the initial dissemination of capabilities, a leak between two entities can be prevented in any future derived state.

In the analysis, the ability to create entities is not excluded. Moreover, I do not make any assumptions about the ordering of create operations. Contrary to some of the pen-and-paper proofs in the literature [San88, AS91, Min84, San92a], here I directly show the property for any sequence of commands, including ones that add new entities to the state and without any assumptions on ordering of commands in the sequence.

The only restriction is that the entities e_x and e_y already exist in s; otherwise the statement itself does not make sense in state s. I will discuss later why this does not constitute any loss of generality.

Recall the initial state s_i created by the kernel for the initial resource manager (see Section 5.3.4). Starting from this state, the resource manager executes some sequence of operations to bootstrap the rest of the system. For our discussion here, the exact sequence of commands the initial resource manager executes to bootstrap is irrelevant. It is sufficient to note that this sequence can only contain operations defined by the protection model. Moreover, the same is true for any subsequently created entity.

I have already proved that sanity is invariant (see Section 5.3.5). Thus, I only need to consider *sane* states for the proof.

The structure of the proof is as follows. First, I introduce a property, related to *leak*, and that is preserved by *step*. Such a property, as we shall see later naturally lends itself to identifying the restriction that will prevent a leak in the future.

The main invariant property of the system relating to dissemination of authority is the symmetric, reflexive, transitive closure over the *leak* relation — the arcs in the capability graph with *Grant* right (see Figure 5.2). The symmetric closure of *leak* is called *connected* and I write $s \vdash e_x \leftrightarrow e_y$ to denote e_x and e_y are *connected* in state s:

$$s \vdash e_x \leftrightarrow e_y = s \vdash e_x \rightarrow e_y \vee s \vdash e_y \rightarrow e_x$$

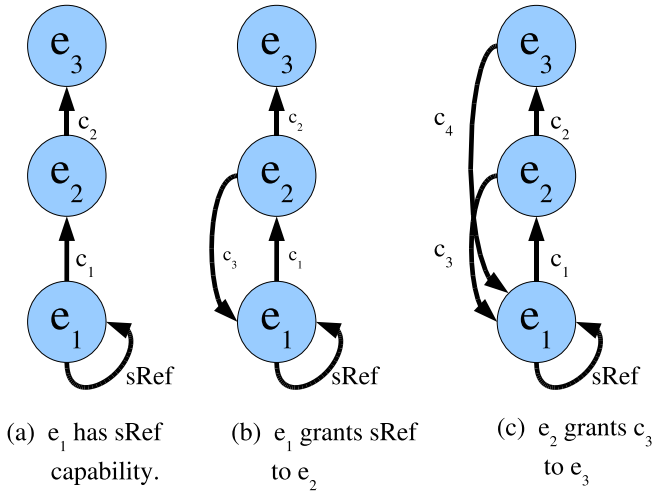


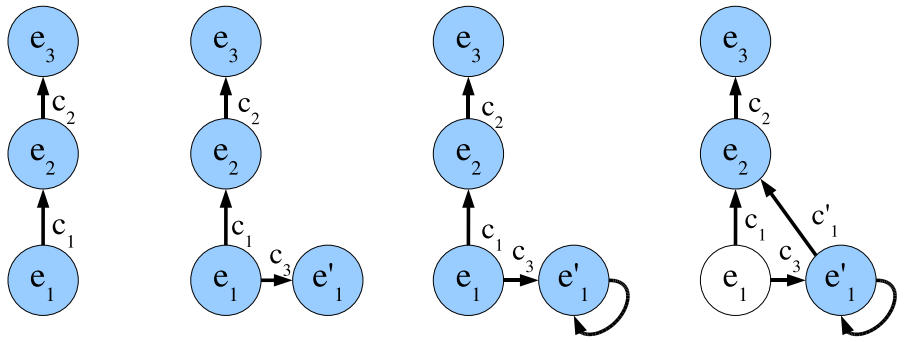
Figure 6.2: The effect of self-referencing capabilities

The intuition behind this invariant is the following. We are looking at grant capabilities only, because these are the only ones that can disseminate authority. We need the transitive closure, because we are looking at an arbitrary number of execution steps. We need the symmetric closure, because as soon as there is one entity in the transitive closure that has a grant capability to itself or has the ability to create, it can use its authority to reverse the direction of potential authority flow (or *invert*) in all or part of the grant arcs it is transitively connected to. Given the transitive and symmetric part, the reflexiveness follows. I further explain the need for symmetric and reflexive closures below.

As an illustrative example of inverting the grant arcs, consider the three entities in Figure 6.2. Shown here are only the grant capabilities; capabilities that contain at least the `Grant` authority. Initially, entity e_1 has a grant capability, `sRef` to itself (see part (a)). Part (b) of the figure shows the capability configuration when e_1 grants a copy of `sRef` to e_2. As it can bee seen from the diagram, in this state the initial arc from e_1 to e_2; arc c_1, is reversed by the capability c_3. The system state after e_2 granting a copy of c_3 to e_3 is shown in part (c) of the figure. Note how, all arcs in part (a), are reversed, either directly, or transitively. Say, for instance, the initial arc from e_2 to e_3 is reversed transitively; from e_3 we follow c_4 and arrive at e_1 and from there take arc c_1 to arrive at e_2. As for the reflexiveness, starting from any entity we can find path back to it (see part (c) of the figure).

Suppose e_1 did not have a grant capability to itself, but possesses the authority to create. In this case, as shown in Figure 6.3, e_1 can use this authority and create a system state that resembles that of part (a) of Figure 6.2; first, by exercising its authority to allocate, e_1 creates a new entity e'_1 (see part (b) of Figure 6.3), and in the process gets a grant capability, capability c_3 to the new entity. Then in steps (c) and (d), by using c_3, e_1 grants copies of c_3 and c_1 to entity e'_1 respectively. Note the similarity between the state (d) in Figure 6.3 and state (a) of Figure 6.2. Thus, using the same reasoning one can see that all arcs between e'_1, e_2 and e_3 can be inverted.

The ability to invert grant arcs in the presence of the `Grant` operation is well studied in the literature [LM82, Min84, Bis84] and two broad classes of solutions are proposed to make the authority flow unidirectional: (a) remove the `Grant` operation and authorise propagations from the receiver side (a variant of this is the *diminished-take* model [Sha99]), and (b) remove self-referencing capabilities — all applications have a nontransferable,



(a) Initial state. (b) e_1 creates e'_1 (c) e_1 grants c_3 to e'_1 (d) e_1 grants c_1 to e'_1

Figure 6.3: The effect of creating new entities

intrinsic authority to modify its address space. Removing the `Grant` operation means the sender loses its ability to decide what authority to propagate.

However, adopting schemes such as diminished-take, where it is possible to enforce filters by controlling the authority in the receiver side complicates the protection model. Moreover, adding an intrinsic right to manage the address space makes resource management hard. As such, these techniques were not considered in seL4.

The two assumptions: symmetry and reflexivity, is what makes the analysis an approximation. Without these, as explained above, the closure is not invariant over the grant and create operations. With symmetry and reflexivity, we might claim that a given entity can gain more authority in the future than what it in fact can: a) if there is no self-referential grant capability in the system, and b) if there is no create authority in the system. Although it is possible to build such systems in seL4, and for small, static systems this might even occur in practise, these are very simple to analyse and it is unlikely that the approximation will lead to undue false alarms in the general case. For the vast majority — those with the ability to create and to grant to themselves — the invariant and therefore the prediction is precise.

Recall the definition of `leak`: it is computed based on the authority one entity possesses over the other. Given our states are `sane`, we see that only existing entities can be `connected`.

Lemma 5. *Connected implies existence:*
$[\![\texttt{sane s; s} \vdash e_x \leftrightarrow e_y]\!] \Longrightarrow \texttt{isEntityOf s } e_x \wedge \texttt{isEntityOf s } e_y$

Proof. By unfolding the definitions, and observing from `leak` that one entity must possess a capability that points to the other. Thus, given the state is `sane`, both must be existing entities in that state. □

Next I analyse how each of the operations affects the `connected` relation. In particular, the analysis focuses on properties of the form, ”if two entities become connected after execution of an operation, what was their relation at the starting state”.

The `SysNoOP` operation, as the name implies does nothing. Similarly, `SysRead` and `SysWrite` mutate data and have no effect on the capability distribution. Therefore, it is trivial to conclude that these three operations will not effect the `connected` relation. Thus, the following lemmas:

Lemma 6. *Connected is invariant over NoOP:*
`step (SysNoOP e) s` $\vdash e_x \leftrightarrow e_y \implies$ `s` $\vdash e_x \leftrightarrow e_y$

Lemma 7. *Connected is invariant over Read:*
`step (SysRead e c) s` $\vdash e_x \leftrightarrow e_y \implies$ `s` $\vdash e_x \leftrightarrow e_y$

Lemma 8. *Connected is invariant over Write:*
`step (SysWrite e c) s` $\vdash e_x \leftrightarrow e_y \implies$ `s` $\vdash e_x \leftrightarrow e_y$

Proof. By unfolding the definition of `step`. □

The remaining four operations — `SysCreate`, `SysGrant`, `SysRemove` and `SysRevoke`, modify the capability distribution, and therefore have the potential to modify the predicate. I now examine their behaviour in detail.

Out of these operations, `SysRemove` and `SysRevoke`, remove capabilities: from a single entity in the case of `SysRemove` or from a set of entities in the case of `SysRevoke`. By recalling that `leak` and therefore `connected` checks for the existence of a particular capability, we see that neither operation has the potential to connect two entities that are disconnected. This leads to the following two lemmas.

Lemma 9. *Connected is invariant over Remove:*
`step (SysRemove e` c_1 c_2`) s` $\vdash e_x \leftrightarrow e_y \implies$ `s` $\vdash e_x \leftrightarrow e_y$

Proof. By unfolding the semantics of remove and observing that they can not add capabilities to the state. □

Lemma 10. *Connected is invariant over Revoke:*
`step (SysRevoke e c) s` $\vdash e_x \leftrightarrow e_y \implies$ `s` $\vdash e_x \leftrightarrow e_y$

Proof. By induction over the list of capabilities to remove and Lemma 9. □

The `SysGrant` operation on the other hand, does have the ability to connect two entities that previously had not been connected. However, only under restricted conditions: the grant operation can connect two entities only if they were transitively connected in the state before.

Lemma 11. *Grant preserves the transitive, reflexive closure of connections* [1]*:*
`step (SysGrant e` c_1 c_2 `R) s` $\vdash e_x \leftrightarrow e_y \implies$ `s` $\vdash e_x \leftrightarrow^* e_y$

Proof. Suppose `s` $\vdash e_x \leftrightarrow e_y$, then by definition of the transitive closure, the lemma is true. Thus, the case we need to consider is when $\neg$ $s_i \vdash e_x \leftrightarrow e_y$, but `step (SysGrant e` c_1 c_2 `R) s` $\vdash e_x \leftrightarrow e_y$. For this to happen, either e_x or e_y, in the derived state, must possess a `grantCap` to the other. Let this entity be `x` and the other `y`. From the definition of `SysGrant`, we see that `entity` c_1 `= y`, `entity` c_2 `= x` and `Grant` $\in$ `rights` c_2. Moreover, from `legal` we see {c_1, c_2} $\subseteq$ `caps_of s e` and `Grant` $\in$ `rights` c_1. Thus, by definition and symmetry of `connected`, we have `s` $\vdash$ `x` $\leftrightarrow$ `e` and `s` $\vdash$ `e` $\leftrightarrow$ `y`. Given these facts, from the definition of transitive and reflexive closure, we can conclude the lemma. □

Unlike the operations we considered thus far, `SysCreate` introduces a complication in that it introduces new entities in to the system. However, if we consider existing entities and make use of the `sane` state property, we see that:

[1] Note that, in standard logic the superscript "*" denotes the transitive and reflexive closure

Lemma 12. *Create preserves connected on existing entities:*
$[\![\texttt{isEntityOf}\ s\ e_x;\ \texttt{isEntityOf}\ s\ e_y;\ \texttt{step}\ (\texttt{SysCreate}\ e\ c_1\ c_2)\ s \vdash e_x \leftrightarrow e_y]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow e_y$

Proof. By unfolding the definition of create operations and making use of the fact that there can be no dangling references that might point to the new entity (given the state is `sane`), and that the new entity does not overlap with any of the existing ones. □

From these lemmas we see that if two existing entities become connected after executing a single command, then they should either be connected in the previous state, in the case of `SysNoOP, SysRead, SysWrite, SysRemove, SysRevoke` and `SysCreate`, or connected transitively, in the case of `SysGrant`. By combining the previous lemmas we have:

Lemma 13. *Connected after a single command:*
$[\![\texttt{isEntityOf}\ s\ e_x;\ \texttt{isEntityOf}\ s\ e_y;\ \texttt{step}\ cmd\ s \vdash e_x \leftrightarrow e_y]\!] \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Proof. By case distinction on the command, and using the appropriate lemma from Lemma 6 to Lemma 12. □

The plan for the rest of the proof is as follows. First lift Lemma 13 to the reflexive, transitive closure, such that $\texttt{step}\ cmd\ s \vdash e_x \leftrightarrow^* e_y \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$, for any two existing entities e_x and e_y. Then, by induction over the command sequence we prove that $\texttt{execute}\ cmds\ s \vdash e_x \leftrightarrow^* e_y \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$.

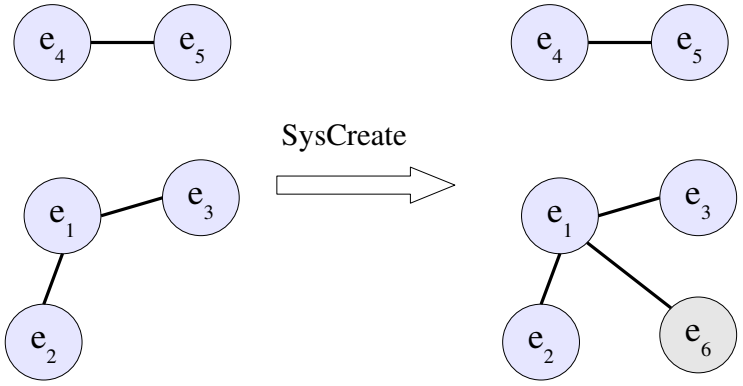
The motivation behind proving the above lemma is the following observation: Suppose e_x can leak to e_y in the derived state $\texttt{execute}\ cmds\ s$. Then by definition of `connected`, $\texttt{execute}\ cmds\ s \vdash e_x \leftrightarrow e_y$, and hence from the definition of transitive closure $\texttt{execute}\ cmds\ s \vdash e_x \leftrightarrow^* e_y$. Then from the above lemma, we can concluded $s \vdash e_x \leftrightarrow^* e_y$. In other words, if e_x can leak to e_y in some future state derived from s the e_x and e_y must be transitively connected in the starting state s.

It turns out, that the first step — lifting Lemma 13 to the reflexive, transitive closure, is the most interesting one. This proof was done by induction over the reflexive, transitive closure. Although, we are considering the `connected` relationship between existing entities, the proof obligation in the induction step is more general — it requires us to consider entities that might have being introduced by the current command. Recall that there is only one command that introduces a new entity — `SysCreate`. It turns out that Lemma 12 is not strong enough to get through the induction step, because it requires both entities to exist in the pre-state.

Hence, I break the proof into two parts: I treat `SysCreate` separately from the other commands. I call the commands that do not introduce new entities *transporters*. The lemma I proved for transporters is as follows:

Lemma 14. *Transport commands preserve* $\texttt{connected}^*$ *in sane states:*
$[\![\texttt{step}\ cmd\ s \vdash e_x \leftrightarrow^* e_y;\ \texttt{sane}\ s;\ \forall e\ c_1\ c_2.\ cmd \neq \texttt{SysCreate}\ e\ c_1\ c_2]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Proof. Firstly we see that the derived state is `sane`, this follows from Lemma 2. Next, we induct over the transitive and reflexive closure. If two entities, x and y, are `connected` in the derived state, from Lemma 5 it follows both x and y exists in that state. Given that transporters do not add new entities, x and y are entities in s, hence from Lemma 13 we can conclude the lemma. □



a) Initial state b) After Create

Figure 6.4: The effect of SysCreate on connected

The complication that arises with *SysCreate* is the newly introduced entity. The entities considered in the induction step may include this newly created entity. To get through the induction step, one needs a stronger lemma which answers the following question: Can two entities, let us call them x and y, become transitively connected through the newly introduced entity, and if so what is the relationship between x and y in the previous state?

The intuition behind the proof is the following. Recall *SysCreate* — the newly added entity brings with it no authority. If we assume a graph based representation of the system state as shown in Figure 6.4, where nodes represent entities and arcs represent capabilities with *Grant* rights, then there are no arcs going out of the new entity (e_6 in the Figure). Moreover, if the state is *sane*, then there is only a single arc connecting the new node to the rest of the nodes. Together these two properties guarantee that, if two entities say x and y, are connected transitively in the post-state through the new entity, then x and y should have been transitively connected in the pre-state through the entity which is holding the capability to the new entity in the post-state. As an illustrative example, in part b) of Figure 6.4 nodes e_2 and e_3 are transitively connected to one another through the new node e_6 only because they were transitively connected to one another through e_1, in the initial state.

Note that the lack of dangling references and creating non-overlapping objects; in other words having a *sane* state, is central to this proof. In part b) of the figure, suppose e_6 overlaps with the e_4 or if e_4 has an dangling arc which happens to point to e_6. Then in the post-state, all entities becomes transitively connected to one another, which violates the property I just stated.
The properties I just mentioned are captured in the following lemma:

Lemma 15. *Given two entities* e_x *and* e_z *are transitively connected in the state after* `SysCreate e` c_1 c_2, *given that* e_x *exists in the pre-state* `s`, *and given that* `sane s`, *it is true that,* `s` $\vdash$ e_x $\leftrightarrow^*$ `e` *(where* `e` *is the entity that performed the create operation) if* e_z *is the entity just created or* `s` $\vdash$ e_x $\leftrightarrow^*$ e_z *otherwise. In Isabelle:*
$[\![$`step (SysCreate e` c_1 c_2`) s` $\vdash$ e_x $\leftrightarrow^*$ e_z`; isEntityOf s` e_x`; sane s`$]\!]$
$\Longrightarrow$ `if` e_z `= next_id s then s` $\vdash$ e_x $\leftrightarrow^*$ `e else s` $\vdash$ e_x $\leftrightarrow^*$ e_z

Proof. We note that the derived state is *sane*, which follows from Lemma 2. Then we

induct over the transitive and reflexive closure. The base case is trivial. The induction step has the following form: given $\mathit{isEntityOf}\ s\ e_x$, $\mathit{sane}\ s$, and $\mathit{step}\ (\mathit{SysCreate}\ e\ c_1\ c_2)\ s \vdash e_x \leftrightarrow^* e_y$, $\mathit{step}\ (\mathit{SysCreate}\ e\ c_1\ c_2)\ s \vdash e_y \leftrightarrow e_z$, and the induction hypothesis $\mathit{if}\ e_y = \mathit{next\_id}\ s\ \mathit{then}\ s \vdash e_x \leftrightarrow^* e\ \mathit{else}\ s \vdash e_x \leftrightarrow^* e_y$, we show $e_z = \mathit{next\_id}\ s \longrightarrow s \vdash e_x \leftrightarrow^* e$ and $e_z \neq \mathit{next\_id}\ s \longrightarrow s \vdash e_x \leftrightarrow^* e_z$. We make a case distinction on whether e_y and e_z are existing or newly introduced entities.

If both are existing entities we conclude using Lemma 13. If both are newly added, we have $s \vdash e_x \leftrightarrow^* e$ by assumption.

If e_y is newly created, and e_z is existing, we know from the induction hypothesis that $s \vdash e_x \leftrightarrow^* e$ and need to show $s \vdash e_x \leftrightarrow^* e_z$. This is true, since from the assumption $\mathit{step}\ (\mathit{SysCreate}\ e\ c_1\ c_2)\ s \vdash e_y \leftrightarrow e_z$, the definition of legal, and $\mathit{sane}\ s$, we have that $s \vdash e \leftrightarrow e_z$ and therefore with transitivity what we had to show.

The remaining case is the dual. We know that e_y is an existing entity, and hence $s \vdash e_x \leftrightarrow^* e_y$. We also know that e_z is new and therefore need to show $s \vdash e_x \leftrightarrow^* e$. This reduces to showing $s \vdash e_y \leftrightarrow e$ which again follows from the assumption $\mathit{step}\ (\mathit{SysCreate}\ e\ c_1\ c_2)\ s \vdash e_y \leftrightarrow e_z$, e_z being new, the definition of legal, and $\mathit{sane}\ s$. This concludes the proof of Lemma 15. □

By combining Lemma 14 and Lemma 15 we get the following lemma:

Lemma 16. *Single execution steps preserve* $\mathtt{connected}^*$ *for existing entities in sane states:*
$[\![\mathit{step}\ \mathit{cmd}\ s \vdash e_x \leftrightarrow^* e_z;\ \mathit{isEntityOf}\ s\ e_x;\ \mathit{isEntityOf}\ s\ e_z;\ \mathit{sane}\ s]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_z$

Proof. By case distinction on transporter commands and create, and using Lemma 14 and Lemma 15 to prove each case respectively. □

The rest of the proof is relatively easy. By induction over a sequence of commands, one can immediately conclude:

Lemma 17. *Execution preserves* $\mathtt{connected}^*$ *for all entities existing in any sane initial state:*
$[\![\mathit{sane}\ s;\ \mathit{isEntityOf}\ s\ e_x;\ \mathit{isEntityOf}\ s\ e_y;\ \mathit{execute}\ \mathit{cmds}\ s \vdash e_x \leftrightarrow^* e_y]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Proof. By induction on the list of commands and Lemma 16 □

Together with $s \vdash e_x \rightarrow e_i \Longrightarrow s \vdash e_x \leftrightarrow^* e_i$, we conclude our theorem on how to decide future leaks:

Lemma 18. *In a sane state, if one existing entity can leak authority to another entity at any time in the future, then they are connected now:*
$[\![\mathit{sane}\ s;\ \mathit{isEntityOf}\ s\ e_x;\ \mathit{isEntityOf}\ s\ e_y;\ \Diamond\ s \vdash e_x \rightarrow e_y]\!] \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Proof. By Lemma 17 and the definition of $\Diamond\ s \vdash e_x \rightarrow e_y$. □

The decidability (or approximation thereof) of this model is more naturally phrased like in the literature as the contra-positive of Lemma 18. It clearly identifies the restriction that will prevent a leak from e_x to e_y in any future state:

Theorem 1. *In any sane state, if two existing entity are not connected, they will never be able to leak authority to each other.*
$[\![\mathit{sane}\ s;\ \mathit{isEntityOf}\ s\ e_x;\ \mathit{isEntityOf}\ s\ e_y;\ \neg\ s \vdash e_x \leftrightarrow^* e_y]\!]$
$\Longrightarrow \neg\ \Diamond\ s \vdash e_x \rightarrow e_y$

Proof. Contra-positive of Lemma 18. □

There are two noteworthy aspects of the above theorem. First, it shows that it is possible to decide if some entity e_x can leak authority to another entity e_y, in any future state derived from s, by looking at the capability distribution of s. By computing the symmetric, reflexive and transitive closure over `leak` on state s — for which there are number of well knows efficient algorithms (for example [Nuu95]) — the initial resource manager, who has the full control and the knowledge over the initial distribution of grant rights, can predict capability leakages that might happen in future states.

