Design, Implementation and Performance of Protection in the Mungi Single Address Space Operating System

A Dissertation submitted to the School of Computer Science and Engineering of The University of New South Wales in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Jeroen Doron Vochteloo

30 June 1998

'I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement is made in the text'

Jeroen Doron Vochteloo
August 16, 1999

Abstract

Traditional operating system rely on separate address-spaces for protection. The advent of 64-bit architectures has allowed the construction of operating systems that have a single, shared address space (single-address-space operating systems). All data in a single address space can be identified by a unique, globally valid name, its address, making the sharing of information easy. Due to the rejection of the separate address space model, a new model for protection needs to be designed.

This thesis describes the Mungi single-address-space operating system, and in particular, its protection system. Protection was an integral part of Mungi's design right from the start. Protection in Mungi has one overriding goal: not to negate the advantages that are intrinsic in a single address space. In order to achieve this, protection in Mungi is based on password capabilities that allow users to share information without the intervention of the kernel. Mungi also provides support for privileged procedures that allow the safe extension of the Mungi system, as well providing system-enforced object encapsulation. These protected procedure calls are used for device drivers, page fault handlers, and protected subsystems such as database servers.

The implementation of the above model will be presented and will show that the Mungi protection mechanisms are:

- flexible
- simple,
- easy to use,
- fast, and
- do not rely on specialised hardware.

The conclusions that can be drawn from this thesis are: a single-address-space operating system provides an ideal environment for sharing, protection based on password capabilities can be efficient, and that protected procedure calls based on extension and implemented in software can be especially fast.

Acknowledgements

Although a thesis is ones own work, it nonetheless contains input from others, both intellectually and emotionally.

On the intellectual side I would like to thank my supervisor Gernot Heiser for all the support over a long number of years, particularly for the detailed proof reading of the final versions of the thesis.

To my co-supervisor, Stephen Russell, I also owe a great debt of gratitude, for without him I would never have embarked on the journey that is a thesis.

Other members of the DiSy group have also provided me with moral and academic support, Kevin Elphinstone, in particular was always there to provide a solution to my problems or to go have a beer with (or both).

Other people that I should thank are Paul Compton for finding a solution to my not being able to type; Cassandra Nock for being that solution, and staying up and taking my disjointed dictation; members of the school of computer science and engineering in general, for putting up with me for ten years.

My final thanks goes to my family, in particular my mother, who passed away during my years as a PhD student, thanks for ALWAYS being there. Without family, you have nothing....

Contents

1 Introduction **1**
1.1 Single Address Space Operating Systems 1
1.2 Mungi 2
1.3 Protection 2
1.4 Contribution and Structure of Thesis 3

2 Security and Protection **5**
2.1 Security Taxonomy/Security Principles 6
2.1.1 Hydra 6
2.1.2 DoD Orange Book 8
2.2 Security Models 9
2.2.1 Access matrix 9
2.2.2 Bell LaPadula 10
2.2.3 Data integrity 10
2.2.4 Chinese Wall 12
2.3 Mechanisms 13
2.3.1 Access control lists 13
2.3.2 Capabilities 13
2.3.3 Reference monitors 15
2.4 Case Studies 16
2.4.1 Multics 16
2.4.2 Unix 17
2.4.3 Hydra 18
2.4.4 CAP 19
2.4.5 IBM System/38 and AS/400 20
2.4.6 Eden 21
2.4.7 Amoeba 22
2.4.8 The Monash Password Capability System 22
2.4.9 Monads and Grasshopper 23
2.4.10 Mach 24
2.4.11 EROS 25
2.5 Single-Address-Space Operating Systems 25
2.5.1 Angel 25
2.5.2 Opal 25
2.5.3 Nemesis 26
2.6 Summary 26

3 Mungi **28**
3.1 Introduction . . . 28
3.2 Etymology . . . 28
3.3 Motivation . . . 28
3.4 Mungi Abstractions . . . 29
3.5 System Interface . . . 30
3.6 Object Table . . . 31
3.7 Summary . . . 31