Second, the theorem identifies the restriction that needs to be enforced on state s to prevent a capability leak. If the initial resource manager distributes capabilities in such a manner that the two entities, e_x and e_y, are not in the symmetric, reflexive and transitive closure over `leak` in state s, then the protection mechanism guarantees that e_x will never be able to leak a capability to e_y, in any future state derived from s. How this restriction is enforced in a practical situation is discussed latter in Section 6.4.1.

The assumptions of Theorem 1 are that the state s is sane and that both entities exist in s. Sanity is not an issue. I have already shown that it is a global system invariant. The restriction to existing entities might be a reason for concern. Formally, we can make no useful statement in s over entities that do not exist yet. However, intuitively, we would like the non-leakage property to be true as well for all entities that do not exist yet. The theorem implies that e_x can not leak authority to e_y via any of these new entities (create operations were not excluded in the proof), but what about a new entity e, created later, leaking new authority to e_y? The theorem does not make a statement about this (because the precondition `isEntityOf s e` is false).

However, we can run the system up until e is freshly created. In this state e exists, has no authority yet, the state is sane, and e_y still exists. The theorem is then applicable and says that e will not be able to leak to e_y if e_y is not transitively connected to e at that point.

The classic non-leakage property does not fully express our intuition about the subsystem — we want to show that e_x cannot acquire more authority, even from an entity that is not in existence yet, but might come into existence later (possibly in another subsystem).

The next section generalises the above theorem to make a more intuitive and direct statement about authority distribution and show how subsystems can be enforced by using the model.

6.4 Enforcing Subsystems

In this section, I generalise the non-leakage result to show that it is feasible to implement subsystems using the seL4 protection model. As I mentioned earlier, a subsystem is merely a set of entities such that none of the entities in the subsystem will gain access to a capability to an entity of another subsystem if that authority is not already present in the subsystem. For instance, if there is no `Write` authority over a particular entity e_x within a subsystem at the start, then none of the entities in that subsystem will ever gain write authority over e_x, in any run of the system. Thus, guaranteeing that isolated subsystems will remain isolated in all possible future states. If the authority is already present, then I show that it cannot be increased.

Subsystems can grow over time (given they have the authority to do so) and therefore, unlike the leakage proof in Section 6.3, the statement also includes the effects of entities that currently do not exist yet.

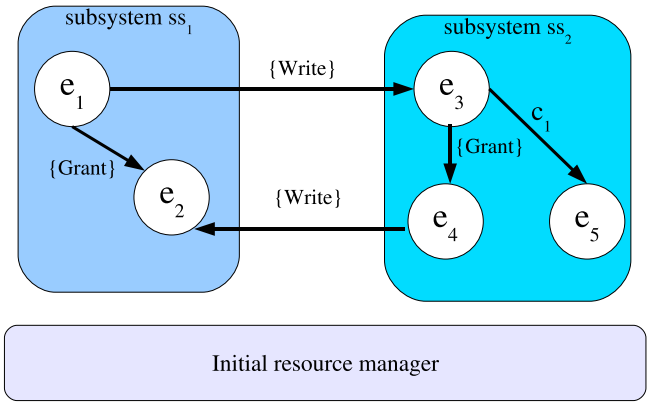


Figure 6.5: Example System Configuration

Formally, a subsystem is identified by any of its entities e_s and it is defined as the set of entities in the symmetric, reflexive, transitive closure of grant arcs to e_s, or in short $connected^*$:

$$subSys :: state \Rightarrow entity\_id \Rightarrow entity\_id\ set$$
$$subSys\ s\ e_s \equiv \{e_i.\ s \vdash e_i \leftrightarrow^* e_s\}$$

As earlier, I start with an illustrative, informal example. Figure 6.5 shows the configuration of a system with two subsystems: subsystem ss_1 and ss_2. The state shown here is the initial configuration (or s_0) just after bootstrapping. One can obtain the entities in a subsystem, using the $subSys$ function, specifying the current system state and one of the entities in that subsystem. For instance, the entities of subsystem ss_1 in s_0 are given by $subSys\ s_0\ e_1 = \{e_1, e_2\}$.

Note that, in state s_0 shown in Figure 6.5, neither of the two entities e_1 and e_2 has a capability to the entity e_5. For subsystem boundaries to hold, I need to show that in no future system state any of the entities in ss_1 — which might grow/shrink depending on create/remove/revoke operations — will have a capability to e_5. Thus, making the two subsystems strongly isolated spatially.

The above property alone is sufficient for isolation of authority — it shows that if a subsystem does not possess any authority over an entity, then the subsystem will never be able to acquire any authority over that entity. However, it is somewhat limiting; because it does not allow us to reason about existing authority. For instance, using the above property, we can not reason about the behaviour of the two communication channels that exists between e_1 and e_3, and e_4 and e_2 respectively.

To make it possible to reason about both the existing and non-existing authority of a subsystem, I introduce a different formulation: the collective authority possessed within a subsystem, over an entity in a different subsystem can not increase. To formally phrase this statement, I need two more concepts — the $subSysCaps$ function and the $dominate$ (infix $:>$) operator.

$$subSysCaps :: state \Rightarrow entity\_id \Rightarrow cap\ set$$
$$subSysCaps\ s\ x \equiv \bigcup caps\_of\ s\ `\ subSys\ s\ x$$

$$c :> CS \equiv \forall c' \in CS.\ entity\ c' = entity\ c \longrightarrow rights\ c' \subseteq rights\ c$$

The $subSysCaps$ function initially finds the set of entities in the subsystem by using $subSys$,

and then returns the union of all capabilities possessed by the entities in that subsystem. Note that symbol ‘ stands for mapping a function to a set.
A capability c dominates a capability set CS, (x :> CS) if CS provides at most as much authority as capability c over the entity c points to. That means, for example, if c only possesses a $Grant$ right to some entity e, then no capability in CS will provide more than a $Grant$ right to e. For instance, in Figure 6.5, using the above notation, I write

$noCap\ e_5$:> $subSysCaps\ s_0\ e_1$

with $noCap\ e \equiv (\!|entity = e, rights = \{\}|\!)$ to indicate that there is no e_5 capability anywhere in subsystem ss_1. Similarly, for e_3 I write $wCap\ e_3$:> $subSysCaps\ s_1\ e_1$, where $wCap\ e \equiv (\!|entity = e, rights = \{Write\}|\!)$. That means, the maximum authority within subsystem ss_1 over the entity e_3 is $\{Write\}$.
For subsystem boundaries to hold I need to show that:

$$\forall cmds.\ c :> subSysCaps\ (execute\ cmds\ s_0)\ e_1$$

where c is some capability to an entity in s_0. For example c can be $noCap\ e_5$, if we are considering the flow of authority over entity e_5, or $wCap\ e_3$ if the interest is on the communication channel to e_3.

Below, I show such mandatory subsystem boundaries are enforceable. Following that, in Section 6.4.1, I provide a formal example of subsystems together with a description of how a resource manager bootstraps them.

For this proof, I have two main assumptions. First, I assume that the entity we are interested in gaining authority to exists in the state we use for reasoning. In the examples I provided above, note that entity e_5 and e_3 are already in existence in state s_0. Second, I assume that at least one entity in the subsystem under consideration exists now, otherwise the subsystem would be empty. In the example, for instance $ss_1 = \{e_1, e_2\}$. I will discuss later how the results of this analysis can be used in situations that does not comply with these two assumptions.

Moreover, as in the previous analysis, all states considered in this proof are sane. I have already proved that sanity is a system invariant.

Enforcing Subsystems

To enforce mandatory, subsystem boundaries, I build upon the results from Section 6.3. It turns out that the main classic take-grant theorem of Section 6.3 is not of much direct use in this proof. However, the central Lemma 16 of Section 6.3 can be used to good advantage. Recall, that for two existing entities in any $sane$ state s, they can not become transitively connected after executing a single command, unless they were already transitively connected before:

$$[\![step\ cmd\ s \vdash e_x \leftrightarrow^* e_y;\ isEntityOf\ s\ e_x;\ isEntityOf\ s\ e_y;\ sane\ s]\!]$$
$$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$$

The main lemma I would like to show in this section is that single step execution does not increase the authority of a subsystem; that is, a lemma similar to the following:

$$c :> subSysCaps\ s\ e_s \Longrightarrow c :> subSysCaps\ (step\ cmd\ s)\ e_s$$

which can then be lifted to command sequences by induction.

I start the analysis by closely examining the term c :> $subSysCaps\ s\ e_s$. This term can be expressed directly by referring to the entities of subsystem e_s. Then by definition, we get for any entity that is transitively connected to e_s in s, that there is no capability that contains more authority than c. Formally:

Lemma 19. *Unfolding* `c :> subSysCaps s e_s`*:*

$$(c :> subSysCaps\ s\ e_s) = (\forall e_x.\ s \vdash e_x \leftrightarrow^* e_s \longrightarrow c :> caps\_of\ s\ e_x)$$

Proof. By unfolding the definitions of `:>`, `subSysCaps` and `subSys`. □

So, to show that subsystem boundaries are not violated, I need to show that the above property is true for all states derived from `s`.

Lemma 20. *Single execution steps do not increase subsystem authority:*
$[\![sane\ s;\ isEntityOf\ s\ e_s;\ isEntityOf\ s\ e;\ entity\ c = e;\ c :> subSysCaps\ s\ e_s]\!]$
$\Longrightarrow c :> subSysCaps\ (step\ cmd\ s)\ e_s$

Proof. One may assume a sane state `s`, and two entities e_s and `e` such that `e` is the entity the capability `c` points to. Let us consider the situation after executing a single command on `s`. Let the new state be `s'`. After unfolding the goal as above, we may additionally assume $s' \vdash e_x \leftrightarrow^* e_s$ for an arbitrary, but fixed e_x, and now have to show $c :> caps\_of\ s'\ e_x$. We proceed by case distinction on whether $c :> caps\_of\ s\ e_x$, that is if `c` already dominated all authority of e_x before the command was executed.

- We start with the case $\neg\ c :> caps\_of\ s\ e_x$. That means, in `s` the entity e_x already had a capability with more authority than `c`.

  We know by assumption that `isEntityOf s` e_s. Moreover, given we are considering `sane` states, e_x also must already exist in `s` — otherwise it could not have any capability, in particular not one stronger than `c`. Given both of these are entities in `s` and we know by assumption that $s' \vdash e_x \leftrightarrow^* e_s$, we get via Lemma 16 that $s \vdash e_x \leftrightarrow^* e_s$. But if that is the case, then e_x was already part of the e_s subsystem in `s`, and thus the subsystem e_s in `s` already had more authority than `c`, which is a contradiction.

- In the second case, we assume $c :> caps\_of\ s\ e_x$ and we still need to show that the execution step did not add authority stronger than `c`. Here, we proceed by case distinction over the command that was executed. The only interesting cases are `SysGrant` and `SysCreate`.

  If the operation was some `SysGrant` $e_g\ c_1\ c_2$ `R`, the capability `diminish` c_2 `R` is being added to the entity of c_1. If that entity is not e_x, then $caps\_of\ s'\ e_x = caps\_of\ s\ e_x$ and we are done. If the entity of c_1 happens to be e_x we need to check that `diminish` c_2 `R` has less authority than `c`. Via `sane` we know that e_x exists in `s` (c_1 points to it) and from $s' \vdash e_x \leftrightarrow^* e_s$ we get again via Lemma 16 that $s \vdash e_x \leftrightarrow^* e_s$. From `legal` we know that $\{c_1, c_2\} \subseteq caps\_of\ s\ e_g$ (where e_g is the entity initiating the grant operation) and `Grant` $\in$ `rights` c_1 and therefore $s \vdash e_x \leftrightarrow e_g$. By transitivity, e_g is in the subsystem of e_s, and by assumption $c :>$ `subSysCaps s` e_s. The capability c_2 is part of e_g, hence part of `subSysCaps s` e_s and therefore has less authority than `c`. The diminished version of c_2 has even less (or equal) authority and in particular less than `c`. This concludes the grant case.

  If the operation was `SysCreate` $e_c\ c_1\ c_2$, then there are three possibilities: e_x is the entity that was created, or it was the target of c_2 that gets the new capability to the entity that was created, or it is neither of these. In the latter case, $caps\_of\ s'\ e_x = caps\_of\ s\ e_x$ and we are done. In the first case, $caps\_of\ s'\ e_x$ `= {}` and since trivially $c :> \{\}$, we are done as well. In the remaining case, we add the capability

`newCap s` to `caps_of s` e_x. We know that `newCap s` points to `next_id s` which was not an entity in `s`. On the other hand, we know by assumption that the target of `c` is an entity in `s`. Thus, the addition of `newCap s` to `caps_of s` e_x does not increase authority over the target of `c`. This concludes the create case and the proof.

□

The above lemma is essentially the induction step of the final theorem. In addition, there is one more simple lemma that I need. Observe that if `e` is an entity in `s`, then it will be an entity in any subsequent state:

Lemma 21. *Entities are preserved by execution:*
`isEntityOf s x ⟹ isEntityOf (execute cmds s) x`

Proof. By induction on `cmds`, then unfolding the definitions and observing that `next_id` never decreases. □

This leads us to the final isolation theorem.

Theorem 2 (Isolation of authority)**.** *Given a sane state `s`, a non-empty subsystem* e_s *in `s`, and a capability `c` with a target entity `e` that exists in `s`, if the authority of the subsystem does not exceed `c` in `s`, then it will not exceed `c` in any future state of the system.*
⟦`sane s; isEntityOf s` e_s`; isEntityOf s e; entity c = e; c :> subSysCaps s` e_s⟧
`⟹ c :> subSysCaps (execute cmds s)` e_s

Proof. By induction over the command sequence, and using the Lemma 20, Lemma 21, and Lemma 3 to prove the induction step and the required preconditions. □

This proves that seL4 protection model is capable of enforcing subsystem boundaries. The authority that any subsystem collectively has over another entity can not grow beyond what is conferred.

Going back to our example in Figure 6.5, this means that no entity in subsystem ss_1 will ever gain any authority over the entity e_5. Moreover, none of these entities will gain more authority than `{Write}` over entity e_3.

6.4.1 Example System

The proof in the previous section shows that the protection model is capable of enforcing subsystem boundaries — once created, the collective authority within a subsystem over another entity in a different subsystem can not grow. But how can we create these subsystems in the first place?.

In this section, I show how subsystems are bootstrapped by an abstract resource manager. Note that there are a number of methods a resource manager can use.

After the system bootstraps itself, it creates the state for the resource manager — the state s_i formally defined as: $s_i \equiv$ (|`heap = [0 ↦ {allCap 0}], next_id = 1`|). This state, contains only the resource manager, with full access rights to itself and with the authority over all the unused physical memory (see also Section 5.3.4).

It is the responsibility of the resource manager to create and set up the rest of the system, such that subsystem boundaries are not violated in any derived state. Once the subsystems are setup the resource manager exits the system by removing all its authority to freshly created subsystems. Moreover, the resource manger is trusted to enforce subsystems.

For the sake of clarity, the state created by the resource manager, after bootstrapping all user-level applications and exiting is called s_0. Note that all applications in s_0 and the capabilities these applications possess in s_0 are strictly under the control of the resource manager.

The current mechanisms used by the resource manager for bootstrapping is as follows. For each of the subsystems, it creates a *subsystem resource manager* who is responsible for bootstrapping the rest of that particular subsystem. This scheme stems from a major application domain of seL4: running para-virtualised guest operating systems (OS) in each subsystem. For each subsystem, the resource manager creates a single entity — the guest OS kernel. It is then up to the guest OS to bootstrap and manage the remainder of that subsystem.

Given that all the user-level applications in s_0 and their capability configurations are created by the resource manager, it can be assumed that the resource manager has information about the global system state. In this case, the global system state is provided to the resource manager through a simple specification language. This language allows the developer to specify the subsystems that should be created together with the authority they should possess over one another and the amount of physical memory that should be committed to each subsystem manager. Given below is an example written in this specification language:

```
"ss0" {
        text { 1024 to 4096 };
        data { 4096 to 5120 };
        resource { 4 };
        comm { this → ss2 };
};

"ss1" {
        text { 5120 to 6144 };
        resource { 4 };
        comm { this → ss1 };
};
```

To explain the above specification, the system has two subsystems *ss0* and *ss1*. Given this specification the resource manager creates two subsystems by creating two subsystem resource managers. The keywords *text* and *data* specify where to find the text and data segment of the program. The *resource* specifies the amount of physical memory — untyped capabilities — each should have access to. Keyword *comm* specifies the required communication channels. We write *comm {this → ss1}* to say that the current entity should be able to send information to *ss1*.

For *ss0* and *ss1* to be subsystems, what should be guaranteed is that neither *ss0* nor *ss1* has a capability with *Grant* authority that points to the other. Formally, $\neg\ s \vdash ss0 \leftrightarrow ss1$. This property is guaranteed by our specification language — there is no language construct to specify such a connection. Note that this does not mean that there will be no grant operations in the system at all. The subsystem resource managers are free to provide grant authority within each of the subsystems. The specification language merely excludes the language constructs that break the policy it enforces.

This specification ”compiled” into a compact representation is called the *global state*. Given this global state, the resource manager starts bootstrapping the system. Initially, it creates a pool of entities. Then it populates each entity in accordance with the above

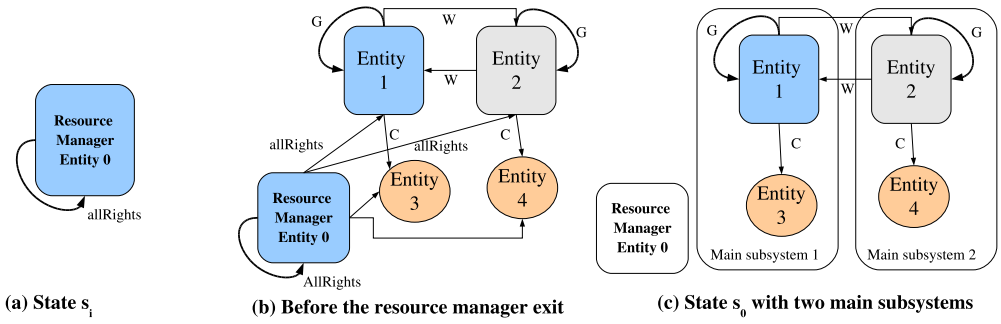


Figure 6.6: Example Subsystem Configuration

description. In addition to what is in the description, each entity, or in this case, each subsystem resource manager, gets a capability to itself with *Grant* authority to enable authority distribution within the subsystem itself.

A graphical representation of the state created by the system after bootstrapping itself, is shown in part (a) of Figure 6.6. As mentioned earlier, this state consists of a single entity: the resource manager (entity 0). In the diagram, *allRights* refers to full authority, and *C, G, R, W* is used to denote *Create*, *Grant*, *Read* and *Write* rights respectively. The configuration soon after creating and populating each entity with the required authority, according to the above specification is given in part (b) of Figure 6.6. Note that in part (b) of the diagram the two entities (entity 1 and 2) are connected through the resource manager which formally means that both entities still inhabit the same single subsystem. The final task of the resource manager is to break these connections. Once all the required capabilities are in place, the resource manager removes its own capabilities to the bootstrapped entities and exits. Thereby, it breaks the connection and makes them isolated subsystems, as shown in part (c) of Figure 6.6 — which represent state s_0. Once the resource manager exit the subsystems which were inactive thus far, become active and start executing.

One possible sequence of commands the resource manager (entity 0) can execute to produce s_0 is given below:

```
cmdSeq ≡
[SysCreate 0 (allCap 0) (allCap 0), SysCreate 0 (allCap 0) (allCap 0),
 SysCreate 0 (allCap 0) (allCap 0), SysCreate 0 (allCap 0) (allCap 0),
 SysGrant 0 (allCap 1) (allCap 1) {Grant},
 SysGrant 0 (allCap 1) (allCap 2) {Write},
 SysGrant 0 (allCap 1) (allCap 3) {Create},
 SysGrant 0 (allCap 2) (allCap 2) {Grant},
 SysGrant 0 (allCap 2) (allCap 1) {Write},
 SysGrant 0 (allCap 2) (allCap 4) {Create},
 SysRemove 0 (allCap 0) (allCap 1), SysRemove 0 (allCap 0) (allCap 2),
 SysRemove 0 (allCap 0) (allCap 3), SysRemove 0 (allCap 0) (allCap 4)]
```

where

allCap e ≡ (|*entity = e, rights = allRights*|) and

grantCap e ≡ (|*entity = e, rights = {Grant}*|).

The direct, formal description of the final state created by the resource manager (i.e. the state shown in part (c) of Figure 6.6) is given below:

```
s0 = (|heap = [1 ↦ {grantCap 1, writeCap 2, utCap 3}, 2
↦ {grantCap 2, writeCap 1, utCap 4}], next_id = 5|)
```

where

`writeCap e` $\equiv$ $(\!|$`entity = e, rights = {Write}`$|\!)$ and
`utCap e` $\equiv$ $(\!|$`entity = e, rights = {Create}`$|\!)$.

Note that in s_0 all subsystems, including `1` and `2`, have only one entity. Strictly speaking, there are 5 subsystems in s_0, each with one entity. But only the entities `1` and `2` contain capabilities. They constitute the two main subsystems. The two main subsystems thus created cannot increase the authority they have over each other. For example, we can show:

Lemma 22. *For no sequence of commands can the subsystem `1` gain authority over entity `4` which corresponds to the physical memory resources of subsystem `2`.*

$\forall$ `cmds. noCap 4 :> subSysCaps (execute cmds` s_0`) 1`

Proof. Firstly, we note `sane` s_0. Moreover, by examining s_0 we see that `isEntityOf` s_0 `1` and `isEntityOf` s_0 `4`. Then we observe that `noCap 4 :> subSysCaps` s_0 `1`. Given these facts, we can directly apply Theorem 2 and conclude. □

6.4.2 Relation to Concrete System

The bootstrapping mechanism described in the previous section is based on the abstract protection model of seL4. To enforce subsystems, at this abstract level, the resource manager needs to guarantee that any two entities (subsystem resource managers) are not *connected*. In this section, I describe how these abstract concepts relates to the concrete kernel and thus what restrictions a concrete resource manager needs to adhere in order to enforce subsystems.

As I mentioned earlier, abstract entities correspond to threads and their associated CSpace and VSpace in the concrete system. At the abstract level, a *connection* implies a capability in ones possession that points to the other, and with *Grant* authority. For enforcing subsystems, such *connections* needs to be avoided. At the concrete level preventing two threads from being connected implies:

- There should not be any GrantIPC channel between any two threads,
- CNodes and page-tables to which one thread has *Grant* authority cannot be shared between threads, and
- A capability with *Grant* authority to the TCB of a thread cannot be placed in another threads possession.