4 Protection in Mungi **32**
4.1 Protection Philosophy . . . 32
4.2 Password Capabilities in Mungi . . . 33
4.2.1 Capability types and access modes . . . 34
4.2.2 Object table entries . . . 35
4.2.3 Password management . . . 35
4.3 Protection Domains . . . 36
4.3.1 Implicit validation . . . 36
4.3.2 Clists . . . 37
4.3.3 Active protection domains . . . 37
4.3.4 Access validation . . . 38
4.3.5 Caching access rights . . . 38
4.3.6 Tailoring protection domains . . . 38
4.4 Thread Creation . . . 40
4.5 Capability Handlers . . . 40
4.6 Protection Hardware . . . 40
4.7 Related Work . . . 41
4.8 Summary . . . 42

5 Protection Domain Extension **44**
5.1 Motivation . . . 44
5.2 Extensibility . . . 44
5.2.1 Exokernel . . . 45
5.2.2 SPIN . . . 46
5.2.3 Software fault isolation . . . 46
5.3 Mungi PDX . . . 47
5.3.1 PDX . . . 47
5.4 Device drivers . . . 49
5.4.1 Case Study: Serial Driver . . . 50
5.5 User-level page fault handlers . . . 52
5.6 Object Support . . . 52
5.6.1 Encapsulation . . . 52
5.6.2 Inheritance . . . 52
5.7 Other Services . . . 53
5.7.1 Parameter passing . . . 54
5.7.2 Case Study: OO1 . . . 54
5.8 Summary . . . 55

6 Security in Mungi **57**
6.1 Common Security Problems . . . 57
6.1.1 Hydra security requirements . . . 57
6.1.2 Other policies . . . 59
6.1.3 Language based protection . . . 60

6.1.4 Other operating systems 60
6.2 Mungi Framework 62
6.2.1 Login 62
6.2.2 Group management 62
6.2.3 Name server 63
6.2.4 Capability refinement 63
6.3 Summary 63

7 Mungi Implementation 65
7.1 Implementation History 65
7.2 L4 66
7.2.1 L4 address spaces 66
7.2.2 Tasks and threads 67
7.2.3 Interprocess communication 68
7.2.4 Page faults, interrupts and exceptions 69
7.2.5 Clans and chiefs 69
7.3 The Mungi Kernel Implementation 69
7.3.1 System calls 70
7.3.2 Bootstrapping 70
7.3.3 Thread and task creation 72
7.4 PDX 74
7.4.1 Page faults 75
7.4.2 Interrupts 75
7.4.3 Exceptions 75
7.4.4 Capability handlers 76
7.5 Supporting Data Structures 76
7.5.1 Kernel information page 76
7.5.2 Object table 76
7.6 L4 Limitations 77
7.7 Summary 78

8 Mungi Performance 79
8.1 Microbenchmarks 79
8.1.1 Null system call 80
8.1.2 Tasks, threads and IPC 80
8.1.3 Objects 81
8.2 Protection Performance 81
8.2.1 Page faults and validation 81
8.2.2 PDX 83
8.2.3 Object benchmarks 83
8.3 Summary 85

9 Conclusion 86

A Benchmarking Details 88
A.1 Environment 88
A.2 Software 88
A.3 Measurement Details 88
A.3.1 Thread measurements 89
A.3.2 Other benchmarks 89
A.4 Cache effects 90

List of Figures

2.1 Mutual suspicion . . . 7
2.2 Confinement . . . 7
2.3 Access matrix . . . 10
2.4 Bell LaPadula read-up/write-down . . . 11
2.5 Chinese Wall partitioning . . . 12
2.6 Sparse capabilities . . . 15
2.7 Reference monitor . . . 16
2.8 A typical Amoeba capability . . . 22

4.1 Mungi password capability . . . 34
4.2 A valid RWX capability . . . 35
4.3 A protection domain . . . 37
4.4 Validation . . . 41

5.1 Mungi PDX . . . 48
5.2 Generic Mungi device driver . . . 49
5.3 Serial port driver top/bottom half interaction . . . 50
5.4 Inheritance . . . 53
5.5 OO1 database example . . . 56

6.1 Unix-style access control . . . 61
6.2 Hierarchy of derived capabilities . . . 64

7.1 L4 address-space operations . . . 67
7.2 L4 clans and chiefs . . . 70
7.3 Mungi logical structure . . . 71
7.4 Mungi user threads in same APD . . . 73
7.5 Mungi thread creation . . . 73
7.6 Mungi validation caches . . . 74
7.7 An object table entry . . . 77
7.8 Mungi structure . . . 77