Recall from our discussion in Section 5.4, that the above three restrictions cover all possibilities of propagating a capability from one process to another (in the current state). Thus, if these restrictions are enforced at the concrete level, then it implies the corresponding entities are not *connected* at the abstract level. Then from the proof, subsystem boundaries cannot be violated in any derived state.

6.4.3 Reducing Resource Manager's Complexity

The enforcement of the subsystem policy depends on the correctness of the resource manager. If the resource manager distributes the capabilities according to the (arguably) simple rules specified above, then seL4's protection model guarantees the policy enforcement.

However, bootstrapping an application, in particular a paravirtualised OS, needs more than distributing capabilities. This is solved by having another bootstrapping state — the resource manager distributes the capabilities and then calls a *secondary* resource manager, integrated to the guest OS that sets up a suitable environment for the guest OS to run.

The motivation behind this two stage bootstrapping is to reduce the complexity of the security critical part. If the resource manager distributes capabilities correctly, then the subsystems will hold, irrespective of what happens within the guest.

6.5 Information Access Control

Subsystems are compartments for confining capabilities, and therefore the right to use a region of physical memory — a subsystem can only use regions of memory that were granted to it by the initial resource manager and nothing more.

Besides physical memory, the seL4 protection model can also be used to reason about *direct* and *indirect* access to information via *overt* channels.

A primitive designed to facilitate information access is called an overt channel [Lam73]. An illustrative example of an overt channel is the IPC primitive. The discussion in this section deals *only* with accessing information via overt channels. Hereafter I use the term access to information to refer to overt access to information.

Access to information can be either direct or indirect via a proxy. I explain these two access patterns latter.

The access control policy that I investigate is the bidirectional isolation of access to information. To facilitate the discussion, I call a set of entities that may access information from one another, either directly or indirectly, a *flowDomain*. Moreover, if the flowDomains of two entities, say x and y, are disjoint then I call the domains *isolated*. Informally, a flowDomain is isolated if all the entities that possess the ability to, either directly or indirectly, access information from an entity in the flowDomain is within the flowDomain itself. Formal descriptions of these terms are provided in Section 6.5.1.

The motivation behind bidirectional (or complete) isolation is the application of seL4 as a virtual machine monitor. The proof demonstrates that the seL4 protection model is capable of completely isolating the virtual machines from another and identifies the invariants required for doing so.

For achieving such isolation, however, the definition of a subsystem is not strong enough; because the definition does not preclude inter-subsystem communication. However, most of the lemmas and especially the proof engineering techniques from the subsystem proof are used in the following analysis. In fact, it was possible to prove most of the (new) lemmas by applying the same Isabelle commands.

The remainder of this section is organised in the following manner: Section 6.5.1 introduces formal definitions and predicates regarding information access, together with a rationale behind using it. Then Section 6.5.2 provides a formal proof which demonstrates that seL4 protection model is capable of enforcing isolation. Finally, Section 6.5.3 discusses the implications of isolation on the previous bootstrapping mechanism.

6.5.1 Isolated Components

As indicated above, isolation is the bidirectional absence of access. That is to say that, information can neither get in or get out from a set of isolated entities. Since it is bidirectional, by definition, the isolation property should be commutative.

Now let me examine the conditions under which an entity e_1 can access information from another entity e_2 in some state s. There are two *direct* possibilities: in state s (a) e_1 has a capability with *Read* authority that points to e_2, or (b) e_1 has a capability with *Write* authority that points to e_2. The former case facilitates a read operation and the latter a write operation. These two conditions cover the *de-jure* information access; accesses that are directly authorised by the access control system.

In state s, e_1 might not have direct access to e_2's information, but might have a capability with *Grant* authority that points to e_2 which would allow e_1 to setup such a channel.

In summary, if e_1 is in possession of a capability with at least one of $\{$ *Read*, *Write*, *Grant* $\}$, rights that points to e_2, then it can, in the two former cases directly read or write information from and to e_2, or in the latter case setup an information access channel in a subsequent state. To capture this requirement, I introduce the *grwCap*:

```
grwCap :: entity_id ⇒ cap
grwCap e ≡ (|entity = e, rights = {Read, Write, Grant}|)
```

Note that all access rights that allow or may lead to a state that would allow a direct information access channel to some entity y is contained in *grwCap* y capability. In state s, if entity x has a capability that points to y and contains *at least* one of these rights, then x already has or can setup a direct channel with y. I denote this by writing *canGRW* s x y, and its definitions is as follows:

```
canGRW :: state ⇒ entity_id ⇒ entity_id ⇒ bool
canGRW s x y ≡ grwCap y ≍ caps_of s x
```

The infix operator $\asymp$ is used to indicate that a set of capabilities provides at least one of the access rights contained in the given capability.

```
c ≍ C ≡ ∃c'∈C. entity c' = entity c ∧ rights c' ∩ rights c ≠ {}
```

When analysing the access to information, *Read* and *Write* access rights are symmetric [Sha99]: x can write to y has the same effect as y can read from x. The only difference is who initiates the access — which is irrelevant, since we are trying to prevent all possible accesses. Further, as I have shown in the previous section, it is irrelevant who has the capability with *Grant* authority, because grant arcs can be reversed.

So, to compute the entities that may directly access information we need the symmetric closure of the *canGRW*. I call this *canAccess* and write *(x, y)* ∈ *canAccess s* to denote x and y satisfies the *canAccess* relationship in state s.

```
canAccess :: state ⇒ (entity_id × entity_id) set
canAccess s ≡ {(x, y). canGRW s x y ∨ canGRW s y x}
```

Given a system state s, *canAccess* s returns a set of pairs of entities that can directly communicate or can setup such direct communication channel with one another.

In addition to direct access to information covered by *canAccess*, entity x may access information from y indirectly by collusion with a third party. For example, x may write some information to z which in turn writes that to y, yielding an indirect write from x to y. Such accesses, known as *de-facto* access, can take place via a chain of entities. I cover de-facto accesses by taking the transitive closure and reflexive closure of *canAccess*. I call this *canAccessTrans*:

```
canAccessTrans :: state ⇒ (entity_id × entity_id) set
canAccess* s ≡ (canAccess s)*
```

The *canAccessTrans* function, given a protection state, returns a set of pairs of entities that can directly or indirectly access information from one another. The syntax $s \vdash x \rightleftharpoons^*$

y means x and y are in the *canAccessTrans* relationship in s.

Given a state s, using *canAccessTrans* function I can compute the set of entities a given entity x may directly or indirectly access information from. I call this set an *accessDomain*:

```
accessDomain :: state ⇒ entity_id ⇒ entity_id set
accessDomain s x ≡ {e. s ⊢ e ⇌* x}
```

The *accessDomain* function returns the set of all entities e such that e and x satisfies the *canAccessTrans* relationship in state s.

Isolation Predicate

If entities x and y are isolated from one another in state s, then the two *accessDomains* computed for x and y in that state are disjoint. Formally:

```
isolated :: state ⇒ entity_id ⇒ entity_id ⇒ bool
isolated s x y ≡ accessDomain s x ∩ accessDomain s y = {}
```

In other words, in state s, if there is no common entity that both x and y may directly or indirectly access information from, then they are *isolated* in state s.

6.5.2 Proof of Isolation

Similar to subsystems, isolating two entities in the initial state (s_0) is trivial — this state is created by the initial resource manager, hence all the capabilities an entity possesses and therefore its *accessDomain* is under the strict control of the initial resource manager. What is required is to reason about what happens to two initially isolated domains as the system evolves.

To show that the seL4 protection model is capable of enforcing isolation, I need to prove that two initially isolated domains will always remain isolated after executing any sequence of commands. The isolation theorem I need to prove is:

```
isolated s x y ⟹ isolated (execute cmds s) x y
```

Similar to the previous proofs, I make use of the fact that sanity is a system invariant—all states considered in this proof are *sane*. Moreover, the scope of our reasoning is limited only to existing entities. That is, the two entities x and y must exist in the current state. Once again, note that I am not excluding the creation of entities: after executing a sequence of commands the access domain of x, depending on the commands it executes may contains entities which were not present in the previous state. These subsequently created entities are also covered by the proof.

This proof is very similar to the subsystem proof. The main difference is the use of a different closure: here we consider the transitive, symmetric and reflexive closure of *grwCap*, where as the subsystem proof considered the transitive, symmetric and reflexive closure of *grantCap*. In theory, one should be able to derive this proof from the lemmas in the previous section.

However, it was much more convenient to take the proof techniques from the previous section and prove the required lemmas about the new closure separately rather than deriving from existing ones.

The isolation proof itself is about 500 lines of Isabelle scripts. Where ever possible I used existing proofs from the previous section. In particular, previous lemmas relating to sanity and existence of entities are directly used here. However, lemmas relating to

canAccess relationship are proved separately, but using the same proof techniques. In fact, in most cases, the Isabelle commands for proving the lemmas were almost identical to its counterpart — the difference between the two closures was automatically handled by Isabelle. As such, I will only provide a brief overview of the proof, and show the essential lemmas together with a very brief description of the techniques used for proving.

Before introducing the proof, I introduce some properties of *canAccess* and the relationship between $canAccess^*$ and $connected^*$.

In a *sane* state, two entities in *canAccess* relationship have the following property:

Lemma 23. *In a sane state only existing entities can be in the* canAccess *relationship:*
$[\![sane\ s;\ s \vdash x \rightleftharpoons y]\!] \Longrightarrow isEntityOf\ s\ x \wedge isEntityOf\ s\ y$

Proof. By unfolding the definition of *canAccess* and *sane* □

By definition, the $canAccess^*$ and $connected^*$ relates to one another in the following manner:

Lemma 24. *If two entities are in the* $s \vdash x \leftrightarrow^* y$ *relationship, then they are in the* $s \vdash x \rightleftharpoons^* y$ *relationship:*
$s \vdash x \leftrightarrow^* y \Longrightarrow s \vdash x \rightleftharpoons^* y$

Proof. By inducting over the transitive and reflexive closure and by using the fact that by definition $(x,y) \in connected \Longrightarrow (x,y) \in canAccess\ s$ □

Furthermore, by definition of *isolated*:

Lemma 25. *Isolated entities are not in* $canAccess^*$ *relationship:*
$isolated\ s\ x\ y \Longrightarrow \neg\ s \vdash x \rightleftharpoons^* y$

Proof. By unfolding the definitions. □

Now I start the analysis by investigating the effect executing a single command has on the $canAccess^*$ relationship. Then I lift this relationship to a command sequence by induction. Here also, similar to the previous proof, I treat transporters — commands that do not introduce new entities into the system — separately from *SysCreate* which does introduces a new entity. For transporters I proved:

Lemma 26. *In a sane state, transport commands preserve* $canAccess^*$*:*
$[\![sane\ s;\ \forall e\ c_1\ c_2.\ cmd \neq SysCreate\ e\ c_1\ c_2;\ step\ cmd\ s \vdash x \rightleftharpoons^* y]\!]$
$\Longrightarrow s \vdash x \rightleftharpoons^* y$

Proof. Firstly, note that derived state is *sane*, which follows from the Lemma 2. Next, we induct over the transitive and reflexive closure. If two entities x and y are in *canAccess* relationship in the derived state, then from Lemma 23 both of them must exists in the derived state and since transporters do not introduce new entities they must exist in s. Except for *SysGrant*, none of the transport commands add capabilities, and therefore cannot lift two entities into the *canAccess* relationship unless they are already in the relationship in state s. In the case of *SysGrant*, suppose some entity z grants x a capability, with appropriate rights to y. Then by definition $s \vdash x \leftrightarrow z$ and $s \vdash z \rightleftharpoons y$. Since $s \vdash x \leftrightarrow z$, from Lemma 24 and the definition of transitive closure $s \vdash x \rightleftharpoons z$. Then since $s \vdash z \rightleftharpoons y$ we can conclude $s \vdash x \rightleftharpoons^* y$. □

For the *SysCreate* operation I proved:

Lemma 27. *Let* `x` *and* `y` *be two entities in a sane state* `s`*. Suppose* `x` *already exists in state* `s` *and* `x` *and* `y` *are in the* `canAccess`* *relationship in the derived state after executing a* `SysCreate`*. Then, if* `y` *is the newly created entity, then* `x` *and the entity that initiated the create operation (entity* `e`*) are in the* `canAccess`* *relationship in* `s`*, and if not* `x` *and* `y` *are in* `canAccess`* *relationship is* `s`*:*
$[\![$`sane s; isEntityOf s x; step (SysCreate e` c_1 c_2`) s` $\vdash$ `x` $\rightleftharpoons^*$ `z`$]\!]$
$\Longrightarrow$ `if z = next_id s then s` $\vdash$ `x` $\rightleftharpoons^*$ `e else s` $\vdash$ `x` $\rightleftharpoons^*$ `z`

Proof. Using the same proof techniques as Lemma 15. □

Given these two lemmas, I can now prove that execution of a single command does not violate isolation:

Lemma 28. *If two existing entities* `x` *and* `y` *are isolated in a* `sane` *state* `s`*, then they will remain isolated after executing a single command:*
$[\![$`sane s; isEntityOf s x; isEntityOf s y; isolated s x y`$]\!]$
$\Longrightarrow$ `isolated (step cmd s) x y`

Proof. The proof obligation here is to show that it is not possible that some entity `z` is in the `canAccess`* relationship in the derived state with both `x` and `y`. Firstly, we note that the derived state is `sane` and that $\neg$ `s` $\vdash$ `x` $\rightleftharpoons^*$ `y`. These follow from Lemma 2 and Lemma 25 respectively.

Now, we do a case distinction on the command. We treat transporters separately from `SysCreate`. For transporters, from Lemma 26 we see that `s` $\vdash$ `z` $\rightleftharpoons^*$ `x` and `s` $\vdash$ `z` $\rightleftharpoons^*$ `y`. Moreover, from the definition, `canAccess`* commutes and therefore we see that `s` $\vdash$ `x` $\rightleftharpoons^*$ `y`. This is a contradiction.

For `SysCreate`, it follows from Lemma 27 that if `z` is the newly created entity, then both `x` and `y` must be in the `canAccess`* relationship with some entity `e` — the entity initiating the create operation in state `s` — or if it is not then `x` and `y` are in `canAccess`* relationship with `z` in state `s`. In either case, we see that `x` and `y` are in transitive relationship — either via `e` or `z`. Thus, we can conclude the proof by contradiction. □

Then I lift Lemma 28 to a sequence of commands by induction to prove the final isolation theorem:

Theorem 3. *Suppose we have two entities* `x` *and* `y` *already existing in* `isolation` *in a* `sane` *state* `s`*. Then, they will remain* `isolated` *in any subsequent derived state:*
$[\![$`sane s; isEntityOf s x; isEntityOf s y; isolated s x y`$]\!]$
$\Longrightarrow$ `isolated (execute cmds s) x y`

Proof. By induction over the command sequence, and using Lemma 28, Lemma 21, and Lemma 3 to prove the induction step and the pre-conditions required for the induction step. □

The above theorem proves the ability to enforce isolation between entities: if two entities are bootstrapped in a manner such that they are isolated, then the seL4 protection model guarantees that they will remain isolated in any derived state, reachable via the execution of any sequence of commands.

6.5.3 Enforcing Isolation

In this section I discuss how to bootstrap isolated components and provide an example sequence of commands that an abstract initial resource manager can use in doing so. The bootstrapping mechanism is derived from that used for subsystems (see Section 6.4.1). As such, its main application area is bootstrapping para-virtualised operating systems in each isolation domain.

Similar to bootstrapping subsystems, for each isolation domain, the initial resource manager creates a single entity (or thread) — the *domain resource manager*. The functionality of a domain resource manager is similar to that of a subsystem resource manager.

As I have discussed in Section 6.4.1 when granting authority to each concrete, subsystem resource manager (thread) the initial resource manager adheres to the following rules:

- There are no GrantIPC channel between any two threads,
- CNodes and page-tables to which one thread has `Grant` authority cannot be shared, and
- A capability with *Grant* authority to the TCB of a thread cannot be placed in another threads possession.

The authority distribution rules required for enforcing isolation, is more stringent than those for enforcing subsystems; because subsystems do not preclude the ability to access information across its boundary. We only disallow the propagation of authority across a subsystem boundary. For isolation, on the other hand we need to prevents both the propagation of authority and the access to information across an isolation boundary. As such, the rules required for isolation are more restrictive. These rules are:

- There are no IPC channel of any sort between any two threads. This includes both the synchronous and the asynchronous IPC primitives provided by the kernel and channels that are setup using shared Frame objects.
- CNodes and page-tables to which one thread has `Grant` authority cannot be shared, and
- A capability (with any authority) to the TCB of a thread cannot be placed in another threads possession.

Now I demonstrate how an abstract initial resource manager can bootstrap a system with two isolated entities. Using our normal notation, a formal description of a protection state with two isolated components soon after initial resource manager has exited is given below:

s_{iso} =
(|heap = [1 ↦ {grantCap 1, utCap 3}, 2 ↦ {grantCap 2,
utCap 4}], next_id = 5|)

The two entities; entity `1` and `2`, corresponds to the two domain resource managers. Each of the domain resource manager is authorised to grant capabilities to itself and create other entities. As an example, for entity `1`, the above authorities are conferred via `grantCap 1` and `utCap 3` respectively.

A possible sequence of commands the initial resource manager (entity 0) can execute, starting from s_i, to produce s_{iso} is given below:
`isoCMD` ≡

```
[SysCreate 0 (allCap 0) (allCap 0), SysCreate 0 (allCap 0) (allCap 0),
 SysCreate 0 (allCap 0) (allCap 0), SysCreate 0 (allCap 0) (allCap 0),
 SysGrant 0 (allCap 1) (allCap 1) {Grant},
 SysGrant 0 (allCap 1) (allCap 3) {Create},
 SysGrant 0 (allCap 2) (allCap 2) {Grant},
 SysGrant 0 (allCap 2) (allCap 4) {Create},
 SysRemove 0 (allCap 0) (allCap 1), SysRemove 0 (allCap 0) (allCap 2),
 SysRemove 0 (allCap 0) (allCap 3), SysRemove 0 (allCap 0) (allCap 4),
 SysRemove 0 (allCap 0) (allCap 0)]
```

By using Theorem 3 I can show that the two entities — entity `1` and `2` — remain isolated in any derived state:

Theorem 4. *The two isolation domains in* s_{iso} *remains isolated in any derived state:* `isolated (execute cmds` s_{iso}`) 1 2`

Proof. By using Theorem 3, noting that preconditions required for isolation are satisfied in s_{iso}. □

6.6 Summary

This chapter has analysed the formal characteristics of the seL4 protection model. The formal, machine-checked proofs presented here shows that the protection model is capable of enforcing *at least* two, commonly used policies — spatial partitioning and isolation.

The proofs identify a set of invariants an initial resource manager needs to enforce while bootstrapping the system to guarantee the enforcement of either policy. Moreover, the formal examples demonstrates that it is possible to bootstrap a system while adhering to the invariants identified by the proof.

Chapter 7

seL4::Pistachio

In this chapter I report the results of experiments conducted to measure the overheads introduced by the proposed model to the primitive kernel operations. These experiments characterise the performance of frequently used kernel primitives of a microkernel-based system, such as inter-process communication (IPC), thread operations, virtual memory primitives and in particular the performance of the kernel memory management interface and its impact on other primitives.

The motivation for analysing the micro-level performance is two fold. First, micro-level analysis exposes effects the model has on each microkernel primitive, which are not that obvious at a macro-level. Second, by knowing the performance characteristics of primitive operations, one can better structure the higher-level software which uses the primitives.

All results reported here were obtained on a *KZM-ARM11* evaluation board, which comprises an *ARM1136JF*-based [ARM05], Freescale™ i.MX31 processor running at 532MHz, a unified L2 cache of 128KB and 128MB DDR RAM. The proposed memory allocation model is implemented by a prototype microkernel called *seL4::Pistachio*. Implementation details presented in this chapter are based on the ARM11 implementation of the prototype kernel. seL4::Pistachio is loosely based on the *L4-embedded* [NIC05] code base.

For comparing the performance, I have selected *OK Labs'* [Ope] L4 kernel (version *2.1-dev.35*) or *OKL4* — a successor version of L4-embedded. Hereafter I call this kernel L4. Except for the proposed memory allocation model and the associated protection mechanism, L4 and seL4::Pistachio provide similar abstractions and functionality. They differ mainly in the way they manage in-kernel memory — L4 uses a standard in-kernel allocator in contrast to the proposed memory allocation scheme used by seL4::Pistachio. Besides that, the only other conceptual difference between the two kernels is the use of capabilities in seL4::Pistachio — which stems from the proposed protection model. Since the two kernels are similar in all other aspects, comparing their numbers will avoid any discrepancies not imposed by the proposed scheme.

Section 7.1 examines the *raw* cost of managing kernel memory from user-level and then examines different allocation techniques that can be used to reduce the overheads. The seL4 kernel is based on capabilities — all seL4 system calls require at least one capability to authorise the operation. Consequently, the cost of capabilities in the system call path is critical to the performance of seL4::Pistachio. Section 7.2 examines these costs, techniques a user-level application can use to reduce them, and the cost of managing a capability address space. Then, Section 7.3 analyses the effect capabilities have on primitive kernel operations by comparing the performance of seL4::Pistachio with the numbers for similar

operations on L4. Finally, Section 7.4 summarises the main findings of the chapter.

7.1 Cost of Managing Kernel Objects

Since all dynamically allocated memory within the kernel is represented as an object at the API level, seL4 objects can be categorised into two parts — architecture-independent and dependent objects. There are five architecture-independent objects in seL4 — Untyped, TCB, Endpoint, AsyncEndpoint and CNode . On the ARM architecture, there are three architecture-dependent object types that need kernel-memory allocation — PageDirectory, PageTable and Frame. Detailed discussion on the functionality of these objects can be found in Chapter 4.

To create any seL4 abstraction, a server needs to either allocate the corresponding kernel object or reuse an already allocated object from a different context. As an illustrative example, to create a thread a server (resource manager) either needs to allocate a TCB object or reuse an existing TCB from a different context. When the abstraction provided by an object is no longer required, a server can reclaim the underlying memory from the object and reuse it to implement other objects. In this section, I investigate the performance of the primitives provided by seL4 to manage kernel objects.

As I described in Chapter 4, there are three primitive operations to manage seL4 kernel objects — the *retype* method to allocate new kernel objects, the *revoke* method to reclaim memory from an object no longer required for the system and the *recycle* method to reuse an existing object in a different context. Following three subsections investigate the performance of each of the above primitives and the final subsection (Section 7.1.6) discusses a general scheme for managing kernel objects from user space.

Unfortunately, the performance of seL4 kernel memory management primitives are not directly comparable to L4 or other similar systems. This is because of the implicit nature of kernel-memory allocations in those systems — as and when the need arises, the kernel allocates memory implicitly without the knowledge of the user-level application. Comparing the cost of an implicit allocation with the cost of seL4 object allocation directly is unfair because the former includes the cost of policy decision made by the in-kernel allocator, which is not included in the latter. If we factor out the policy decision, then we are mainly left with the cost of object initialisation which is irrelevant to the topic. A fair comparison can be made by implementing a similar allocation policy to that used by the in-kernel allocator using the seL4 primitives and comparing the cost for standard OS operations such as thread creation, process creation, virtual memory objects etc. Such a comparison at a micro-level and at a macro-level is presented in Section 7.3 and Chapter 8 respectively.

The main focus of this section is to identify the raw costs of managing kernel objects, associated trade-offs and the effects different policies have on the overall management cost.

7.1.1 Cost of Allocating Kernel Objects

Allocation of kernel objects is performed by the *retype* method, implemented by the untyped memory (*UM*) objects. Before examining the performance measurements, I summarise the relevant parts of the retype method presented in Chapter 4.

Possession of an UM capability (*parent UM*) provides sufficient authority to allocate kernel objects in the corresponding region — by invoking a UM capability with the correct

Object type	Number of cycles
AsyncEndpoint	582
CNode (16 slots)	841
Endpoint	578
Frame (4KB)	2246
PageTable	2249
PageDirectory	15545
TCB	758
UM (4KB)	575
UM (16KB)	575

Table 7.1: The cost of allocating various kernel objects.

set of arguments, a user-level application can request that the kernel refines that region into other kernel objects (including smaller UM objects).

When the retype method creates new kernel object(s), it returns capabilities to the newly-created objects — one capability for each new object. The user-level application that created these new objects can then delegate all or part of its authority to others by using the appropriate capability copy operation — mint to create a CDT child with partial or equal authority or imitate to make a CDT sibling with equal rights. To facilitate the discussion in this section, I call the capabilities directly created via the retype operation, *master* capabilities and a copy made from master a *slave*.

Recall from our discussion in Chapter 4, the retype operation also consumes memory in terms of CNode slots for placing the master capabilities. These CNode slots are explicitly identified in the retype operation —- the arguments passed to the retype call includes specifying empty CNode slots for placing the master capabilities.

Table 7.1 shows the cost of allocating different kernel object types. The numbers shown in the table are the average of 500 measurements and the maximum coefficient of variation for any measurement is less than 2%. These measurements include the cost of marshaling system call arguments and trapping in and out of the kernel. Moreover, these were taken for a two-level capability tree; a representative CSpace layout of a large-scale server. This means that, every capability required to complete the operation is located within a two-level capability tree.

The retype operation, requires three lookups for; (a) locating the invoked UM capability, (b) locating the root CNode of the destination CSpace for the newly created capabilities, and (c) locating the CNode within the above CSpace for placing the capabilities. Once these capabilities are located, the kernel affirms that it is safe to allocate new objects using the located UM capability; i.e. the UM capability has no existing children and that the destination slots for placing capabilities are empty. As I explained in Chapter 4, this safety test takes constant time, and in the implementation requires only three memory references; one to locate the next CDT entry via the linked list and another two memory accesses to retrieve the required information. The collective cost of these operations, I call the *start-up* cost.

Besides the start-up cost, the other main contributing factor for the above measurements is the cost of initialising the allocated object. For instance, the high cost of allocating a Page-directory stems from the kernel initialising the 16KB of memory required for the object and the cost of allocating a frame or a page table is dominated by the work required

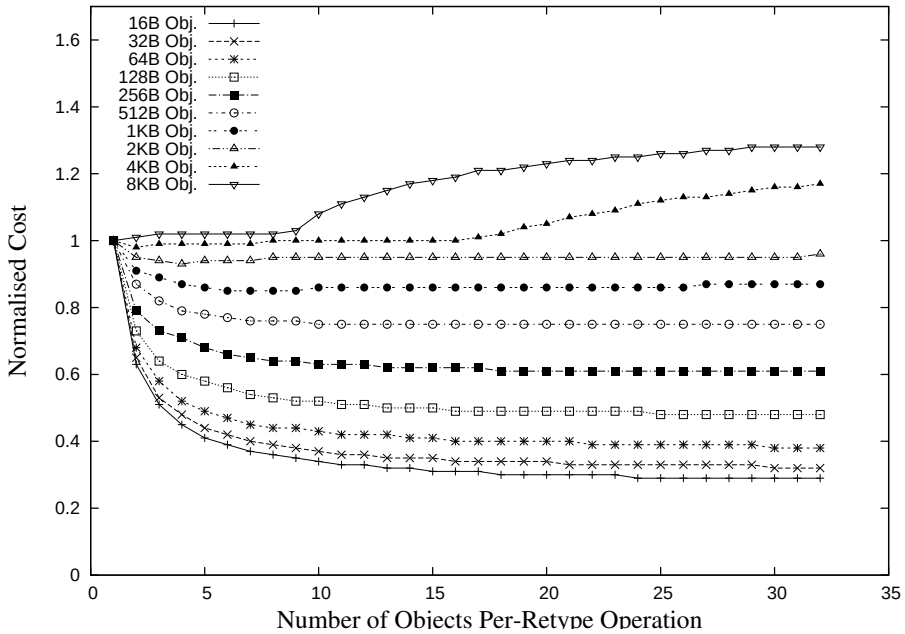


Figure 7.1: Variation of the object allocation cost against the object size and the number of objects created per retype operation. The *X* axis shows the number of objects created per retype and *Y* axis is the normalised cost.

to initialise the 4KB of memory implementing the object. Moreover, note that the cost of allocating a Page-directory is significantly higher than that of allocating a page table. By comparing the memory sizes, one may expect the cost of allocating a Page-directory to be about 4 times that of allocating a page table. The measurements however, show a significantly higher number. This increase in the cost is due to the trashing of the 16KB L1 cache of i.MX31 processor.

When creating UM objects, the kernel does not initialise the memory — the memory is initialised at the time of allocating *typed objects*. This is why there is no difference in the cost of creating 4KB and 16KB UM objects. Moreover, this is the base-line cost of creating a kernel object; for all other object types, the cost is this plus the initialisation cost.

Slab Allocation

One possibility of reducing the allocation cost is by using the slab allocation technique initially proposed to *SunOS 5.4* [Bon94] and later adopted (with various changes) by many main stream systems including Linux (Slab cache [vR01]) and FreeBSD (Zone allocator [MBKQ96]). The basic idea of slab allocation is to allocate a batch of objects at one time, store them in a cache and serve subsequent requests from this cache. Once an object is released, it is put back to the cache rather than destroyed.

One important requirement for implementing a slab allocation strategy is the ability to reuse objects from one context in another. In seL4 this operation is supported by the recycle method, which I discuss in Section 7.1.5.

The motivation here is that by allocating several objects per retype operation, the start-up cost can be amortised over the several objects created. Moreover, the scheme has natural synergy with client-server systems; where the server keeps a pool of objects out of which it serves its clients.

As one would imagine, slab allocation is not ideal for all situations. It improves the temporal performance, but depending on the application domain, may lead to underutilisation of memory. For instance, a memory constrained embedded device might trade the temporal gain from slab allocation to a scheme with better memory utilisation. Or in another context, the developer might prefer the speedup of slab-based allocation [MYHH08] because it reduces energy consumption by cutting down the execution time and hence improving the battery life. I envisage that seL4 mechanisms are flexible enough to support these diverse, domain specific, memory allocation schemes.

Figure 7.1 shows the normalised cost of allocating kernel objects of different sizes, when objects are created in batches of size 1 to 32. The *X* axis of the figure shows the number of objects allocated per retype operation and the *Y* axis shows the cost, normalised to that of creating a single object. The object size was changed by allocating a CNode with different number of slots. The memory required for the object is initialised at the time of its creation, and since every CNode slot is made-up of four machine words (16-bytes), when allocating a CNode with *X* slots the kernel needs to initialise 16*X* bytes of memory. Moreover, the measurements are taken with a hot data and instruction cache.

The results in Figure 7.1 shows that batch allocation reduces the cost per object, when the object is small in size; i.e. the amount of memory that needs to be initialised is small. For the smallest object considered in the above experiment, the cost reduces by over 65% (from 616 cycles to 207), when allocation is done in batches of 10 or more objects. When allocating small kernel objects the dominant cost is the start-up cost, and by allocating in batches this cost gets amortised over the number of objects.

When the size of the object grows, batch allocation becomes less effective. This is because the predominant cost of creating a larger object is that of object initialisation, which does not change with the number of object allocated, making batch allocation ineffective.

Additionally, note that the normalised cost increases with the number of objects allocated when the size of the object itself is large. For example, if the object is 8KB, then the normalised cost grows beyond 1 around the point when 8 objects are allocated per retype operation. For a 4KB object this point is around 16 objects per retype. This effect is caused by trashing the L1 data cache of the KZM-ARM11 processor.

Hybrid Allocators

Most systems adopt hybrid allocation techniques; usually a slab allocator running on top of a buddy system or similar allocation layer. The basic idea is to use different allocators for objects of different size — a buddy system for large objects and a slab allocator running on top of this layer managing memory for smaller objects. For example the FreeBSD zone allocator runs on top of a page allocator. When needed, it requests memory from the page allocation layer and uses that memory to slab allocate smaller objects.

Such a hybrid allocator can be implemented using the seL4 kernel memory management interface. The first layer will deal with Untyped memory objects. Starting with a large parent Untyped object this layer will keep on splitting the Untyped objects until a suitable size is reached. The slab allocator, depending on the objects it wants to allocate may required for different sized Untyped objects which is then used to allocate pools of other kernel objects.

7.1.2 Prototype Allocator

A slab allocator, with a few simplifications, is implemented for seL4. This allocator is used as the underlying kernel memory manager for all the experiments I report in this chapter.

An in-depth analysis of this allocator is presented in Chapter 8. Here I brief on the relevant parts of its implementation to aid the analysis of the performance measurements reported in this chapter which were obtained using this allocator in the back-end to manage kernel objects.

The allocator starts with access to some number of large Untyped memory objects and few small Untyped memory objects. Small Untyped objects are 4KB in size and cover regions of memory that cannot be used to implement large objects (which are 1MB) due to alignment requirements — the region covered by an Untyped object needs to be size aligned.

The allocator has two layers. The first layer is responsible for managing Untyped objects and the second layer manages all other (*typed*) kernel objects.

The first layer, when the need arises, retypes a large Untyped memory objects into 256 small Untyped objects and populates the object pool. Similarly, when a particular typed object pool is empty, the allocator requests for a suitably sized Untyped object (either a large or a small) from the first layer and repopulate the pool using that memory. All subsequent requests are satisfied from the objects in a particular pool and objects are only allocated when the pool is empty and when there is a pending request.

7.1.3 Memory Overhead of Untyped Capabilities

The memory overhead depends on system structure and construction. Though the focus of seL4 is on reasoning about and control of memory allocations and not minimising memory consumption, here I present a basic theoretical analysis of some selected scenarios to illustrate overheads.

The seL4 model incurs two types of memory overheads. First, to confer the authority over any kernel object, one requires a capability. I call such a capability an *object* capability. The use of object capabilities is common to all capability based systems (for example, EROS [Sha98] and L4.sec [Kau05b]). Thus, I exclude them from the discussion here.

However, note that each seL4 capability consumes 2 extra machine words to store the CDT information. The information in the CDT is central for the proposed memory management scheme.

Second, object capabilities in seL4 are created using Untyped capabilities (see Section 4.3 for more details). This process introduces an additional overhead to seL4 based systems when compared with other capability based systems. In this section, I present a basic theoretical analysis of the memory overhead due to Untyped capabilities.

The analysis presented here is based on the prototype allocator introduced in Section 7.1.2. In addition, I briefly comment on the suitability of other allocation schemes for a seL4 based system.

In summary, the prototype allocator uses a slab (like) allocation scheme to manage kernel objects. For each object type the allocator maintains a pool, out of which the requests are served. If the pool is empty, the allocator selects, depending on the size of the object to be allocated, a suitable Untyped object—either a 4KB or 1MB—creates kernel objects and populates the pool. For example, if Endpoint object pool is empty, the allocator creates 256 Endpoint object capabilities using a 4KB Untyped object and populates the pool. Thus, the

additional overhead (ignoring the object capabilities) is the memory consumed to confer the authority over the Untyped object.

Authority over an Untyped object is conferred using a capability. A capability consumes 4 machine words within the kernel — that is, 16bytes on a 32bit architecture. Furthermore, seL4 does not enforce any constraints on the placement of an Untyped object within an address space and its size. Thus, when it is required to minimise the memory overhead of an address space, one can avoid having intermediate page-table levels that are typically required to match the object size with its placement within an address space. As a result, to confer authority over a 4KB region, seL4 requires, *at best* (i.e. if one selects to avoid intermediate page-table levels) additional 16bytes; an overhead of 0.4% compared to the memory region. For a larger Untyped object the overhead reduces to 0.001%.

The prototype allocator refrains from creating Untyped objects smaller than 4KB — the amount of memory required to allocate a Frame object (on ARM). This restriction guarantees that any Untyped object can be used to allocate at least one Frame. Consequently, the maximum memory overhead due to Untyped capabilities is 0.4%.

Further, the prototype allocator keeps the Untyped capabilities contiguous in CSpace. Thus, the resource manager only needs two machine words to track the available memory regions (e.g. start address and a count). For a slab allocator that does not reuse the memory (to implement a different object type) the above information alone is sufficient to bookkeep Untyped objects; upon receiving a request for a particular object type, the resource manager picks the next Untyped object, allocate and populate the corresponding object pool and update its internal bookkeeping. When all the small Untyped objects are consumed, the resource manager creates another set of small Untyped objects such that these new capabilities are contiguous.

In comparison to a slab allocator, a buddy allocator would consume a lot more memory. This increase is due to the intermediate Untyped objects created by dividing the region into halves and the user-level bookkeeping required to track these intermediate objects. Ignoring the user-level bookkeeping, to allocate 256, 16byte objects — the smallest object size of seL4 — using a 4KB Untyped object a buddy allocator requires 8 splits and a total of 510 Untyped capabilities in those 8 levels. To explain the above computation, in each i^{th} level one requires 2^i number of Untyped capabilities. By adding all the Untyped capabilities in each intermediate level (i.e. $2^8 + 2^7 + 2^6 .... + 2^1$) we get 510. With each capability costing 16bytes, this is an unacceptable overhead of almost 200% (compared to the total region of memory). Even though the overhead can be reduced by reducing the size of Untyped objects, seL4 mechanisms are not totally conducive for buddy allocation. I envisage seL4 user-level managers to use slab or a hybrid allocation scheme to reduce the memory overhead of splitting large Untyped objects into small regions.

As such, depending on the allocation scheme, the seL4 memory model may introduce moderate to high memory overheads. But in return the model provides a mechanism to reason about and enforce memory management policies.

7.1.4 Cost of Revoking Memory

Possession of the parent UM capability that was used to allocate kernel objects provides sufficient authority to delete those objects. By calling the revoke operation on the parent capability, user-level resource manager can remove *all* the CDT descendants. Recall from our discussion in Chapter 4 that any capability that points to an object that resides within the memory region covered by the parent UM capability is a CDT descendant of the UM

Object type	Single master object	Additional cost per-master objects	Additional cost per-slave copy
AsyncEndpoint	458	204	180
CNode (16 slots)	867	609	180
Endpoint	454	204	180
TCB	589	341	180
UM (4KB)	346	96	180
Frames (4K)	452	200	180
PageTable	454	202	180
PageDirectory	459	211	180

Table 7.2: Cost of reusing memory by calling revoke on the parent UM capability

capability. Once the revoke operation is completed, the memory region covered by the parent UM object can be used to allocate other kernel objects. Moreover, when this revoke operation removes the last capability to an object, the kernel deactivates the object and breaks all internal dependencies between the deactivated object and others.

The cost of reusing memory from a parent UM object depends on several factors; (a) the number of objects which is equal to the number of master capabilities created by the previous retype, (b) the type and the status of the allocated objects and (c) the number of copies made from master capabilities.

The number of master copies determines the number of objects that need to be destroyed in the revocation path and the destruction itself depends on the type and the status of the object. The number of copies made from the master ones increases the number of capabilities that needs to be removed.

Under normal operation, a resource server will attempt to reuse a region of memory only when the current object(s) in the region are *inactive* and no longer required — a resource server will reclaim memory from a TCB object once the corresponding thread has exited, memory from an Endpoint object will be reclaimed when all messages destined to it are served, and so on.

Table 7.2 summarises the number of machine cycles required to reuse the memory from the parent UM when the previously allocated objects are in an inactive state. All numbers presented in this table are averaged over 500 iterations and the distributions had a maximum coefficient of variation of 3%. The first column of the table shows the number of machine cycles required to complete the revocation when there is only one inactive object of the given type and a single (master) capability pointing to it; which includes the cost of starting the revoke operation, destroying the object and removing the last capability that was pointing to the destroyed object. The second column shows the number of additional cycles required for destroying and removing the last capability reference to an inactive object. The difference between these two columns (roughly 250 cycles) is the fixed cost of invoking a revoke operation. The final column of the table shows the cost of revoking a copy of a master capability.

Out of these numbers, the only stand out is the relatively high cost of destroying a CNode. Even though the CNode being destroyed is inactive (i.e. contains no capabilities), unlike other objects, CNodes do not provide a simple test to affirm this. As such, the kernel needs to check each CNode slot, and affirm that there are no valid capabilities and therefore no CDT references from other CNodes to slots in the one being destroyed. The

Object type	Number of cycles
AsyncEndpoint	339
CNode (16 slots)	752
Endpoint	332
Frame (4KB)	2111
PageTable	2138
PageDirectory	15550
TCB	543
UM (4KB)	318
UM (16KB)	318

Table 7.3: The cost of recycling kernel objects.

limited bookkeeping provision within the CNode capability is used to store the information required for the guarded lookup — which is in the performance critical path, rather than storing metadata required for the relatively infrequent destroy operation.

In contrast, the status of other kernel objects can be established either by examining the object — in the case of two Endpoints objects, and TCBs, or by examining their capability — in the case of UM, Frame, PageTable and PageDirectory objects. This is an implementation choice rather than a consequence of the model.

7.1.5 Cost of Recycling Kernel Objects

To support slab style allocators, the kernel facilitates reuse of the same object in a different context via the recycle operation.

The possession of the master capability to an object is sufficient authority to recycle the object. Recall the definition of a master capability — a capability returned by the retype method is called a master capability. By calling the recycle operation on the master capability a resource manager can remove all the CDT children of the master, break dependencies between the recycled object and the other objects in the system and reinitialise the object.

Before looking at the performance measurements, the difference between revoke and recycle is worth mentioning. The revoke operation, when performed on a capability removes all its CDT children. If these removed capabilities are the only remaining reference to an object, then revoke will delete the object, breaking any dependencies other objects may have with the deleted object. This situation *only* occurs when revoke is performed on a parent UM object. For all other typed objects revoke will only remove a set of capabilities and leave the object as it is. Recycle on the other hand, removes all the CDT children of the capability, breaks all the dependencies the object may have with others and re-initialises the object.

Table 7.3 shows the number of machine cycles required to recycle an object of the given type. Similar to our previous measurements, the numbers shown in this table are averaged over 500 iterations and all the distributions had a coefficient of variation less than 2%. For the same reason mentioned in the previous section, the object being recycled is in an inactive state. Moreover, only the master capability points to the recycled object — there are no (slave) copies made from the master capability that needs removing. If there are slaves, similar to previous case, the cost increases on an average by 180 cycles per child

capability.

Recycling a smaller object—for example an Endpoint or a TCB—is cost effective than newly creating a single object from a parent UM. As I mentioned earlier, the cost of creating a smaller object is predominately the start-up cost of retype, as oppose to object initialisation cost. In comparison to retype, recycle is light weight, resulting in better performance.

On the other hand, for large objects such as Frames and Pagetables, there is no significant difference between the cost of recycling and allocating from a parent UM. The predominant cost in this case is the initialisation cost, which is constant in both cases.

7.1.6 Different Management Policies

There are number of policies and techniques a resource manager could apply when managing physical memory. seL4 mechanisms are powerful enough to enforce a variety of policies and common memory allocation techniques can be implemented reasonably easily and with varying degrees of efficiency.

An example system configuration is shown in Figure 7.2. In this figure running directly on top of seL4 is the initial resource manager. As I explained earlier, it is the responsibility of the initial resource manager to enforce a suitable, domain-specific policy among the different subsystems that constitute the system; the initial resource manager for instance, may chose to enforce a static partitioning policy over physical memory by enforcing the invariants we identified in Chapter 5 on the distribution of typed and UM capabilities. Or it may chose a dynamic *ballooning* [Wal02] algorithm to provide physical memory for the subsystem.

Each subsystem resource manager is *free* to implement a suitable policy over the resources it receives from the initial resource manager. However, the freedom a subsystem resource manager has in selecting a policy depends on the policy of the initial resource manager; for instance, if the policy enforced by the initial resource manager is dynamic then subsystem resource managers working above it cannot make strong availability guarantees. Therefore, a subsystem resource managers can only refine the policy to suit the needs to that particular subsystem.

In this manner, a seL4 system supports a hierarchy of resource managers, with each resource manager in the hierarchy refining the policy enforced by the top-level managers to suite the needs of the applications it manages.

Moreover, seL4 supports diverse concurrent policies via different resource managers implementing diverse policies on the UM memory objects under their control.

Conceptually, this hierarchy can be arbitrarily large — bounded by the maximum depth of the CDT tree which is conceptually, arbitrarily large. However, due to the limited storage space in CNode slots, the maximum depth in seL4::Pistachio is 128. This is obviously not a conceptual limitation, but an implementation decision to reduce to size of capability storage. Chapter 8 analyses the implementation and the performance of a system with hierarchical resource managers. In this prototype system the maximum depth reached was four. Thus, I believe 128 is not a serious limitation for any real system.

7.1.7 Limitations

All seL4 objects are fairly coarse grained and reasonably long-lived. As such, the performance overhead due to the allocation model is negligible. But, in particular, for systems

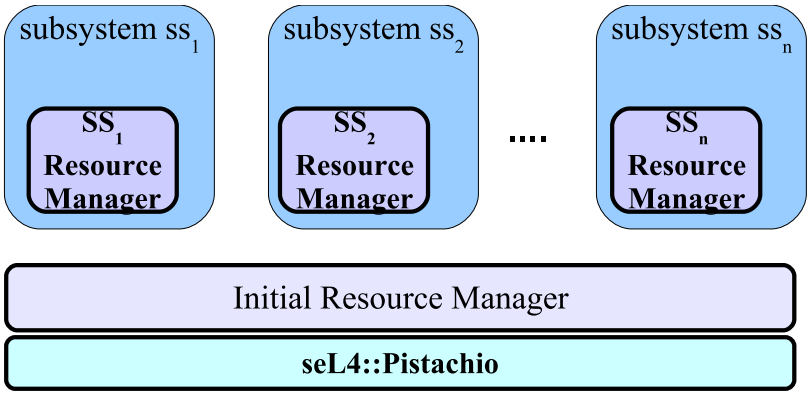


Figure 7.2: Example system with diverse resource management policies implemented by user-level resource managers.

that require allocation and de-allocation of memory at a rapid rate — for example, managing buffers on run-time for a giga-bit ethernet driver — the model will be inefficient. However, techniques such as ring-buffers [dBB08a, dBB08b] can be used make objects long-lived and thereby reduce the run-time allocations.

Moreover, the allocation scheme requires that all objects are statically sized — the memory requirement must be known at the time of creating the object. Implications of this requirement on kernel services is discussed in Section 4.6.