List of Tables

3.1 The Mungi system calls . . . 31

4.1 Object system calls . . . 36
4.2 APD system calls . . . 39

5.1 PdxCall parameters . . . 48

8.1 Microbenchmark timings . . . 80
8.2 Validation overhead . . . 82
8.3 Cross domain calls . . . 83
8.4 OO1 benchmark times, single process . . . 84
8.5 OO1 benchmark times, multiple process . . . 84

Chapter 1

Introduction

Modern operating systems are a complex interplay of a large number of integrated subsystems, like memory management, I/O management and process management. A concentration on these subsystems has often seen protection and security added to an operating system as an afterthought. This design decision results in less than acceptable operation of the protection and security systems. Mungi is a system that has incorporated protection from the start to produce a simple, flexible protection system. The design, implementation and performance of the Mungi protection system is presented in this thesis.

1.1 Single Address Space Operating Systems

A system in which all transient and persistent data can be addressed using a globally valid name is called a single-address-space system. Such systems provide an ideal environment for the sharing of data. This advantage was recognised by systems like IBM System/38 [HSH81], Monads [RA85] and Psyche [SLM90]. The first two systems used custom hardware to implement an address space that was bigger then the available 32 bits. It was not until the advent of the HP PA-RISC [Lee89], the MIPS R4000 [Hei91], and the DEC Alpha [Dig92] processors that "off the shelf" hardware became available that had a 64–bit addressing space[1].

A 64-bit address space represents a vast increase in addressability over the previous 32 bits, making it possible to hold all the transient and persistent data in a building sized system of thousands of nodes. If a single-address-space system were to allocate a 4 Gbyte object every second (this represents the full addressability of a 32-bit address space), it would take a full 136 years to exhaust the address space. For another comparison, consider this box ▪, if we assume this represents the size of a 32-bit address space, then a 64-bit address space would be represented by a large, white sheet of paper measuring 65m by 65m.

The advantages of single address operating systems have been described before [Fab74] [Sol96] [WMR+95]. These include:

[1]Although most of them don't support the full 64-bit address space yet.

- Data sharing is made easier. The traditional multiple address space model relies on the fact that one process cannot address data that is in another address space. Sharing information between separate address spaces requires the intervention of the kernel and some other subsystem (usually the file system, or message passing system). In contrast, a single-address-space system gives all processes the ability to address all data in the system. This uniformity of naming and lack of explicit kernel involvement in communications makes a SASOS an ideal environment for sharing information between processes.

- Trivial process migration. In a distributed single-address-space system, process migration might be as simple as moving a thread control block from one node to another. As the process starts execution on the new node its working set is automatically faulted over to the new machine.

- Removal of reliance on message passing. The global address space is the natural conduit for the transfer of information. There is no need to introduce another abstraction like IPC channels, as data can just shared between processes by simply accessing memory.

- Lower context switch overheads. As a result of of the fact that all physical to virtual translations are the same for all processes, there is no need to flush caches and translation information when switching processes. A context switch in a single address space should only change the protection information associated with processes.

1.2 Mungi

Mungi [RSE+92, HERV94] is a persistent, 64-bit, single-address-space operating system that has been designed and implemented at the University of New South Wales. Mungi provides all processes with the same view of the shared address space, which is then used as the basis for all services such as device management and inter-process communication. Unlike some other systems, Mungi does not introduce other name spaces. As a result, Mungi's naming scheme is particularly pure; all resources are named by their virtual address.

A single-node 64–bit prototype of Mungi is currently implemented on the MIPS R4x00 processor family, specifically on a development board [Alg95], a purpose built U4600 processor platform [Pea98] and on an SGI Indy.

As with any system that is still under development, it is difficult to describe a moving target. Therefore, for the purposes of this thesis, a snapshot of 10th of April version of the system will be discussed.

1.3 Protection

Mungi features a protection system that was integrated into the system design right from the start. Protection in Mungi has one overriding goal: not to negate the advantages that

are intrinsic in a single address space. In order to achieve this, protection in Mungi is based on password capabilities that allow users to share information without the intervention of the kernel. Other design goals for the protection mechanism were:

- Flexibility: The protection mechanism should not limit the security policies that can be implemented using Mungi primitives. A limited range of security policies restricts the use of the operating system to a specific range of of environments.
- Simplicity: Protection can only be applied to entities that can be named. In the Mungi single address space there is only one name for each entity, its virtual address. This allows Mungi to implement protection simply by applying the "3 R's" (read, write and run).
- Ease of use: Good protection mechanisms don't necessarily translate to good security. If protection is too invasive or counter-intuitive users will not routinely apply it. Therefore, operating system primitives should allow the intuitive application of good security practice. Further, while a naïve user should not need to understand the protection mechanism to apply good security practice, the security conscious user, who does understand the protection mechanism, should be able to arbitrarily enact any security policy that they wish.
- Performance: The need to minimise the performance impact is obvious. Protection is not tolerated if the costs are prohibitive.

The Mungi protection mechanism provides support for privileged procedures that allow the safe extension of the Mungi system, and are used for device drivers, page fault handlers, and protected subsystems such as database servers.

1.4 Contribution and Structure of Thesis

The major contributions made by this thesis is that it:

- introduces a protection system for a single-address-space operating system based on password capabilities;
- provides compelling evidence that software implementations of password capabilities can be fast;
- adapts and extends the protected procedure call mechanism introduced by the IBM System/38. This is done in software without sacrificing performance;
- describes an implementation of Mungi running of top of the L4 μkernel and measures the performance of the Mungi kernel;
- shows that the capability model of Mungi is a flexible protection system, capable of solving a set of well known security problems;

- demonstrates that Mungi's "pure" address space environment together with the protected procedure calls can be used to implement device drivers;
- describes how Mungi protected procedure calls can be used to support object encapsulation; and
- discusses approaches to the confinement problem in a password capability system.

The remainder of this thesis will describe the design, implementation and performance of the protection system in Mungi. Chapter 2 will give an overview of the wide area of protection and security. Chapter 3 gives a brief outline of the general Mungi primitives, before discussing the Mungi protection primitives in Chapter 4. More specific protection and security issues are discussed in the next two chapters; protection domain extension in Chapter 5, and Mungi security in Chapter 6. Chapters 7 and 8 will give an insight into the implementation and performance of Mungi, before the conclusion set out in Chapter 9.

Chapter 2

Security and Protection

Any multiuser operating system must be able to ensure that processes execute free from malicious or accidental modification by other processes. This requirement can be viewed as an arbitration of the contention between all active entities in a system, for the set of resources in the system. The role of arbiter usually falls to the protection and security components of the operating system.

A protection system provides the mechanisms to support contention resolution policies (security policies). Protection was defined by Butler Lampson [Lam71] as "*a general term for all the mechanisms which control the access of a program to other things in the system.*". Defined like this, protection is a design consideration that pervades all levels of an operating system; It has implications ranging from hardware design through to user psychology. On the other hand, Hogan [Hog88] described security as a tripartite co-operative effort between "*a secure operating system managed by security-conscious system administrators and accessed by security-conscious users*". Simplistically, this reduces to: protection is the mechanism, security the policy.

A need for flexibility and good software engineering principles dictate that policy and mechanism are separated [WCC+74]. This is of particular importance in the design of a protection mechanism, because protection is intertwined with all levels of operating system design. If flexibility were not a major consideration in protection design, then any change in security policy would present a major headache in the re-implementation of not only the protection system, but a major amount of other operating system code.

In order to discuss security and protection the reader will need to be familiar with three definitions: those of *subject, object* and *protection domain.* Subject, also called agent, refers to any entity to which access policies can be applied (eg. users, processes). An object is an entity, access to which is controlled by the access control policy (eg. files, disks, printers). Each subject's protection domain is the set of objects that it can access at any time and the associated rights. The role of an access policy is to define the contents of all protection domains.

2.1 Security Taxonomy/Security Principles

In order to eventually discuss various methods of implementation of a protection mechanism, we first need to look at some common requirements of security policies.

2.1.1 Hydra

Hydra[CJ75] listed and defined a set of problems that should be handled by a security system and thus should be supported in a protection mechanism:

- mutual suspicion,
- modification,
- limiting the propagation of capabilities,
- conservation and
- confinement.

Mutual Suspicion

Mutual suspicion is a set of two related problems, both stemming from the "*principle of least privilege*". The principle of least privilege states that any procedure should execute with only the privilege that it needs to perform its job and no more. Mutual suspicion becomes an issue when one procedure calls another. The calling procedure must make sure that it only passes on the rights that the called procedure needs. The called procedure, on the other, hand must make sure that the caller does not gain access to its private objects. As an example consider, a user who wants to print a file on a printer. The system is set up so that the user cannot access the printer spool area directly, but has to make a call to the printer spooler. If the user and the printer spooler are mutually suspicious, the the following would happen:

- the user would only pass the read rights for the file to the printer spooler; this being the only file that the spooler would need access to;
- the spooler would have to make sure that the user did not get the rights to access the spool area directly.