7.2 Cost of Capabilities

The seL4 memory management scheme and the protection model is based on kernel objects and capabilities that confer authority to obtain services from them. As such, every kernel operation, without exception, requires *at least* one capability for authority to obtain the service. Consequently, the capability *lookup* operation — locating the capability corresponding to user-provided CapIndex— has a significant impact on the performance.

In this section, I investigate the temporal overhead associated with locating a capability and the cost of maintaining a capability address space. The impact of these overheads on primitive kernel operations is analysed in the next section (see Section 7.3).

7.2.1 Layout of a CSpace

Before we turn our attention to performance numbers, an overview of the structure of CSpace is warranted. For a comprehensive discussion of the CSpace refer to Chapter 4.

A CSpace is a local, 32bit, logical address space of capabilities. The kernel itself does not provide an abstraction of a CSpace instead it provides the CNode object type. CSpaces are formed by connecting CNodes to one another forming a directed graph.

Each CNode decodes a part of the most significant bits of an address. In addition to translating bits via indexing, the decoding method also uses a *guarded-page-table* [LE96] like short-cut mechanism to reduce the resources (both temporal and spatial) required to decode all bits via many levels of CNodes. Each CNode capability, among other information, contains data required for the guarded shortcut — 16 bits of provision for a *guard* and 5 bits for *guard size*. A single CNode, therefore, can strip 16 bits of non-zero most significant bits from an address via the guard in addition to bits decoded via indexing. However,

in the case where the guard is zero, the kernel allow its size to be 31 bits (maximum number with 5 bits).

Recall that every CNode must at least translate one bit via indexing — this restriction guarantees the termination of translations even in the presence of loops in the CNode graph. Consequently, if need be, even the smallest CNode is sufficient to decode the whole 32-bit address — 31 bits from the guard and 1 bit by indexing. This layout allows for an efficient representation of small address spaces.

The address translation terminates after decoding *at most* 32 bits. The process can also terminate early successfully if a slot identified by indexing contains a non-CNode capability (or cause an exception if the slot is empty). The translations that terminate after decoding 32 bits, I term *bit terminated* and the ones that terminate early *object terminated*.

The selection of a guarded page-table structure to implement the CSpace is an implementation choice rather than a requirement of the model. The selection is motivated by the attractive properties of guarded page-tables over multilevel page-tables — they can efficiently map even a sparsely populated address space [Elp99], and provides flexibility in selecting a suitable CSpace layout which can be adjusted dynamically when the load changes. For example, a client application in a client server setup that has only a few capabilities in its address space (about 2 to 16 in practise), can use a single CNode as its CSpace stripping off the unused address bits using the guard. A server on the other hand, given the large number of capabilities it manages, would prefer a two-level tree. If the need arises, the server can add more levels to the tree, reduce the guard in top-level CNode and thereby expand the address space on demand. Still, if there is an object the server needs fast access to, then it can leverage the characteristics of the translation mechanism — it can copy that capability to the top-most CNode and setup a single-level, object-terminated lookup.

7.2.2 Capability Lookup Cost

Given its importance for the overall performance of the system, the capability lookup is implemented in hand-optimised assembly code. Moreover, the data structure implementing a capability; in particular the layout and the contents of the CNode-capability were hand-crafted to minimise the work that must be done during the lookup. These data structure optimisations and the lookup algorithm, which is based on the algorithm in [LE96], is ARM-specific — they are based on the instructions provided by ARM, but opaque external to the kernel.

The measurements are obtained by instrumenting the capability lookup code within the kernel. The cycle counter is reset at the start of the lookup and the value is read once the lookup terminates. An additional measurement is taken by resetting the cycle counter and reading its value immediately afterwards to factor out the cost of resetting and reading the counter itself. The difference of the two measurements is taken as the cost of a lookup and the reported numbers are the average over 1000 runs. For cold-cache measurements, the corresponding cache was flushed just before starting the measurement.

The number of machine cycles required to locate a capability for different CSpace configurations and different termination modes is shown in Table 7.4. The first column of the table shows the number of CNodes (*levels*) visited before the lookup terminates and the mode of termination. The second column is the number of cycles required in the best case — when both the data and instruction caches are hot. The last two columns shows the number of cycles required when the data cache is cold and when both data and instruction caches are cold, respectively.

Levels	Termination method	Hot caches	Cold D-cache	Cold I & D caches
1	bit terminated	19	61	100
1	object terminated	25	64	119
2	bit terminated	48	100	141
2	object terminated	54	109	161
3	bit terminated	72	139	180
3	object terminated	79	148	200

Table 7.4: Performance of capability lookups

The difference in lookup cost for different termination methods stems from the implementation. The current algorithm is biased towards single-level, bit-terminated lookups.

In the best case — bit-terminated lookup in a single level tree — the overhead incurred due to indirection through capabilities is negligible. From a system construction point of view, a single level CSpace with 2 to 16 slots in the node is sufficient for most applications; in particular for clients in a classical client-server based architecture. Since every CNode can be made to bit-terminate by adjusting the guard, a client application can generally expect to have best case performance from the above table — a 19-cycle overhead in the system-call path. For severs, on the other hand, a single-level tree is insufficient and in general would use a two-level tree. This is mainly because they manage a large, sparsely populated address space that needs to grow and shrink on demand. The cost of this flexibility is the modest overhead introduced by the two-level lookup — an overhead of 48 cycles in the best case is added to the system call.

In a classical client-server architecture, servers (in particular the ones that would require a two-level tree) generally perform heavy-weight operations. Clients usually send a light-weight request (via IPC) and the actual operation is done in the server context. These heavy weight operations amortise the modest overheads servers experience.

However, the best case numbers depend on the behaviour of the cache. Figure 7.3 shows the cost of a single-level, bit-terminated lookup under a realistic system load. These measurements are taken from a system running a para-virtualised Linux kernel (and few other system servers) on top of seL4::Pistachio. The Linux kernel is running the *lmbench* [MS96] benchmarking suite and the measurements are obtained by instrumenting the lookup code in the seL4::Pistachio kernel. Lmbench exercises various OS services of the Linux kernel. In providing these different services, the Linux kernel accesses different regions of its capability address space and therefore stresses the data cache.

The X axis of the figure is the number of machine cycles and the Y axis shows the cumulative percentage. The results of this experiment show that 52% of the time, the cost was 19 cycles and 72% of the time the lookup terminated with 24 (26% deviation from the best case) cycles or less. I believe this is an acceptable overhead for most kernel operations.

A server managing a large, sparsely populated CSpace that needs to grow and shrink on demand will prefer a two-level tree over a single-level one. For a such a server, based on the results of Table 7.4, the best case is a 48-cycle overhead in the system call path. To investigate how this overhead varies under a realistic workload, I repeat the above experiment for a bit-terminated, two-level lookup. The results of this experiment is shown in Figure 7.4. In this figure, the X axis shows the number of machine cycles taken from the lookup, and the Y axis gives the cumulative percentage.

However, unlike a single-level tree, the results do not show the best case under a real-

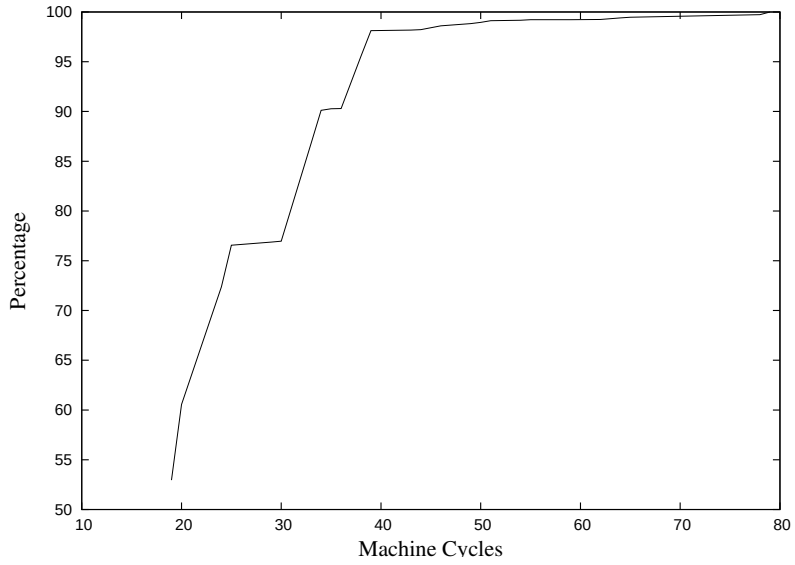


Figure 7.3: The variation of capability lookup cost for a single-level CSpace under a realistic workload. The *X* axis shows the number of machine cycles and the *Y* axis shows the cumulative percentage.

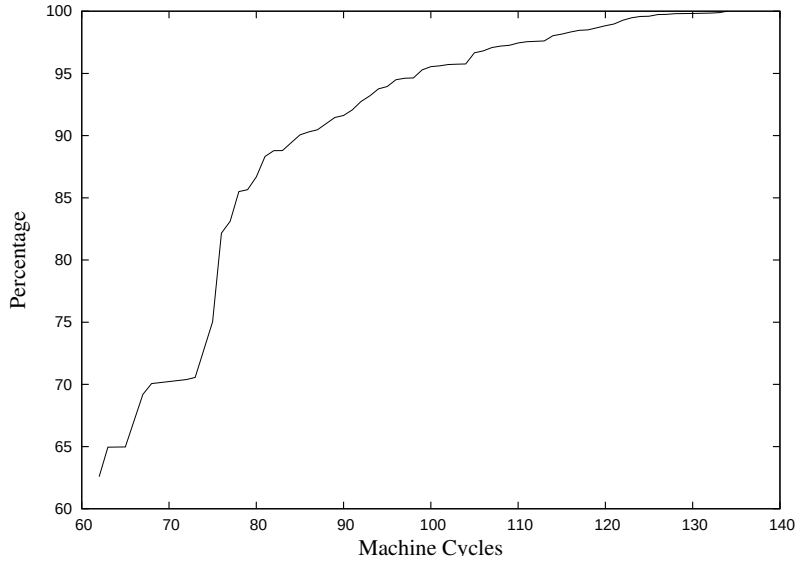


Figure 7.4: The variation of capability lookup cost for a two-level CSpace under a realistic workload. The *X* axis shows the number of machine cycles and the *Y* axis is the cumulative percentage.

istic load. Under the workload considered, the best case was 62 machine cycles and with 75% of lookups terminating under 75 cycles (up to 56% deviation from the best case).

This overhead is not a major drawback for two reasons. First, capability lookup cost is not a *complete* overhead. In capability-based access control a capability with the appropriate rights is sufficient authority to perform an operation — once the capability is located and tested for the correct permissions (usually an *AND* operation), the operation can proceed without any further checks. Because of this, the cycles spent in locating a capability, or at least some of it, are gained elsewhere. This however, is only true when comparing to a kernel which enforces some sort of control over who can perform what operation. Most kernels enforce some control of this sort, and if they do not, this is essentially the price for security.

Secondly, as mentioned earlier, servers, in particular the ones that would benefit from a two-level CSpace tree, have a tendency to perform heavy-weight operations. Examples of such operations include forking threads, paging clients, file system management and so on. These heavy-weight operations amortise the cost they incur due to the two-level tree. I further substantiate this claim in Section 7.3 by investigating the effects these overheads have on the primitive operations of the kernel and in Chapter 8 by examining the performance of a para-virtualised Linux server.

If the overhead is not acceptable to the server, then it can trade the flexibility offered by a two-level tree in terms of the ability to grow and shrink on demand with the better performance of a single-level tree. Furthermore, if a server has only a few performance critical objects that require fast lookups, these capabilities can be copied to the top-most CNode of the CSpace, causing an object termination at the first level and still have the flexibility of a two-level tree.

7.2.3 Managing a CSpace

The management of a CSpace is performed by invoking a CNode capability. The possession of a capability to the root CNode of a CSpace allows the management of all CNodes that constitute a 32-bit address space rooted at that CNode.

To invoke a CNode capability, the CNode capability itself must be mapped at a *bit*-terminated slot in the manager's CSpace — the ability to manage a CSpace is not inherent, but explicit.

Broadly speaking, the kernel provides operations to add and remove capabilities. There are two main operations for adding capabilities — imitate and mint — and two operations for removing — delete and revoke. I already discussed these operations in Chapter 4. To recall: both mint and imitate create copies of existing capabilities — imitate creates a CDT sibling with equal rights and mint creates a CDT child with lesser access rights. Delete removes a single capability from the system while revoke deletes all the CDT children of a given capability. The operations that adds capabilities to a CNode requires (at least) *Grant* right in the CNode capability.

When creating a new client application, a server will allocate a single CNode to use as the CSpace of the client. Then, the server will populate the client's CSpace with the required authority for the client to run. In most cases the server will delegate part of the authority it possesses over an object to the client. Partial delegations are facilitated by the mint operation —- the server can mint a copy of the capability it possess (in its two-level tree) to the client's single-level CSpace. If the resource is exclusive to a client, the server uses the imitate operation rather than mint. If a server wants to stop a particular service

Operation	Cycles	Source tree	Destination tree
mint	518	2-level	1-level
imitate	518	2-level	1-level
delete	438	2-level	NA
revoke	510	2-level	NA
New CSpace	1794	2-level	1-level

Table 7.5: Performance of CSpace management operations

then it can use the revoke operation to remove all capabilities that it handed out to clients to obtain that service. The delete operation, on the other hand, removes a single capability, and therefore stops a single client from obtaining the service. In this section, I investigate the performance of these CSpace operations assuming a classical client-server setup.

Table 7.5 shows the performance of CSpace management operations. The numbers shown in the table are averaged over 500 iterations and all the distributions has a coefficient of variation less than 2%.

The first set of results in this table show the cost of minting and imitating a capability from a two-level CSpace tree (typically the server) to that of a single-level (the client).

The next set of results show the cost of deleting and revoking capabilities from a two-level tree. Note that in both cases, the capability removed, is not the last capability to the object; as such on object destruction occurs. Unlike the delete operation, revoke removes the set of child capabilities. So, its performance depends on the number of descendants. The number shown in the table is the cost of revoking a single child and for each additional child the cost increases linearly by 180 cycles per child.

The final set of results in Table 7.5 shows the cost of creating a new, single-level CSpace tree and populating it with two existing capabilities. The new CSpace contains 16 capability slots, out of which only the first two slots are populated. The number shown above is the total cost; including the cost of creating CNodes via retype and the cost of minting two capabilities. The created CSpace corresponds to that of a typical client. In a typical application scenario, a client starts its execution with only two capabilities mapped to a single-level tree—allowing it to send and receive messages to a sever. The rest of the address space is populated on demand by the server based on messages from the client.

The size and the number of mappings required in a CSpace are highly context dependent. In one extreme, some applications will not require a CSpace at all; illustrative example is clients of a para-virtualised guest operating system. The clients themselves have no knowledge of and therefore do not make any seL4 system calls. When these clients trap into kernel with a system call number which is not handled by seL4, the kernel generates and sends an *exception IPC* to the exception handling endpoint registered within the TCB. Besides the authority to send exception IPCs and receive replies — both of which is conferred via capabilities registered in the TCB, these clients exercise no seL4 authority. Thus, their CSpace is empty. On the other extreme, native seL4 applications may require (depending on the type of services they provide), a large number of capabilities.

Since L4 does not have a notion of a capability address space, it is not possible to directly compare these numbers. Conceptually, mint and imitate are akin to the *map* operation of memory pages, which costs 939 cycles on L4.

Operation	seL4::Pistachio	L4	Description
IPC	202	217	IPC between two clients
IPC	232	217	IPC between client and sever
IPC	260	217	IPC between two servers

Table 7.6: Cost of IPC operation.

7.3 Performance of Kernel Primitives

To examine the performance impact of the use of capabilities in the system call path, I measure the performance of commonly used kernel *operations* in seL4, with different CSpace configurations and compare the numbers with those obtained for L4. Some L4 system calls do not have a directly related counterpart in seL4. There are two reasons for this divergence. Firstly, seL4 exports all its in-kernel memory management decisions to user-level resource managers; which requires more interactions with the kernel as opposed to L4 where the kernel manages its own memory. Secondly, seL4 introduces capabilities and a capability address spaces (CSpace) which requires additional management, which is not present in L4. As such, I compare the cost of performing similar operations, rather than system calls.

All numbers reported are obtained for the KZM-ARM11 development board using the CPU cycle counter with a hot cache. The ARM the cycle counter, however, is not accessible from user-level. Thus, to obtain cycle counts, I introduced two additional system calls to both kernels — one to *reset* the cycle counter and the other to stop and *read* its value. When taking a measurements, the benchmarking code first resets the counter and reads its value immediately afterwards. Then it resets the counter, performs the benchmark and reads the value. The difference between the two values is taken as the cost of the operations. For reasons I discuss below, numbers reported are the average over 1024 runs. In the case of seL4, the kernel objects are managed by the user-level kernel resource manager introduced in Section 7.1.2. As mentioned earlier, this resource manager uses slab allocation to mange seL4 kernel objects. At the start of each benchmark, all kernel object pools required to run the benchmark are empty — in other words, objects required for the benchmark are allocated during the benchmark, and therefore the allocation cost is included in the measurement. When a particular object is requested, resource manager allocates, depending on the object type, some two-to-the-power number of objects and populates the corresponding pool. If a pool is not empty all requests are satisfied from the objects in this pool and new objects are allocated only when there is a request while the pool is empty. By taking the average over 1024 runs I guarantee that all objects allocated during the benchmark are used.

Similar to the previous analysis, I envisage a two-level CSpace tree in a server context and a single-level tree for a client.

In the following subsections, I investigate the performance of three important subsystems of the seL4 kernel. Section 7.3.1 examines the performance of the IPC subsystem. Thread-related operations and their performance is investigated in Section 7.3.2. Finally, Section 7.3.3 analyses the performance of the virtual memory management interface.

7.3.1 IPC operation

Table 7.6 compares the cost of an IPC message of similar size under different CSpace configurations. The shown measurements are averaged over 1000 iterations and the maximum coefficient of variation for any measurement is less than 1%.

Both the kernels use hand-optimised assembly code to deliver the message and the entire message fits in CPU registers.

The difference between the three IPC numbers shown in the table is based on the CSpace configuration; both the IPC partners in the first set are clients using a single-level CSpace tree, in the second set one of the partners is a server with a two-level CSpace tree and the other a client and the final set is the cost of communication between two servers.

There is no significant difference between the cost of seL4 IPC and that of L4. In fact, if both IPC partners are clients, then seL4 outperforms L4 by 7%. Recall that if need be, seL4 design allows any CSpace to be a single-level; servers use two-level trees for convenience rather than a necessity. Even if a server uses a two-level tree, when communicating with a client that resides in a single-level CSpace the overhead is just 7%; which is acceptable in most cases.

The cost of communication between two servers however, is relatively high — 20% overhead compared with L4.

7.3.2 Thread Operations

The results in Table 7.7 shows the cost of thread-related operations. The measurements shown here are averaged over 500 iterations and the maximum coefficient of variation for any measurement is less than 1%. Unlike the IPC experiment, where I used different CSpace configurations, all seL4 numbers are obtained in a server context. The first set of numbers in this table compares the cost of thread operations. This set of operations requires no allocation of kernel objects. The second set compares the cost of creating new execution units and requires allocation of kernel objects via the retype method.

The *exchange register* system call sets a thread's CPU registers. The *set priority* system call allows one to modify the priority of a thread. In these two experiments, the two systems show no significant performance difference. In the case of exchange registers, the slight variation of the two numbers stems mainly from the fact that seL4 provides access to more registers than L4.

The second set of numbers inTable 7.7 shows the cost of creating a "unit of execution". By unit of execution I mean either a thread or a process. The first number in this set is the cost of creating a thread in the same process — within the same address space, or spaces in the case of seL4. A thread in seL4 has two address spaces associated with it — a CSpace for capabilities and a VSpace for virtual memory access. In contrast, L4 thread has only one address space for virtual memory access. To avoid any confusion, note that newer versions of OKL4 kernels have a capability address space associated with a thread, but this is not the case for the one used for benchmarking. The second number is the cost of creating a separate, single threaded process — a single thread in a different address space(s). Before examining the numbers I explain the kernel primitives required to complete the operations.

A privileged thread in L4 can create another thread using the *ThreadControl* system call or another address space using the *SpaceControl* system call. The ThreadControl system call provides an interface to both, allocating and configuring a thread to run in some address space. One of the arguments (the *spaceID*) of ThreadControl specifies the address space in which the newly created thread should reside. The spaceID provided to ThreadControl

must be to an existing address space. When the address space is already in existence a single system call (ThreadControl) is sufficient to create another L4 thread. To create a new process on the other hand, the privileged server needs two interactions with the kernel — first to creates a new address space using SpaceControl and then another to create a thread within that address space using ThreadControl.

Note that both operations, ThreadControl and SpaceControl, are privileged, in that only a privileged thread is authorised to perform them. Other system servers in L4 need to obtain these services from the privileged server. The L4 numbers shown in the table are obtained within the context of the privileged thread.

In contrast, creating a seL4 thread, even in the same address space, requires more interactions with the kernel. These additional interactions stem for the explicit kernel memory management of seL4. Since kernel memory is explicitly allocated, the server needs additional interactions with the kernel to allocate the TCB objects using the retype method before configuring them to run in some address space. The number of additional interactions depends on the allocation technique used by the user-level server creating these threads. L4 uses an in-kernel slab-allocator to manage sub-page sized objects such as TCBs. Recall that the prototype allocator used for these experiments also uses slab allocation, making the comparison fair. The seL4 server requests a TCB object from this allocator and then configures it to execute in the same address space as the server's, using the seL4 *ThreadControl* primitive. The seL4 number is the total cost of the operation, including the time spent for allocation.

The first measurement in the third set of numbers in Table 7.7 compares the cost of creating a thread in the same address space. For this operation, the two kernels show no significant difference. Even though seL4 warrants more interactions with the kernel due to its explicit memory management scheme, the performance of the operation is roughly the same as L4's.

The last measurement in Table 7.7 compares the cost of creating a process. First, note that this operation is expensive on both systems. This high cost is fundamental — it stems from zeroing the relatively large (16KB) first-level page directory. Second, note that seL4 process creation is more expensive than that of L4. The main reason for this difference is the capability address space — to create a seL4 process, one needs to create a separate CSpace in addition to the VSpace. A L4 process, in contrast, has only one address space for virtual memory. Each seL4 process created in this experiment has access to a single-level CSpace implemented using a CNode with 16 slots. As I mentioned earlier, 16 capabilities are in general sufficient for most client applications. Because of the need to create an additional address space, the process creation in seL4 is more expensive.

These experiments show no evidence of significant performance degradations due to exporting the decisions on in-kernel memory allocations to user-level servers. Moreover, the capability mechanism used by the protection system does not add any significant, direct overheads to the system call path. However, the introduction of the capability address space (CSpace) introduces more management overhead since there are two address spaces to manage.

7.3.3 Performance of VSpace Operations

To measure the temporal efficiency of the virtual memory (VM) interface, I ran a set of micro benchmarks on the seL4::Pistachio kernel, using a two-level CSpace tree. As mentioned, a typical seL4 server will use a two-level tree to implement its CSpace. The seL4

Operation	seL4::Pistachio	L4	Description
Exchange registers	385	354	set the register set of a thread
Set priority	323	311	change the priority of a thread
Thread creation	2671	2720	create a thread in same address space
Process creation	30453	29097	create a process

Table 7.7: Performance of thread operations.

numbers for these macro-benchmarks are compared with the numbers for performing similar operations on L4.

A summary of the experiments to investigate the performance of the VM interface is given in Table 7.8. These benchmarks are selected based on the work presented by Apple and Li [AL91]. In this paper, authors investigate several algorithms that make use of user-level page-protection techniques and, importantly to our context, identify a set of kernel primitives required for implementing such techniques. The selection of the micro benchmarks for this experiment is derived based on these primitives, with few modifications to factor out the effects of other kernel operations and to highlight the strong and weak points of seL4's VM interface.

In [AL91], Apple and Li identified and measured the cost of the following kernel primitives:

TRAP: handle page-fault in user mode

PROT1: decrease the accessibility of a page

PROTN: decrease the accessibility of N pages

UNPROT: increase the accessibility of a page

DIRTY: return a list of dirtied pages since the previous call

MAP2: map the same frame (physical page) at two different virtual addresses at two different levels of protection in the same address space

The TRAP cost measures the cost of handling a page fault in user-mode. Both seL4 and L4 translate page faults into IPC massages (called *page-fault IPC*) and propagate them to a server registered with the thread to handle the exceptions. The server then handles the page fault by installing a mapping and replying back to the faulting thread with another IPC to make the thread blocked on the fault runnable. To factor out the cost of IPC, I break TRAP into two parts — *MAP* measures the cost of installing a mapping and *TRAP-IPC* measures the cost of the IPC operations.

As mentioned, the MAP operation of L4, is privileged — only a privileged server is allowed to perform it. Others need to request MAP operations (usually through another IPC) from the privileged server. The L4 measurements presented in this section are taken for a privileged server, and therefore without the proxy cost.

The interfaces provided by seL4 and L4 are different from one another, in that seL4 requires the resource manager to explicitly allocate and install second-level page tables (of the ARM two-level page-table structure) required for mapping the page. The seL4 kernel notifies the resource manager of the second-level page-table status in the page-fault IPC. In contrast, L4 allocates these page tables implicitly as and when the need arises. To investigate the effects of explicit management I introduce two variants to MAP — *MAP-NO-L2* is the cost of a mapping when the second-level page table required for installing it

Benchmark name	Description
MAP-VALID-L2-X	page mapping of size X, required page table is present.
MAP-NO-L2-X	page mapping of size X, required page table is not present.
TRAP-IPC	cost of exporting page-fault handling to user-level.
PROT1	decrease the accessibility of a page.
PROTN	decrease the accessibility of a N pages.
UNPROT	increase the accessibility of a page.

Table 7.8: Summary of the operations performed to investigate the performance of the VM management interface.

is not present, and *MAP-VALID-L2* is the cost of a mapping when the second-level page table required for installing it is already present.

There is a subtle difference between the two VM interfaces that needs investigation. This differences is not conceptual, rather a design decision motivated by various factors. The VM interface of L4 supports *fpage* mappings — an arbitrary 2^n-sized page. Though, it is conceptually possible within the proposed model, the current implementation of seL4::Pistachio only supports mappings that are of hardware-supported page sizes — a design choice made to simplify the VM interface and thereby reduce the effort required in verification. I investigate the impact of this design choice by measuring the cost of mapping memory objects of two different sizes. In the presentation, I distinguish them by adding a suffix to the MAP operation.

Both L4 and seL4 do not provide information about dirty pages and therefore I omit this benchmark from the experiment. One can track dirty pages by installing read-only mappings and observing the generated page faults. Moreover, MAP2 is no difference to MAP in both kernels and hence dropped from the benchmarks.

The performance of seL4 and L4 for the operations summarised in Table 7.8 are reported in Table 7.9. The first column of Table 7.9 is the benchmark and the final two columns give the performance of seL4 and L4 respectively.

The MAP-VALID-L2-X operations show the cost of inserting mappings for a memory object of different sizes (X) when the second-level page table required for the mapping is present. When mapping a 4KB memory object, seL4 has an advantage over L4. This improvement is mostly due to the simplified VM interface of seL4 compared to the fpage based mapping primitive of L4. However, the fpage scheme is efficient when mapping memory objects that are not of a hardware defined size — for example, when mapping a 8KB object L4 outperforms seL4 by 11%. In L4, one can map a 8KB fpage with a single system call (as long as the physical memory frames are also contiguous and size aligned). In seL4 on the other hand, the operation requires two system calls to install two 4KB objects. As I mentioned earlier, the simplified interface is not a direct result of the model but a design decision to keep the kernel interface simple. If need be, the seL4 VM interface can be modified to support fpage semantics.

The second result in Table 7.9 measures the cost of inserting a mapping of 4KB, but in contrast to the former, the second-level page table required for placing the mapping is not present. I explain how the two kernels handle this scenario. The L4 kernel implicitly allocates a page table from its heap, installs it and then proceeds to complete the mapping. The seL4 kernel, on the other hand, does not allocate any memory implicitly. The page-fault IPC message generated by seL4 contains information on whether or not a second-level

Benchmark	seL4 [cycles]	L4 [cycles]
MAP-VALID-L2-4KB	1109	1766
MAP-VALID-L2-8KB	2218	1966
MAP-NO-L2-4KB	4621	4370
TRAP-IPC	600	987
PROT1	1379	1816
PROTN	1379 * N	1816 * N
UNPROT	1379	1816

Table 7.9: Cost of VSpace operations.

page table is present to cover the region in which the page fault took place. If a page table is not present, the user-level handler needs to allocate a page table, using some allocation technique and then install the page table and the page mapping. The seL4 VM interface allows the installation of a page table and a page mapping through one interaction — a single seL4 system call is sufficient to install the page table and the page mapping. In these experiments, similar to the previous ones, kernel objects required for performing an operation is managed using the allocator described in Section 7.1.2.

When the second-level page table required for completing the mapping is not present, seL4 is marginally (about 5%) slower than L4.

The cost of TRAP-IPC is relatively high for L4. The page-fault IPC path of L4 is relatively unoptimised compared to that of seL4. Moreover, note that for both kernels, a page-fault IPC is more expensive than those IPC costs reported in Section 7.3. A page-fault IPC takes a different code path (implemented in C) to that taken by a normal IPC (implemented in assembly).

In both systems, PROT1 — reducing the privileges of a page — is implemented by installing a mapping over the existing. Thus, PORT1 shows similar characteristics to MAP-VALID-L2-4KB. Note that the second-level page table required when performing the operation is always present. PROT1 is slightly more expensive than a MAP operation because of having to flush the TLB for the previous mapping. Both kernels leverage the tagged TLB of ARM11 to optimise TLB flushes. Similar to PROT1, UNPROT is implemented by reinstalling a mapping and therefore shows similar characteristics.

In L4, the performance of PROTN benchmark (reduce the privileges of N pages) depends on the layout of the protected pages. If the pages are contiguous, a single fpage mapping would suffice but if they are not, then we need to perform PROT1 operation N times. For seL4, either case requires N number of PROT1 operations. The reported experiment is conducted assuming PROTN is protecting non-contiguous pages. It should be noted that, similar to the case of MAP of contiguous pages, L4 will have better performance owing to the reasons I described earlier.

As it can be seen from these numbers, there is no significant difference between the VM interfaces provided by the two kernels. In some cases, seL4 has a slight performance advantage over L4, mainly due to its simplified VM interface. This is not a direct consequent of the proposed memory management model, but a design decision motivated by other factors. Exporting the allocation of second-level page tables required for installing a mapping to user-land shows no significant impact on performance, as long as users are aware of the in-kernel resource status.

7.4 Summary

Based on our analysis thus far, the proposed memory allocation model and its associated capability protection model do not add significant overheads to the primitive kernel operations.

System call authorisation via capabilities does not cause undue overheads. However, the introduction of a capability address space introduces additional overhead to process creation, because of having to create (and maintain) another address space.

Exporting the management of all in-kernel memory to user-level resource managers increases the cost of some kernel operations marginally (around 5%).

However, it remains to be seen whether the model imposes any performance implications at a macro level. In the next chapter, I investigate the macro-level performance of the model.

Chapter 8

Performance Evaluation

Having investigated the formal security properties of the proposed scheme and its micro-level performance characteristics in previous chapters, I now turn to a practical application of the security theorem and further extend the performance analysis to a macro level using a complex, realistic resource manager.

This chapter investigates the macro-level performance of a realistic resource manager (a paravirtualised Linux kernel) running on seL4::Pistachio, restricted to a spatial partition using the results of the formal analysis in Chapter 6. The performance of this resource manager is compared to three other configurations of itself; (a) executing native on top of the hardware, (b) executing as a (virtualised) user-level resource manager without any restrictions on its kernel memory consumption, and (c) executing as a user-level resource manager with similar restrictions to spatial partitioning, but enforced by a privileged server monitoring the execution. The measurements obtained from the first system indicate the baseline cost of an operation. The second system represent the *raw* cost of virtualisation, i.e. without any security enforcement and the final system compares the seL4 costs against another commonly used security enforcement technique.

The formal security analysis in Chapter 6 demonstrates that the model is sufficient to enforce spatial partitioning (see Theorem 2). Theorem 2 confirms that by enforcing a set of restrictions on the initial distribution of capabilities, it can be guaranteed that a resource manager cannot gain access to any more kernel memory regions than what was conferred to it in the initial state, in any future state. In other words, the resource manager will be spatially partitioned and this partitioning will hold in all states derived from the initial. The restrictions that need to be enforced on the initial capability distribution to guarantee the partitioning of a resource manager is given in Section 6.4.2. I recap these restrictions in Section 8.1.

The resource manager selected for these experiments is a para-virtualised, Linux-2.6.23 kernel, running as a user-level application, managing the resources of native Linux applications. Hereafter, I call this port of the Linux kernel *seL4::Wombat* and the software configuration *seL4::Wombat system*. The implementation of seL4::Wombat is based on *Wombat* [LvSH05]. Section 8.1 examines the implementation of seL4::Wombat system and in particular discusses the implementation of seL4::Wombat and *Iwana* — a resource management library which acts as a glue layer between the Linux kernel and seL4.

For macro benchmarking, I use the *AIM9* [AIMa] and the lmbench [MS96] benchmarking suites. These benchmark suites exercise OS services such as memory management, IPC, file IO, signal delivery, float point handling, networking and so on.

To evaluate the performance of the seL4 model, I compare the performance numbers

of the above benchmarks with those obtained for three other similar system configurations — with the *L4/Iguana* system and with the *Wombat::Insecure* system, and with the *native* Linux kernel running on same hardware.

Both, L4/Iguana and Wombat::Insecure systems execute Linux-2.6.23 kernel as a de-privileged, para-virtualised user-level application. The two systems differ in that they provide different levels of security guarantees; the L4/Iguana system enforces a strict limit on the amount of kernel memory consumed by the Linux subsystem while Wombat::Insecure attempts no such control. Section 8.2 and Section 8.3 introduces the configuration of L4/Iguana and Wombat::Insecure respectively.

All experiments reported here were conducted on a *KZM-ARM11* evaluation board, which comprises an *ARM1136JF*-based [ARM05], (Freescale™ i.MX31) processor running at 532MHz, an unified L2 cache of 128KB and with 128MB DDR RAM.

8.1 seL4::Wombat

Figure 8.1 shows the software components that constitute the seL4::Wombat system. Running directly on top of seL4::Pistachio is the initial resource manager; responsible for bootstrapping the rest of the system components and enforcing the system-level policy between them.

The system consists of two main components; a timer server and a seL4::Wombat server. The seL4::Wombat component is a paravirtualised Linux kernel, running as a user-level application on top of seL4::Pistachio, providing the abstractions for Linux applications (which are not shown in the figure). The timer server delivers periodic ticks to seL4::Wombat.

In this context, there are two important considerations the initial resource manager needs to address: (a) can seL4 mechanisms be used to enforce the system-level policy? and (b) if so, what invariants should be followed while bootstrapping the system?. Both these questions should ideally be answered within a formal framework similar to that developed in Chapter 5 by doing a formal analysis like the one presented in Chapter 6.

For this experiment, the system-level policy the initial resource manager is to enforce is spatial partitioning [Rus99]. This policy guarantees that seL4::Wombat cannot access any region of physical memory beyond what was initially given to it by the initial resource manager.

Recall that in Chapter 6, I have done a formal analysis of the seL4 security model and affirmed that seL4 mechanisms are sufficient to enforce spatial partitioning. Moreover, the proof also identified a set of invariants the initial resource manager needs to enforce on the distribution of initial capabilities (see Section 6.4.2) for the enforcement. These invariants are as follows:

- There should not be any GrantIPC channel between any two threads in different spatial partitions (a GrantIPC channel is an IPC channel that allows the propagation of a capability),
- writable CNodes and writable page tables cannot be shared between threads in different spatial partitions, and
- a capability with *Grant* authority to the TCB of a thread cannot be placed in another thread's possession if the latter is in a different spatial partition.

In the seL4::Wombat system, the initial resource manager enforces spatial partitioning by enforcing the above invariants on the initial capability distributions while bootstrapping

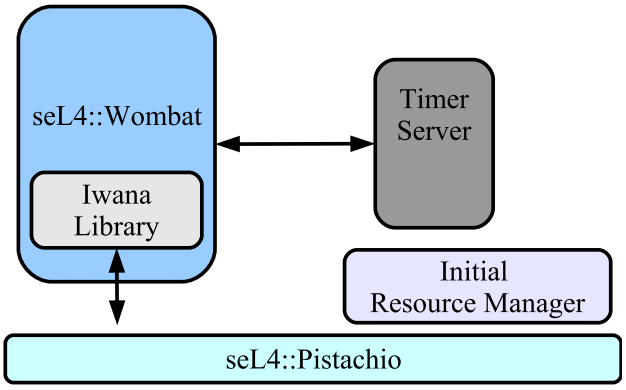


Figure 8.1: The seL4::Wombat system.

the two components. However, when enforcing the above invariants, the initial resource manager makes two simplifications: (a) it does not share CNodes or page tables between seL4::Wombat and the timer server, even though it is possible to share read only ones, and (b) it does not place a TCB capability of a thread in the others CSpace.

The timer server runs as separate process from seL4::Wombat; i.e. uses a separate CSpace and VSpace. Its CSpace contains capabilities to two endpoints; one for receiving timer requests from seL4::Wombat and the other to send timer ticks to seL4::Wombat via IPC, and two capabilities for receiving and acknowledging the timer interrupts generated by the hardware. Its VSpace contains the required mappings for accessing timer hardware, executing code and a special page called the *summary page*, mapped at a predefined location which contains a summary of the capabilities in the CSpace.

The timer server generates periodic ticks for seL4::Wombat. Optionally, one could incorporate the timer server into seL4::Wombat directly and let Linux manipulate the associated hardware directly. This option was dropped to keep the system configuration comparable to L4/Iguana, in which the timer server *may* provide ticks for more than one, mutually untrusted servers.

Compared to the timer server, the address space of seL4::Wombat is more complex. However, this complexity is managed by the Iwana library, and outside of the initial resource manager. The motivation here is to keep the initial resource manager; the one responsible for enforcing the policy, as simple as possible. Thus, if need be, it can be verified against the invariants we have identified.

The initial resource manager bootstraps seL4::Wombat in much the same way as the timer server. It starts by creating a new CSpace and a VSpace. Then populates each with the appropriate authority. Among other operations, seL4::Wombat is authorised to manipulate both of its own address spaces (CSpace and VSpace). When bootstrapping, the initial resource manager performs a minimal configuration — just enough for seL4::Wombat to start executing, and when it does, seL4::Wombat exercises its authority to manipulate its address spaces to configure them in a suitable manner. In addition to the authority over address spaces, seL4::Wombat is granted some number of Untyped memory objects (8MB in total) which it uses to allocate seL4 kernel objects as and when the need arises, two endpoint capabilities to communicate with and receive timer ticks from the timer server, a capability to its own VSpace allowing it to manipulate the VSpace, and few Frame capabilities to memory mapped IO regions depending on what devices it is allowed to access. Its

VSpace only contains mappings for executing the code and accessing the summary page.

Once these components are bootstrapped, the initial resource manager exits from the system and it is the responsibility of the subsystem resource managers to use the resources they received in a suitable manner.

8.1.1 Iwana— Best-Effort Allocator

Iwana is a simple, best-effort allocator that acts as a glue layer between seL4::Pistachio and Linux. It can be viewed as a library OS running in the same process (uses the same VSpace and CSpace) as seL4::Wombat. It provides two main services. Firstly, it is responsible for bootstrapping the Linux subsystem. Secondly, it manages all seL4 kernel objects required for the functionality of Linux.

The rationale behind using such a two-stage bootstrapping process is to reduce the complexity of the initial resource manager — the critical component in terms of enforcing isolation. By recalling the formal analysis in Chapter 6, note that the enforcement of isolation is dependent *only* on the behaviour of initial resource manager. This approach reduces the functionality of initial resource manager to a bare minimum — statically subdividing capabilities in its possession between the system components. The actual bootstrapping of the component, which has the potential to be complex, is kept outside of the critical component.

This reduction of complexity means there is a potential to formally verify the implementation correctness of the initial resource manager. While there is ongoing research on the above topic, it falls outside the scope of this thesis.

For its functionality, Iwana needs knowledge about its address-space configuration. As an illustrative example, it needs to know how much Untyped memory it has access to and where these capabilities are mapped in its CSpace, and so on. Some of this information is known by convention and the rest, in particular the amount of physical memory it can access and where Untyped capabilities are mapped, is passed to Iwana by the initial resource manager by storing this information in the summary page, mapped at a known address.

Initially, Iwana reads this information, updates its internal bookkeeping structures and then turns to bootstrapping the Linux kernel. Bootstrapping Linux includes the following steps: first, it creates a contiguous region of virtual memory, backed by (4K) Frame objects, which the Linux kernel perceives as its heap. The size and the starting address of this region is predefined. To create this region, Iwana creates the required number of CNodes (for allocating the Frames), Frame objects and second level page-tables required for mapping Frames in a VSpace. Then, it maps each Frame to its VSpace such that the virtual page number can be used to compute the address of the corresponding Frame capability (by adding an offset). When handling page faults, the Linux page-fault handler generates "physical" addresses within its heap — in the case of seL4::Wombat addresses within the above region. The above mapping scheme provides an efficient way of locating the corresponding Frame capability.

In addition to managing the Linux heap, Iwana provides access to memory mapped IO regions for Linux device drivers; provided it received the Frame capability to the IO region from the initial resource manager.

The architecture-dependent part of the Linux kernel is replaced with calls into Iwana, which implements these operations by managing appropriate seL4 kernel objects.

Object Management

The allocation strategy used by Iwana is slab allocation. Internally, Iwana keeps pools of different object types and when a particular pool is empty (except for free CNodes slots) it populates the pool by allocating a number of corresponding objects. However, the free-CNode-slots pool is never allowed to run dry.

CNode slots play a crucial role in object allocation— to allocate any object, including CNodes, Iwana needs free CNode slots (see Chapter 4 for more details). Thus, without free CNode slots, Iwana cannot allocate any object. To avoid this situation, Iwana uses a low-water mark for free CNode slots — the number of CNode slots is never allowed to fall below a certain level, provided there are Untyped objects to allocate them from. Moreover, CNode slots are given precedence over all other objects. Before allocating a batch of objects to populate a pool, Iwana checks whether this allocation would take the CNode slots below the above low water mark. If so, it allocates CNode slots, and only then proceeds to allocating the other object type.

The decision to use slab allocation is based on two main observations. First, the Linux kernel consumes more than one instance of an object type. Thus, allocating more than one object instance is not an undue over commitment of memory. Second, based on the micro-level measurements (see Section 7.1.1), allocating a batch of objects reduces the allocation cost per object.

When Linux no longer needs a particular resource, it returns the corresponding kernel object(s) to Iwana. Once an object is released, Iwana re-initialises the object by performing a *recycle* operation and returns it to the corresponding object pool.

Except for CNodes — that are allocated automatically when the number of free nodes falls below a certain level — Iwana allocates other kernel objects on a first-come-first-serve basis, provided it still has unused Untyped objects in its possession. Moreover, under normal operation, it never reclaims objects forcibly from Linux — an object is recycled and used in a different context only when Linux releases it.

However, when Iwana is under heavy memory pressure — that is when it has used all its Untyped memory objects — it resorts to forcibly deleting already allocated objects and reusing that memory for new objects in a systematic manner. Before introducing the memory reusing strategy, I explain the motivation behind it.

Most servers, in particular servers such as Linux, internally track sufficient information to reconstruct the configuration of the underlying kernel objects that implement the abstraction they provide. For example, the Wombat server internally maintains shadow page tables for the address spaces it manages. These tables, together with the bookkeeping within Iwana (information on how heap address relate to Frame capabilities) is sufficient to reconstruct the corresponding seL4 page tables. I call such objects *cache-able*.

Though it is inefficient — because of the potentially large amount of work required to reconstruct the object state once the original is thrown away — the state of a cache-able kernel objects can be thrown away at any point, and reconstructed from the user-level state. Iwana leverages this observation — when it is under heavy memory pressure, Iwana reclaims memory from cache-able objects and reuses it to implement the requested object type (which again can be a cache-able type). For implementing this algorithm, Iwana tracks, in a first-in-first-out (FIFO) structure, the Untyped capabilities used for allocating cache-able kernel objects. I call these Untyped capabilities *reclaimable*. When faced with memory pressure it removes a capability from the reclaimable set, calls *revoke* on that parent Untyped capability and thereby deletes all the cache-able objects allocated in that memory region and then reuses the parent Untyped object to implement the newly

requested object type.

In the current implementation, Iwana treats only the PageTable as cache-able. This is because in comparison, PageTables are relatively easy to reconstruct. The motivation for using a FIFO structure for tracking reclaimable Untyped objects is to introduce some form of fairness — every cache-able object remains in memory for some period of time before the memory is reclaimed. Depending on the context, one can employ a different structure.

The idea of cache-able objects is similar but not identical to the kernel memory management model employed in systems such as the *Cache kernel* [CD94], *EROS* [SSF99] and *HiStar* system [ZBWKM06]. Iwana's allocation policy is dynamic: it switches from being a static allocator to a cache like scheme only under memory pressure. If sufficient memory resources are available objects are not reclaimed until they are no longer required, thus improving the performance. Under heavy memory pressure, it switches to the cache-like scheme where the performance is suboptimal because of the potentially large cost in reconstructing the object, and unpredictable because any cache-able object can be deleted at an unknown time.

When both the initial Untyped object pool and the reclaimable pool are empty, Iwana denies all subsequent requests which requires memory allocation.