In Figure 2.1, the user (U) is calling the printer spooler (P). Note that the user has various other objects in its protection domain, but the only rights that are granted to P are those to the file that is about to be printed. On the spooler side, the printer spool is wholly within the protection domain of the printer spooler.

Modification

Modification concerns the protection system's ability to guarantee that an object that is passed to a procedure is not modified by that procedure.

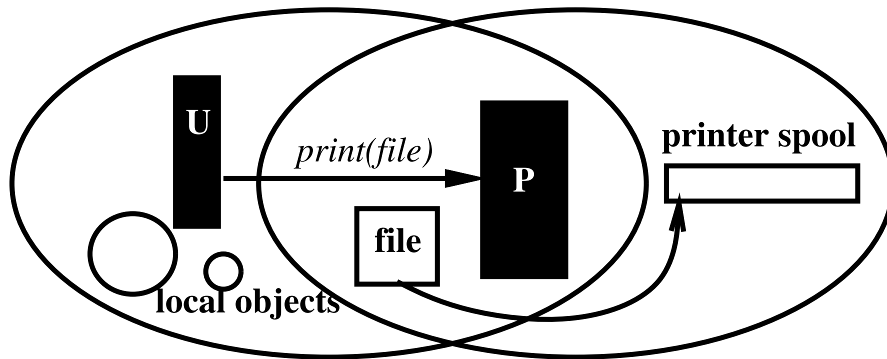

Figure 2.1: Mutual suspicion: user (U) and printer spooler (P)

Capability propagation

This problem involves the propagation of access rights. A user should be able to grant access rights to an object to another user, without the second user being able to hand the rights to a third party. In the above example, we need to prevent the spooler from handing on the right to read the file.

Conservation

This protection problem is a temporal version of the principle of least privilege, in that a procedure does not hold on to a set a rights longer than is needed to complete a job. Using the spooler/user example from earlier, a user wishes to give the spooler access to the file while it is being printed. The user then wants to make sure that access to the file is revoked once the spooler has finished printing the file.

Confinement

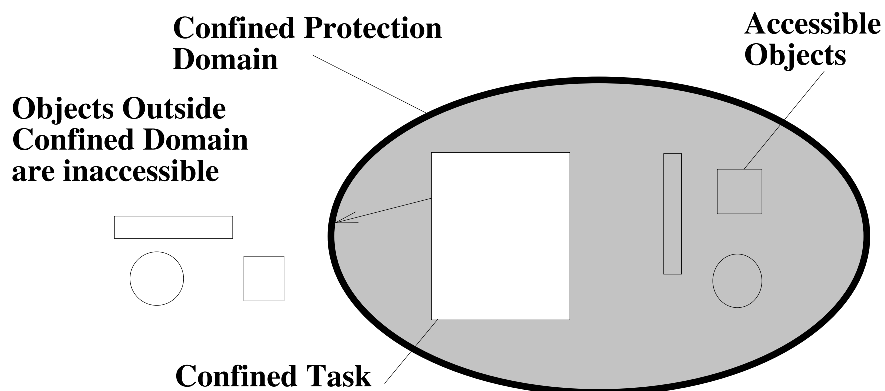

Figure 2.2: Confinement

Sometimes it is required to pass sensitive information to an untrusted procedure. In this case, it is essential that the system has a method to guarantee that the data are not passed onto third party. This is different to capability propagation which is concerned with guaranteeing that rights are not passed on. The issues that are involved in making the guarantee that no data is leaked is called the confinement problem [Lam73]. Logically this involves erecting a barrier around the confined process to prevent the leakage of information, as illustrated in Figure 2.2. The confinement problem has recently received renewed interest in regards with the use of Java and Active X applets that run locally on web servers [SW97].