Note that Iwana is only one example of a best-effort allocator. If need be, a different allocator can replace Iwana, depending on the requirements of the system it caters for, it can even be a static allocator similar to the initial resource manager. A seL4 system can support a stack of such allocators, with each allocator receiving resource from the one below and catering the requirements of the system it manages. The system supports diverse, co-existing policies by means of different allocators. It can support hierarchical policies by stacking these allocators — for instance, seL4::Wombat system is such a hierarchical approach. The initial resource manager uses a static allocation scheme and Iwana, using the resources it receives from the initial resource manager, supports a best-effort allocation scheme. I believe these mechanisms are powerful enough to cater the resources management requirements of most systems.

Benchmarking setup

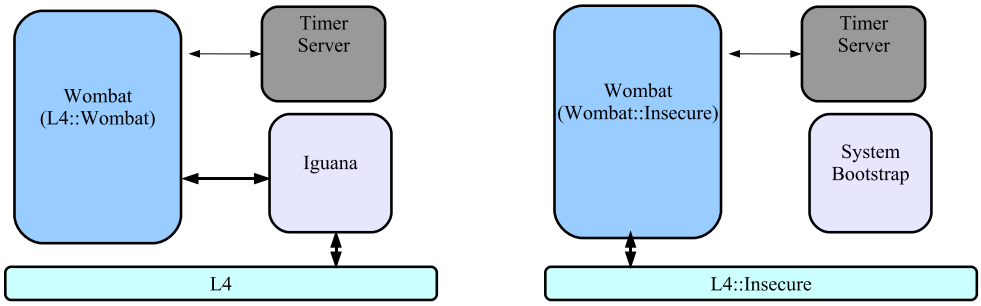
The results of the benchmarks reported below are measured when Iwana is operating under low memory pressure. That is to say that measurements were taken before the allocation policy of Iwana changes to the caching scheme. This is done to make the comparison between the seL4::Wombat and L4::Wombat comparable.

8.2 L4/Iguana System

Similar to seL4::Wombat system, L4/Iguana system is a microkernel-based system. The underlying microkernel in the case of L4/Iguana is the OK Labs L4 kernel (version *2.1-dev.35*) which is based on *L4-embedded* [NIC05].

Part (a) of Figure 8.2 shows the software configuration of L4/Iguana. Iguana [ERT] is a small, capability-based operating system that runs in user-mode on top of L4. In conjunction with L4, it provides and enforces the underlying resource management policy for *Wombat* [LvSH05] — a para-virtualised, port of Linux 2.6.23 kernel. Hereafter, I call the Linux version running on top of L4/Iguana *L4::Wombat*.

The authority to obtain kernel services that require the allocation of kernel metadata is centralised in Iguana — any kernel service that may require the allocation of kernel meta-



a) L4::Wombat System b) Wombat::Insecure System

Figure 8.2: Wombat system configurations.

data must be made through, and therefore is monitored by, Iguana. Through this mechanism, Iguana enforces a strict *limit* over the physical memory consumption of L4::Wombat.

In addition to Iguana and L4::Wombat, there are few other servers in the system providing different services to L4::Wombat. The configuration and the functionality of these servers is similar in both systems.

8.2.1 Spatial Partitioning

The above two systems differ in that they use different mechanisms to enforce the policy. In L4/Iguana, the policy is enforced by centralising the authority to perform sensitive operations (in our case operations that may require allocation of physical memory within the kernel) in Iguana.

In this regards, the functionality of Iguana is conceptually analogous to that of *system domain* in *Nemesis* [Han99], *Domain0* in *Xen* [BDF+03] or the *Supervisory VM* in *Denali* [WSG02]. In all the above systems, the authority to perform sensitive operations is centralised in the particular "trusted" application and as such, others are required to proxy all such requests to the "trusted" application which monitors and enforces the policy.

The main advantage of this scheme is that the kernel memory management policy is implement at user-level. Thus, the policy can be modified by modifying the user-level monitor, as opposed to the kernel. For example, one can change the system-level policy from spatial partitioning to a best-effort, first-come-first-server policy by modifying the monitor, as opposed to the kernel. As mentioned in Chapter 3, kernel modifications are undesirable for a formally verified kernel. The main disadvantages in this approach is that there is a temporal overhead incurred due to indirection through the "trusted" application.

In contrast, the seL4::Wombat system enforces the policy by controlled delegation of authority — seL4::Wombat can directly use the UM capabilities in its possession as it sees fit. Moreover, by enforcing the invariants identified by the proof at the time of bootstrapping, we can guarantee that seL4::Wombat cannot use any region of physical memory other than what is explicitly authorised by the initial resource manager . This approach eliminates the proxy cost and the need for run time checking and dynamic bookkeeping required in the previous case for enforcing the policy.

Further note that in comparison to L4/Iguana, seL4::Wombat system provides additional control over the physical memory consumption of seL4::Wombat — by distributing UM capabilities appropriately we can not only limit the amount of physical memory consumed by the Linux subsystem, but also the regions of physical memory it may consume. The Linux subsystem can only access a memory regions explicitly allocated to it; thus en-

Benchmark Category	Lmbench	AIM9
Resource management	all reported	all reported
Communication	reported	reported
File system	reported	reported
CPU performance	not reported	not reported

Table 8.1: Summary of the sub-benchmarks included in the results.

forcing spatial partitioning. On the other hand, the L4/Iguana system, cannot control the location of allocated kernel memory — the kernel implicitly selects the memory location for an allocation within its heap. As such, L4/Iguana can only limit the amount of memory consumed by the Linux subsystem. The ability to control the region of memory where the object will reside is desirable for a system attempting to enforce a cache colouring scheme to improve real-time predictability [LHH97] or control the memory placement in NUMA machines [MPO08, VDGR96, AJR06] or multi-core machine [CJ06, ZDS09] to improve performance.

This analysis only consider a spatial partitioning policy. However the results can be generalised to other useful policies.

8.3 Wombat::Insecure system

The software configuration of Wombat::Insecure system is shown in part (b) of Figure 8.2. Similar to our previous two systems, the Wombat::Insecure system is a microkernel-based system running the Linux 2.6.23 kernel as a para-virtualised user-level application. Hereafter, I call this port of the Linux kernel Wombat::Insecure.

In contrast to the previous systems, Wombat::Insecure system does not attempt to control the physical memory consumption of the Wombat::Insecure. As and when the need arises, Wombat::Insecure directly calls the underlying microkernel which simply grants the request if the microkernel's heap is not already exhausted — making it trivial for Wombat::Insecure to launch a denial-of-service attack against the other applications running on the system. This system represents the base cost of virtualisation.

The L4::Insecure is a modified version of the same L4 kernel used in the L4/Iguana system. The only difference between the two kernels is that L4::Insecure allows any application to perform operations that may require allocation of kernel memory in contrast to L4 where it is centralised.

8.4 Results

This section reports the results of *lmbench* [MS96], and *AIM9* [AIMb] benchmark suites on the four Linux configurations — seL4::Wombat, L4::Wombat, Wombat::Insecure and native Linux.

The AIM9 benchmark was used in the *singleuser* configuration and with each test configured to run for 2S. The benchmark consists of 58 sub-tests that exercise OS services such as the IPC mechanism, file IO networking and hardware operations such as function calls, integer/float-point operation. Similar to AIM9, lmbench also contains a number of tests that measures the performance of different OS subsystems as well as CPU operations.

As shown in Table 8.1 the tests included in the two benchmarking suites can be categorised into four main groups. Results of all the tests that require allocation of memory within the kernel (be it Linux kernel or the underlying microkernel) are reported. Only a representative set of tests, that examines the communication subsystem and file system are reported. The tests that measure the performance of CPU operations (such as float division, addition, multiplication) are not relevant to our context and therefore omitted from the results reported below.

To avoid any IO latencies not imposed by the OS, all benchmarks was run inside a RAM disk. All experiments reported here were conducted on the KZM-ARM11 evaluation board, which comprises an ARM1136JF-based, (Freescale^{TM} i.MX31) processor running at 532MHz, with an unified L2 cache of 128KB and with 128MB DDR RAM. All software was compiled with a gcc 3.4.4 cross compiler.

8.4.1 Lmbench Results

Lmbench system latency and bandwidth results are shown in Table 8.2. In this table, the columns named *seL4*, *Secure*, *Insecure* and *Native* show the performance of seL4::Wombat, L4::Wombat, Wombat::Insecure and native Linux kernel, respectively. The last two columns show the relative performance gain of seL4::Wombat with respect to Wombat::Insecure (*Insecure*) and L4::Wombat (*Secure*) respectively. The first part of the table shows the latency of various tests in the benchmark suite and the second part shows the memory bandwidth of various lmbench tests.

Of these systems, seL4::Wombat and L4::Wombat enforce a precise control over the amount of memory consumed by the para-virtualised Linux kernel; in the former case by controlling the initial distribution of authority and in the latter by using a privileged server that monitors system calls proxy through it. The third system, Wombat::Insecure does not attempt any such control — it characterises the baseline cost of running a para-virtualised kernel on top of L4, without any attempt to control the resource consumption of the guest kernel. Finally, the native Linux numbers characterise the penalty associated with virtualisation — how does virtualisation effects the system performance. Below, I analyse the performance of the three virtualisation platforms, and latter comment on the cost associated with virtualisation.

The first set of results in *latency* shows the latency of operations that require allocation of physical memory within the kernel. These numbers show the performance benefits of the proposed memory management scheme — seL4 improves the performance of these tests by *at least* 25% compared to L4::Wombat which enforces a similar (but even less restrictive) policy.

By decentralising kernel memory management, seL4 eliminates the need for Linux to proxy these requests through the privileged Iguana server to the underlying microkernel in order to enforce the policy over the physical memory consumption of Linux subsystem. This improves performance in two ways. First, it eliminates two IPC messages and hence two context switches required for sending the request to Iguana and the reply message from Iguana. Second, it eliminates the need for maintaining bookkeeping information within Iguana required for making an informed decision. When Linux makes a request, Iguana needs to make a decision on whether or not to allow it. In order to make this decision, it needs to maintain bookkeeping information on the current resource allocation state of Linux. Removing the need for mediating microkernel system calls in order to enforce the policy has a significant performance advantage.

	seL4	Secure	Insecure	Native	Gain (Insecure)	Gain (Secure)
Latency	$[\mu s]$	$[\mu s]$	$[\mu s]$	$[\mu s]$	%	%
pagefault	10.88	16.71	13.98	6.03	22.2	34.9
fork	1225	1753	1248	700	1.8	30.1
exec	1361	1961	1396	897	2.5	30.6
shell	10810	14516	11211	8993	3.6	25.5
null (syscall)	2.09	2.29	2.31	0.5	9.5	8.7
open (syscall)	16.25	17.01	16.78	12.71	3.2	4.5
fifo	26.1	30.54	30.12	14.71	13.3	14.1
pipe	27.58	30.73	30.74	23.74	10.3	10.3
signal (install)	4.39	4.34	4.34	2.21	-0.5	-1.2
signal (catch)	6.56	6.54	6.66	4.59	1.5	-0.3
semaphore	6.68	6.86	6.96	4.98	4	2.6
tcp	5.02	5.15	5.15	3.04	2.6	2.6
unix	43.23	45.03	45.04	27.67	4	4
connect	109.9	119.3	121.3	96.5	9.4	7.9
Bandwidth	$[MB/s]$	$[MB/s]$	$[MB/s]$	$[MB/s]$	%	%
pipe	124.4	106.8	107.2	150.8	16.1	16.5
tcp	40.95	34.1	33.6	51	21.9	21.1
unix	96.35	85.61	85.52	126.2	12.7	12.5
mem rd	485.7	485.7	485.7	485.7	0	0

Table 8.2: Results of lmbench benchmarking suite. The first four columns show the performance of seL4::Wombat, L4::Wombat, Wombat::Insecure and Native Linux respectively. The last two columns shows the percentage performance gain of seL4::Wombat compared to Wombat::Insecure and L4::Wombat respectively.

Except for *pagefault* latency, the performance of seL4 and Insecure for benchmarks that require the allocation of physical memory within the kernel is, in most cases similar. The slight variations are mainly caused by noise in measurements and the different levels of optimisations within the two kernels.

The only standout in this group of numbers is the handling *pagefault* latency, where seL4 outperforms Insecure by 20%. Recall that the same observation was made in Section 7.9, when comparing the performance of the seL4 virtual memory (VM) interface with that of L4. The L4 VM interface is much more abstract than that provided by seL4. L4 facilitates *fpage* mappings — an arbitrary two-to-the-power-sized page. Depending on the fpage, L4 modifies multiple page-table entries and, if required, allocates and constructs second-level page-table structures. In contrast, the VM mechanism of seL4 is closer to that of the underlying hardware interface; it only allows mappings of the hardware provided frame sizes. This simplification, however is not a result of the proposed memory allocation model, but an implementation decision to make the kernel simpler. If need be, there is no conceptual barrier for introducing fpage like VM primitive to seL4. In summary, the L4 virtual memory interface is a generic abstraction whereas the seL4 interface merely provides protected access to page tables. As such, the operation of installing a frame, in particular when the size is equal to that of hardware-protected frame size, seL4 is faster (see Table 7.9). Moreover, this simplified interface has natural synergy with what the Linux kernel expects.

The second group of *latency* benchmarks and the *bandwidth* benchmarks do not require microkernel memory allocation and therefore, there is no mediation cost (in the case of L4::Wombat) in completing these operations. All three Linux instances can handle these operations without requesting services from another server, so there is no mediation overhead. The performance of these benchmarks is mostly sensitive to the cost of *exception IPC*. When an application traps into the microkernel with a system call number which the kernel does not handle, the kernel generates and sends an exception IPC to the Linux server for it to emulate a Linux system call for the trapping application. For these benchmarks, seL4 exhibits much more modest gains. These modest gains are due to the different levels of optimisations in the exceptions IPC paths. These results are slightly biased towards seL4 as it has a hand-optimised exception IPC path completely written in assembly, versus L4 which relies on a partial hand-optimised assembly path coupled with *optimised* C for exception IPC on ARM11. The deviations in the performance for these measurements are caused by implementation — there is no fundamental reason for such a difference.

Now I compare the performance of seL4 with native Linux. Depending on the benchmark, seL4 shows modest to high performance overheads. If the benchmark is a light-weight operation within the Linux kernel, seL4 shows high performance degradation whereas for heavy-weight operations the overhead is modest. For example, consider the two system call latency tests in Table 8.2. For a *null* system call the overhead is 318% but for the *open* system call it reduces to 28% and for *shell* it further reduces to 20%. When handling a native Linux system call, the virtualised system requires at least two IPC messages — the exception IPC up-call to the Linux server and the reply message from the server to the client. If this communication cost is high compared to that of the operation performed within the Linux kernel (which is the case with *null* system call), seL4 exhibits a high overhead. But for bulkier operations within the Linux kernel the above communication cost becomes insignificant and hence seL4 exhibits a modest percentage overhead.

To validate this claim further, I used the pipe bandwidth test of lmbench with different chunk sizes and the results are shown in Figure 8.3. The *X* axis of this figure shows the

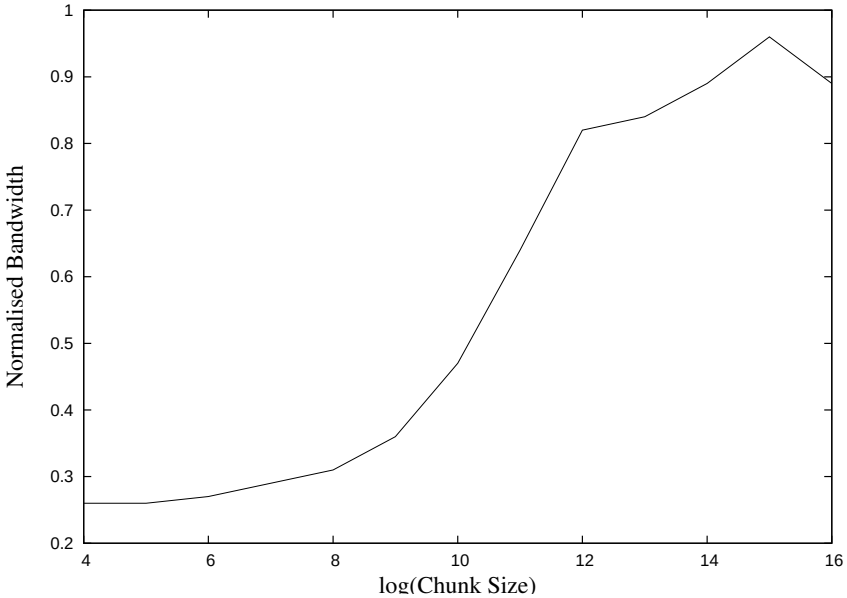


Figure 8.3: The variation of pipe bandwidth of seL4, normalised to native Linux.

chunk size in a log scale and the *Y* axis shows the normalised bandwidth of seL4. For smaller chunks seL4 has high overheads— its bandwidth is only 25% of native. But for higher chunks, the bandwidth improves to 89% of native.

These overheads, however, are not central to the core of this thesis. They stem from the exception handling mechanism of the microkernel.

Xen-Based Virtualisation

Having analysed the overheads associated with running a para-virtualised OS on seL4, let us now examine how seL4-based virtualisation compares to other virtualisation platforms.

There are a number of different systems that support para-virtualisation. To limit the scope, I selected one of the commonly used virtualisation platforms — *Xen* [BDF$^+$03], as a representative instance of a different virtualisation technique.

Unfortunately, to date, Xen does not run on ARM11-based platforms (there are plans to support ARM11 in the 2^{nd} quarter of 2009 [Com]). Therefore, a direct comparison of performance numbers is not possible. However, the Xen numbers reported for other ARM-based processor, in particular numbers reported for ARM9 (v5 architecture) based systems, can be used for a fair comparison by understanding the architectural differences between the two processors and how those differences would effect the given implementation.

Hwang et al. [HSH$^+$08] reports performance of a Xen port (based on Xen 3.0.2) for an *ARM926EJ-S* [ARM04] based system. Using the same terminology used by the above authors, I call this Xen version *Xen on ARM*. Similar to our experiments, they host a para-virtualised Linux 2.6 kernel on Xen on ARM and measure the performance by running the lmbench benchmarking suite. Even though a direct comparison between the performance numbers is infeasible due to the architectural differences of the CPUs, we can compare the *relative slowdown* caused by the two systems to understand how they measure-up as virtualisation platforms. By relative slowdowns I mean the ratio between the cost of an operation in the para-virtualised system and that of the native Linux on same hardware — the performance normalised to the native.

Benchmark	Xen on ARM	seL4
Latency	Ratio	Ratio
fork+exit	3.46	1.75
exec	3.38	1.51
semaphore	1.77	1.44
unix	1.70	1.56
syscall (write)	1.85	1.91
Bandwidth	Ratio	Ratio
pipe	0.89	0.89
unix	0.88	0.76
mem (rd)	1.04	1

Table 8.3: Normalised performance of lmbench tests for seL4::Wombat, and Xen on ARM.

By comparing the ratios, as opposed to say the measured latency of an operation, we can factor out, to some extent, the architectural differences. Then, with an insight of the architectural differences and how these differences may effect implementation and performance, we can use the normalised performances to get a fairly accurate picture of how the two systems compares to one another.

Architecture wise the main difference between ARM9 (v5 architecture) and ARM11 (v6 architecture) is the cache — ARM v5 processors has a virtually indexed and virtually tagged cache in contrast to the physically tagged cache of ARM v6 processors. For cache consistency, an ARM v5 processors requires a cache flush on every context switch, making context switching highly expensive compared to ARM v6. This architectural difference has a major impact on virtualisation when the guest OS and its applications are residing in different address spaces — every system call from a guest application will cause two context switches — from the application to the guest OS, and back. This context switching adds significant overheads to the guest's system call performance.

Xen on ARM avoids such context switching, by mapping the guest OS into the virtual address space of the guest's application and uses the ARM domain protection mechanism to protect the guest OS from its applications. Therefore, a system call from a guest application does not cause a address space switch, but a domain switch. A context switch only occurs when switching from one guest OS to another — which is not included in lmbench. This implementation keeps the main architectural difference between ARM v5 and ARM v6 outside of the critical path for the benchmarks.

The normalised performance for lmbench tests for seL4::Wombat and Xen on ARM is given in Table 8.3. The normalised performances for Xen on ARM is directly taken from [HSH+08].

The first set of numbers in this table compares the normalised performance of latency benchmarks. Except for *fork* and *exec* the two systems show similar slowdown due to virtualisation.

In the case of *fork* and *exec*, seL4::Wombat's performance is far better than Xen on ARM. This is because, Xen on ARM keeps the guest OS mapped in every application's address space, making process creation and deletion expensive — one needs to map and delete the guest OS, to the newly created or from now deleted, address space. Moreover, all map and delete operations (similar to seL4) require *hypercalls* into Xen. seL4::Wombat on the other hand, since context switching is not prohibitively expensive on ARM v6,

Test name	seL4 $[OPs/s]$	Secure $[OPs/s]$	Insecure $[OPs/s]$	Native $[OPs/s]$	Gain (Insecure) %	Gain (Secure) %
fork_test	478	367	451	979	5.8	30.0
exec_test	145	112	136	208	7	30
shell_rtns_1	98	78	94	149	4.3	25.6
brk_test	156144	142651	145915	333333	7	9.5
mem_rtns_1	299732	229970	261213	493297	14.7	30.3
page_test	19889	14915	16457	52438	20.9	33.3
create-clo	34000	31640	31738	65672	7.5	7.1
disk_src	8869	8909	8886	13993	-0.2	-0.4
dir_rtns_1	35300	35564	35619	89054	-0.9	-0.7
sync_disk_cp	4798	4876	4871	9425	-1.5	-1.6
sync_disk_rw	6192	6238	6229	13155	-0.6	-0.7
sync_disk_wrt	6179	6301	6232	13152	-1.3	-1.9

Table 8.4: Results of AIM9 benchmarking suite. The seL4, Secure, Insecure and Native columns show the performance of seL4::Wombat, L4::Wombat, Wombat::Insecure and Native Linux respectively. The last two columns shows the percentage performance gain of seL4::Wombat compared to Wombat::Insecure and L4::Wombat respectively.

keeps the guest OS and its applications in separate address spaces, reducing the number of mappings required to complete above tests. This difference is motivated from architectural factors — it is unfair to compare this set of numbers.

The second set of numbers in the Table 8.3 compares the performance of bandwidth benchmarks. There are no significant differences between the two systems for these benchmarks.

In summary, seL4::Wombat and Xen on ARM shows similar performance. However, unlike Xen on ARM, the seL4::Wombat system provides a strong formal guarantee on the amount of kernel memory a guest OS may consume.

8.4.2 AIM9 Benchmark Suite

Results of AIM9 benchmarking suite for the four Linux configurations — seL4::Wombat, L4::Wombat, Wombat::Insecure and Native Linux is shown in Table 8.4. The first column of this table gives the test name as it appears in the AIM9 suite. The following four columns show the performance of seL4::Wombat, L4::Wombat, Wombat::Insecure and Native Linux respectively. The last two columns show the performance gain of seL4::Wombat when compared with L4::Wombat and Wombat::Insecure.

Each test in the AIM9 suite runs for a fixed amount of time, performing a particular operation and measures the average number of operations per second. Results in the above table, with the exception of file system test, are obtained by running each test for 10s. File-system test results are obtained by running each test only 5s. The limitation here is the amount of ram available for the ram disk — there is not enough ram disk space to run these tests for 10s.

Similar to the previous table, the first set of numbers in this table shows the performance of the tests that required allocation of physical memory with in the operating sys-

tem. While most AIM9 test names are straightforward, some others need explanation. The *shell_rtns_1* test measures the number of shell scripts that can be executed per second and the *mem_rtns_1* measures the number of dynamic memory operations per second. Similar to our previous observation, for these tests, with the exception of *brk_test*, seL4 demonstrates performance benefits by enforcing the policy via controlled decentralisation of resource management and thereby reducing the proxy cost.

The *brk_test*, in contrast, shows no significant performance gain. This test modifies the data segment by performing a *brk()* system call, however it never touches that memory. Upon receiving the exception IPC generated by the above system call, the Linux server modifies its internal shadow page tables and bookkeeping structures accordingly and replies back to the client. Moreover, because the client does not touch that memory, the Linux server does not modify the actual page tables within the microkernel — this is done on a page fault. As such, the *brk_test* does not provide the opportunity to use the seL4's kernel memory management scheme. The slight performance gain seen for this test essentially stems for the different levels of code optimisations rather than a conceptual difference.

The second set of results in the table shows the performance of file-system-related operations that do not require allocation of physical memory within the kernel. Out of these tests, *create-clo* measures the number of file creations and closes per second, *disk_src* measures directory searches per second and *dir_rtns_1* measures directory operations per second. The remaining three tests, namely, *sync_disk_cp*, *sync_disk_rw* and *sync_disk_wrt* measure the number of disk copies, synchronous-random disk writes and synchronous sequential disk writes performed per second. These tests do not require any allocation of metadata within the microkernel and thus have no mediation via proxy. Consequently, they do not show the presence or absence of mediation overheads for the three para-virtualised systems. The performance of these benchmarks is sensitive to the memory layout of the RAM disk. The slight differences in performance for the three systems stems from different cache-hit ratios.

8.5 Summary

The results reported in this chapter show that the proposed memory allocation model causes no significant performance degradation. Furthermore, the proposed model's ability to enforce a system-level policy by controlling the delegation of authority yields better performance when compared to a similar system configuration that enforces the *same* policy (but with less restrictions) via mediation and shows similar performance to a system attempting no such control over the amount of kernel memory consumption of an application. Finally, seL4's performance as a virtualisation platform is similar to that of Xen on ARM platform.

Chapter 9

Conclusion

In-kernel memory allocation to support kernel services has a direct or indirect effect on the security and efficiency of the overall system. If there is a mismatch between the security policy enforced for user-level applications and the kernel's memory management policy, a malicious application may exploit this to circumvent system security.

Existing approaches to in-kernel memory management, with the exception of caching-based schemes, do not provide a clear relationship between the kernel's internal behaviour and the externally visible (API) protection model. Existing protection models only focus on the API level, with the tacit assumption that kernel's internal behaviour does not undermine the properties enforced at the API level. As such, one cannot use the protection model to affirm that a given application cannot circumvent the system's security policy by exploiting the in-kernel memory allocations. Caching-based approaches make providing temporal guarantees difficult if not impossible — limiting the application domains of the high assurance kernel. Moreover, the traditional approach of modifying the kernel's memory management scheme to suit the application domain invalidates any formal assurance made about the implementation correctness of the kernel and requires a significant effort to reestablish refinement proofs.

Ideally, a general purpose, high-assurance security kernel should have a unified protection model that facilitates both the enforcement and reasoning about the enforcement of different policies. This protection model should not only capture the behaviour of the API but also that of the kernel internals.

This thesis proposed a novel, systematic approach to managing the in-kernel memory of a high-assurance small kernel. The proposed model eliminates *all* implicit memory allocations from within the kernel. All memory, without exception, is controlled explicitly and precisely through an externally visible, capability-based API. As such, reasoning about and enforcing in-kernel memory management policies boils down to reasoning about and enforcing policies on the dissemination of capabilities. Further, the thesis proposes a formal model of authority by extending the take-grant model to control dissemination of capabilities. The extensions proposed to the take-grant model makes it feasible to reason about in-kernel memory consumption of an application, which is not possible in the original model.

Since all policy decisions on in-kernel memory management are done by suitably authorised user-level applications, one can enforce different policies simply by modifying user-level code, rather than the assured kernel code. The model supports diverse management policies to co-exist on the same system, by means of different user-level resource managers.

The proposed extension to the take-grant model unifies kernel memory management with the externally-visible, capability-based access-control mechanism. Thus, the proposed protection model makes it feasible to apply capability-based, take-grant protection to precisely control and reason about the memory consumption of an application. Unlike the classic model, the extended version facilitates reasoning about, and if required, controlling the precise amount of memory an application may directly or indirectly consume by analysing and controlling the distribution of capabilities.

I have developed a formal specification of the proposed protection model using the interactive theorem prover Isabelle/HOL. The formal analysis carried out on this model affirms that it is capable of enforcing *at least* two useful policies—spatial partitioning and isolation. All the formalisms, specifications and proofs that constitute the above analysis are machine-checked in the Isabelle/HOL theorem prover.

Moreover, the formal analysis carried out on the extended model shows that the proposed extension preserves the decidability of the classic take-grant model. This work demonstrates that it is feasible to extend the capability-based, take-grant protection to control the in-kernel memory resources while preserving the decidability of take-grant.

Formally connecting the proposed protection model with the kernel implementation and thereby formally connecting the proved properties with the deployed kernel is beyond the scope of this thesis. A related project called L4.verified is working towards achieving such a formal connection. The thesis *only* presents an informal connection between the kernel API and the proposed protection model.

Performance measurements carried out on a prototype kernel implementing the proposed memory-allocation model demonstrates that exporting all in-kernel memory allocations to user-land applications incurs only a small performance penalty in our experiments. At a micro-level, the performance of the prototype kernel is approximately equal to that of a kernel with similar functionality but with a kernel-integrated, implicit memory-allocation scheme.

The macro-level performance characteristics of the proposed model were analysed by using the prototype kernel as a hypervisor to support, and to enforce strict control over the memory consumption of a para-virtualised Linux kernel. Results of standard OS benchmarking suites were compared with two similar hypervisors setups — one which does not attempt any control over the amount of kernel memory the guest OS may consume (*Wombat::Insecure*), and a setup which enforces a strict limit on the amount of kernel memory by intercepting and monitoring system calls (*L4::Wombat*). Both Wombat::Insecure and L4::Wombat setups use an implicit, in-kernel memory allocator to manage the kernel memory allocations. Comparisons show similar performance compared to the Wombat::Insecure system, and shows significant improvements over the L4::Wombat setup for operations where memory allocation is in the critical path, and similar performance when it is not. These performance improvements are the result of replacing the required privileged, controlling intermediary with the proposed capability-based protection model that facilitates fine-grained delegation and enforcement of spatial isolation.

In summary, the proposed memory management scheme and the associated protection model provides the following advantages: (a) a direct relationship between the API-level security model and in-kernel memory consumption, (b) makes it possible to reason about and enforce different policies on in-kernel memory consumption based on authority distribution, (c) different memory management policies can be enforced by modifying the user-level applications, rather than modifying the formally assured kernel, (d) facilitate, fine-grained, user-level management of kernel memory, and (e) modest to no performance

degradation when compared to a counterpart with a in-kernel management policy.

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[AFOTH06] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal on Embedded Systems*, 2:239–247, 2006.

[AIMa] Aim benchmarks. http://sourceforge.net/projects/aimbench.

[AIMb] Aim9 benchmarks. http://www.caldera.com/developers/community/contrib/aim.html.

[AJR06] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *High performance computing*, volume 4297 of *Lecture Notes in Computer Science*, pages 338–352, Berlin Heidelberg, December 2006. Springer.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, 1991.

[APJ+01] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill framework for VM diversity. In *Proceedings of the 6th Australasian Computer Systems Architecture Conference*, Gold Coast, Australia, January 2001. IEEE CS Press.

[ARG89] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in Chorus. In *Proceedings of Workshop on Progress in Distr. Operating Sys. and Distributed System Management*, page 15, Berlin, Germany, 18-19 1989. Springer-Verlag (LNCS).

[ARM04] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, 4th edition, 2004.

[ARM05] ARM Ltd. *ARM1136JF-S and ARM1136J-S Technical Reference Manual*, R1P1 edition, 2005.

[AS90] P. E. Ammann and R. S. Sandhu. Extending the creation operation in the schematic protection model. In *proc. 6th Anual Computer Security Applications*, pages 340–348, Tucson, AZ, USA, March 1990. IEEE Comp. Soc.

[AS91] P. E. Ammann and R. S. Sandhu. Safety analysis for the extended schematic protection model. In *proc. IEEE Symp. Research in Security and Privacy*, pages 87–97. IEEE Comp. Soc., May 1991.

[AS92] P. E. Amman and R. S. Sandhu. Implementing transaction control expressions by checking the absence of rights. In *8th Annual Computer Security Applications Conference*, pages 131–140, December 1992.

[AW88] Mark Anderson and Chris S. Wallace. Some comments on the implementation of capabilities. *The Australian Computer Journal*, 20(3):122–33, 1988.

[BA03] Michael D. Bennett and Neil C. Audsley. Partitioning support for the L4 microkernel. Technical Report YCS-2003-366, Dept. of Computer Science, University of York, 2003.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, USA, February 1999. USENIX.

[BFF+92] Alan C. Bromberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 95–112, Seattle, WA, USA, April 1992.

[BH70] Per Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13:238–250, 1970.

[Bir98] R. S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 2nd edition, 1998.

[Bis81] Matt Bishop. Hierarchical take-grant protection systems. *SIGOPS Oper. Syst. Rev.*, 15(5):109–122, 1981.

[Bis84] Matt Bishop. *Practical Take-Grant Systems: Do They Exist?* PhD thesis, Dept. of Computer Sciences, Purdue University, West Lafayette, May 1984.

[Bis96] Matt Bishop. Conspiracy and information flow in the take-grant protection model. *Journal of Computer Security*, 4(4):331–360, 1996.

[Bis02] Matthew A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, Boston, USA, 2003.

[BKW94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, CA, USA, 1994. USENIX/ACM/IEEE.

[BL73] D.E. Bell and L.J. LaPadula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol 2, MITRE Corp., Bedford, MA, November 1973. Reprinted Journal of Computer Security, 4(2,3), pp. 239263, 1996.

[BL76] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., March 1976.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. Boston, MA, USA, Winter 1994.

[Boy09] Andrew Boyton. A verified shared capability model. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 4th Workshop on Systems Software Verification SSV'09*, Electronic Notes in Computer Science, pages 99–116, Aachen, Germany, June 2009. Elsevier. To appear.

[BS79] Matt Bishop and Lawrence Snyder. The transfer of information and authority in a protection system. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 45–54, New York, NY, USA, 1979. ACM Press.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995.

[CD94] David R. Cheriton and K. Duda. A caching model of operating system functionality. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 14–17, Monterey, CA, USA, November 1994.

[CJ06] Sangyeun Cho and Lei Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[CKS08] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Mu noz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182, Montreal, Canada, August 2008. Springer.

[Com] Xen Open Source Community. Xenarm. http://wiki.xensource.com/xenwiki/XenARM. Last visited 2008-02-10.

[Coy] Coytos web site. URL http://www.coyotos.org/. Last visited 10.04.2009.

[CYC+01] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[Dac93] Marc Dacier. A Petri Net representation of the take-grant model. In *proceedings of the Computer Security Foundations Workshop VI (CSFW '93)*, pages 99–108, Washington - Brussels - Tokyo, June 1993. IEEE.

[dBB08a] Willem de Bruijn and Herbert Bos. Beltway buffers: avoiding the OS traffic jam. In *27th IEEE International Conference on Computer Communications (INFOCOM 2008)*, Phoenix, Arizona, April 2008.

[dBB08b] Willem de Bruijn and Herbert Bos. PipesFS: Fast Linux I/O in the Unix tradition. *ACM Operating Systems Review*, 42(5):55–63, July 2008.

[Den76] Dorothy. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–242, 1976.

[DJ01] Jörg Desel and Gabriel Juhás. "What is a Petri Net?". *Lecture Notes in Computer Science*, 2128:1–25, 2001.

[dL05] Benot des Ligneris. Virtualization of Linux based computers: The Linux-VServer project. *Annual International Symposium on High Performance Computing Systems and Applications*, 0:340–346, 2005.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[Dub98] Rohit Dube. A comparison of the memory management sub-system in FreeBSD and Linux. Technical Report CS-TR-3929, Department of Computer Science, University of Maryland, College Park, MD20742, September 1998.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

[EGK95] Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-level virtual memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.

[EH01] Antony Edwards and Gernot Heiser. Secure OS extensibility needn't cost an arm and a leg. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, page 168, Schloss Elmau, Germany, May 2001.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, San Diego, CA, USA, May 2007.

[EKE07] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. Technical Report NRL-1474, NICTA, October 2007. Available from http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf.

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In Jim Woodcock and Natarajan Shankar, editors, *Proceedings of VSTTE 2008 — Verified Software: Theories, Tools and Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 99–114, Toronto, Canada, October 2008. Springer.

[EKK06] Kevin Elphinstone, Gerwin Klein, and Rafal Kolanski. Formalising a high-performance microkernel. In Rustan Leino, editor, *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, Microsoft Research Technical Report MSR-TR-2006-117, pages 1–7, Seattle, USA, August 2006.

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, New York, NY, USA, 2005. ACM.

[Elp99] Kevin Elphinstone. *Virtual Memory in a 64-bit Microkernel*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 1999. Available from publications page at http://www.disy.cse.unsw.edu.au/.

[ERT] ERTOS. *Iguana user manual*. Available from http://ertos.nicta.com.au/software/kenge/iguana-project/latest/userman.pdf.

[FB96] Jeremy Frank and Matt Bishop. Extending the take-grant protection system. Technical report, Department of Computer Science, University of California at Davis, 1996.

[FN79] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conference Proceedings, 1979 National Computer Conference*, pages 329–334, New York, NY, USA, June 1979.

[Fre06] FreeBSD Documentation Project. *FreeBSD Architecture Handbook*, 2006. http://www.freebsd.org/doc/en/books/arch-handbook/.

[Gli84] Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Trans. Softw. Eng.*, 10(3):320–324, 1984.

[Han99] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, New Orleans, LA, USA, February 1999. USENIX.

[Har84] Michael A. Harrison. Theoretical issues concerning protection in operating systems. Technical report, Berkeley, CA, USA, 1984.

[Har85] Norman Hardy. KeyKOS architecture. *ACM Operating Systems Review*, 19(4):8–25, October 1985.

[Hat97] Les Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, 1997.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89, July 2006.

[HBG+07] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 41–50, Washington, DC, USA, 2007. IEEE Computer Society.

[HE03] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, volume 2823 of *Lecture Notes in Computer Science*, Aizu-Wakamatsu City, Japan, September 2003. Springer Verlag.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, October 1997.

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 113–126, Seattle, WA, USA, April 1992.

[HPHS04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the 11th SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[HR75] Michael A. Harrison and Walter L. Ruzzo. On protection in operating systems. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 14–24. ACM, 1975.

[HR78] M. Harrison and W. Ruzzo. Monotonic protection systems. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 337–365. Academic Press, New York, USA, 1978. Monotonic HRU.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, pages 561–471, 1976.

[HSH+08] Joo-Young Hwang, Sang-bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System

virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference*, pages 257–261, Las Vegas, NV, USA, January 2008.

[HT05a] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd Workshop on Programming Languages and Operating Systems*, July 2005.

[HT05b] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd Workshop on Programming Languages and Operating Systems*, Glasgow, UK, July 2005.

[IAD07] Information Assurance Directorate. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, June 2007. Version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

[IBM02] IBM K42 Team. *Utilizing Linux Kernel Components in K42*, August 2002. Available from http://www.research.ibm.com/K42/.

[Inc] Sun Microsystems Inc. Solaris resource manager 1.0 (white paper). http://www.sun.com/software/white-papers/wp-srm/. Last visited 2008-02-10.

[Kau05a] Bernhard Kauer. L4.sec implementation — kernel memory management. Dipl. thesis, Dresden University of Technology, May 2005.

[Kau05b] Bernhard Kauer. L4.sec implementation: Kernel memory management. Diploma thesis, Chair for Operating Systems, Dresden University of Technology, May 2005.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009. ACM.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[Lam71] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in ACM Operating Systems Review, **8**(1), January 1974, pp 18–24.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[Lan81] Carl E. Landwehr. Formal models for computer security. *ACM Comput. Surv.*, 13(3):247–278, 1981.

[LE96] Jochen Liedtke and Kevin Elphinstone. Guarded page tables on MIPS R4600 or an exercise in architecture-dependent micro optimization. *ACM Operating Systems Review*, 30(1):4–15, January 1996.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 28–31, Cape Cod, MA, USA, May 1997.

[LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 213–227, Montreal, Canada, June 1997. IEEE.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[LIJ97] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 73–79, Cape Cod, MA, USA, May 1997. IEEE.

[LM82] Abe Lockman and Naftaly H. Minsky. Unidirectional transport of rights and take-grant control. *IEEE Trans. Softw. Engin*, 8(6):597–604, November 1982.

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977. Original Take-grant paper.

[LvSH05] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, April 2005.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[McL85] J. McLean. A comment on the 'basic security theorem' of Bell and LaPadula. *IPL: Information Processing Letters*, 20, 1985.

[Min84] Naftaly H. Minsky. Selective and locally controlled transport of privileges. *ACM Trans. Program. Lang. Syst.*, 6(4):573–602, 1984.

[MPO08] Memory placement optimization. opensolaris.org/os/community/performance/mpo_overview.pdf, 2008. Last visited 2008-02-10.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, USA, January 1996.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, August 1990.

[MYHH08] H. Min, S. Yi, J. Heo, and J. Hong. Slab-based memory management scheme for sensor operating system. In T. F. Gonzalez, editor, *Proceedings of Parallel and Distributed Computing and Systems*, Orlando, Florida, USA., November 2008. ACTA Press.

[Nem00] University of Cambridge Computer Laboratory. *The Nemesis System Documentation*, 2nd edition, January 2000.

[NF03] Peter G. Neumann and Richard J. Feiertag. PSOS revisited. In *19th Annual Computer Security Applications Conference*, Las Vegas, December 2003.

[NIC05] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N1*, October 2005. http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[Nuu95] Esko Nuutila. *Efficient transitive closure computation in large digraphs*. PhD thesis, Helsinki University of Technology, June 1995.

[Ope] Open Kernel Labs. OKL4 community site. http://okl4.org.

[osU08] Heise open source UK. Kernel log: More than 10 million lines of Linux source files. http://www.heise-online.co.uk/open/Kernel-Log-More-than-10-million-lines-of-Linux-source-files--/news/111759, October 2008. Visited Feb. 2009.

[PT04] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX Large Installation System Administration Conference*, Atlanta, GA, USA, November 2004.

[RDH+96] John Rosenberg, Alan Dearle, David Hulse, Anders Lindström, and Stephen Norris. Operating system support for persistent and recoverable computations. *Communications of the ACM*, 39(9):62–69, September 1996.

[RF97] Dickon Reed and Robin Fairbairns. *Nemesis Kernel Overview*, May 1997.

[RMSK00] John Reumann, Ashish Mehra, Kang G. Shin, and Dilip Kandlur. Virtual services: a new abstraction for server consolidation. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[Rus81] John M. Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 12–21, 1981.

[Rus99] John Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[San88] Ravinderpal Singh Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM*, 35(2):404–432, April 1988.

[San92a] R. S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136. IEEE, 1992.

[San92b] R. S. Sandhu. Undecidability of safety for the schematic protection model with cyclic creates. *J. Comput. Syst. Sci*, 44(1):141–159, 1992.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–228, November 1996.

[SFS96] Jonathan S. Shapiro, David F. Faber, and Jonathan M. Smith. State caching in the EROS kernel—implementing efficient orthogonal peristence in a pure capability system. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems*, pages 89–100, Seattle, WA, USA, November 1996.

[Sha98] Jonathan S. Shapiro. *EROS Object Reference Manual*, 1998.

[Sha99] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[Sha03] Jonathan S. Shapiro. The practical application of a decidable access model. Technical Report SRL-2003-04, SRL, Baltimore, MD 21218, November 2003.

[SJV+05] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *21st Annual Computer Security Applications Conference*, 2005.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 335–350, Stevenson, WA, USA, October 2007.

[Sny81] Lawrence Snyder. Theft and conspiracy in the Take-Grant protection model. *Journal of Computer and System Sciences*, 23(3):333–347, December 1981.

[SP99] Oliver Spatscheck and Larry L. Petersen. Defending against denial of service attacks in Scout. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999.

[SPHH06] Lenin Singaravelu, Carlton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the 1st EuroSys Conference*, pages 161–174, Leuven, Belgium, April 2006.

[SS92] R. S. Sandhu and G. S. Suri. Non-monotonic transformation of access rights. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 148–163, 1992.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Charleston, SC, USA, December 1999.

[SSJ+05] Shahriari, Sadoddin, Jalili, Zakeri, and Omidian. Network vulnerability analysis through vulnerability take-grant model (VTG). In *ICIS: International Conference on Information and Communications Security (ICIS), LNCS*, 2005.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24:412–418, 1981.

[SW00] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *IEEE Symposium on Security and Privacy*, pages 166–181, Washington, DC, USA, May 2000.

[Szm01] Cristan Szmajda. Calypso: A portable translation layer. In K. Elphinstone, editor, *2nd Workshop on Microkernels and Microkernel-based Systems*, Lake Louise, Alta, Canada, October 2001.

[TC04] Andrew Tucker and David Comay. Solaris zones: Operating system support for server consolidation. In *Proceedings of the 3rd USENIX Virtual Machine Symposium (USENIX-VM)*, 2004.

[TKH05] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, Santa Fe, NM, USA, June 2005. USENIX.

[TLFH96] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level checkpointing through exportable kernel state. In *IWOOOS '96: proc 5th International Workshop on Object Orientation in Operating Systems*, pages 85–88, Washington, DC, USA, 1996. IEEE Computer Society.

[TP98] Jonathon Tidswell and John Potter. A dynamically typed access control model. In *Proceedings of the 3rd Australian Conference on Information Security and Privacy*, July 1998.

[TP01] Jonathon Tidswell and John Potter. A graphical definition of authorization schema in the DTAC model. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 109 – 120, 2001.

[VDGR96] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 279–289, New York, NY, USA, 1996. ACM.

[VEK+07] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11–53, 2007.

[vR01] Rik van Riel. Page replacement in Linux 2.4 memory management. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 165–172, Berkeley, CA, USA, 2001. USENIX Association.

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, 2002.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17:337–345, 1974.

[Whe01] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2001.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, December 2002.

[Wu90] M. S. Wu. Hierarchical protection systems. In *IEEE Symposium on Security and Privacy*, pages 113–123, Los Alamitos, Ca., USA, April 1990. IEEE Computer Society Press.

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, 1987.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[ZDS09] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multi-core cache management. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, April 2009.

[ZLN05] Xinwen Zhang, Yingjiu Li, and Divya Nalla. An attribute-based access matrix model. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 359–363, New York, NY, USA, 2005. ACM.