In addressing this problem, Lampson highlighted a series of channels where information might be leaked to a third party. Obviously, blocking all of those channels results in confinement. To complicate the problem, there are channels that are not easy to block. These so-called *covert channels* can involve approaches such as a confined process varying the page fault rate in order to communicate information to a third party. Lipner [Lip75] claims the that use of "virtual time" provides a solution to blocking covert channels, but concedes that his method could not be used with a time-sharing system that uses any form of adaptive resource sharing. Virtual time means that a user program has associated with it its own clock and that operations like page faults have assigned to them a fixed time that does not depend on other processes that can be running at the time. In general, the problem of blocking covert channels is ignored.

2.1.2 DoD Orange Book

The US Department of Defence defined a taxonomy of security policies in the "Department of Defence Trusted Computer System Evaluation Criteria" [DoD85] (also known as the "Orange Book"). The three main aims of the taxonomy were:

1. to allow users to have some idea of the amount of trust to be placed in the operating system;
2. to give system designers some targets to aim for; and
3. to allow specification of security requirements.

It is interesting to note that suppliers of various versions of Unix such as SCO and Solaris [Sun] as well as Microsoft Windows NT [NT95], and other modern operating systems have gone through the trouble of getting a DoD security accreditation for their operating systems. In this respect, the above aims seem to have succeeded to some degree.

The Orange Book defines 4 levels of security, with levels from A to D. Each of the levels in this taxonomy also may have a number of sub-levels. The taxonomy is arranged so that each defined level has all the features of the levels below it.

Class A1 Verified Design

Class B3 Security Domains

Class B2 Structured Protection

Class B1 Labeled Security Protection

Class C2 Controlled Access Protection

Class C1 Discretionary Security Protection

Class D Minimal Protection

Level C provides discretionary access control, which gives the control over access to an object to its owner. The owner is able to set arbitrary $user \rightarrow \{right\}$ mappings (where some of those rights might be null). Level B requires mandatory access control, in which access control policies are only set by the system, and must not be modifiable by any users. Mandatory access control is commonly used in military systems. Level A is essentially the same as Level B, but with formal verification ensuring that the implementation and design are correct. The lowest level, D, is for those systems that do not meet the specifications of any of the higher levels.

2.2 Security Models

A given system is "secure" only with respect to some specific policy.

Ames, Gasser and Schell, 1983

A security policy determines the validity of all accesses that are made in a system. Security policies can be based on who a certain user is, what objects that users has previously accessed, time of day, or just about any other factor. The protection mechanisms provided by the system should not limit the choice of a suitable security policy. To illustrate some of the environments that a protection system might have to support, it is worthwhile discussing a cross section of existing security policies:

- access matrix
- Bell-LaPadula
- data integrity
- Chinese Wall

2.2.1 Access matrix

An access matrix is a logical representation of a security policy. It uses a matrix to model the set of allowable accesses from subjects to objects in the system at any one instance; the term access matrix was introduced by Lampson [Lam71]. One dimension of an access matrix represents all the active entities in the system, while the other dimension represents all the resources in the system Each entry in the access matrix corresponds to the set of operations that an active entity can perform on the resource (see Figure 2.3).

domain \ object	Device 1	Printer 1	File A	File B	Disk 1	Device 2	File C	Disk 2	Printer 2
D1	read	print					read		
D2				read write			exec		
User A	read	print	read				exec	read write	
User B		print					exec		print
D 3			read write				exec		
User C					read write				

Figure 2.3: Access matrix

2.2.2 Bell LaPadula

Bell and LaPadula [BL76] introduced a security policy which was aimed at a military environment. Their policy consisted of subjects, object and an access matrix like the system described by Lampson. The difference came, however, in a supplementation to the system, an ordered list of security levels (eg. *unclassified, confidential, secret* and *top secret*). Each object in the system has an associated security label corresponding to one of the security levels. In addition, each subject is associated with a *clearance level*. The legality of any operation in the system is determined by the access matrix, subject to the following two constraints, which are shown in Figure 2.4:

read-down: A subject may only read an object if their clearance level is above that of the security label of the object.

write-up: A subject may have only append rights to objects that have a security label that is higher than the security clearance of the subject.

This model is concerned about *information flow*, which is of great importance when dealing with military data.

2.2.3 Data integrity

Clark and Wilson [CW87] felt that most of the discussion about security policy had centred around military environments where information flow was of major concern, and felt that in a commercial environment there were more pressing concerns. The main concern that they isolated was *data integrity*; that is, how to make sure that the information that exists is correct and has only been modified in a controlled manner.

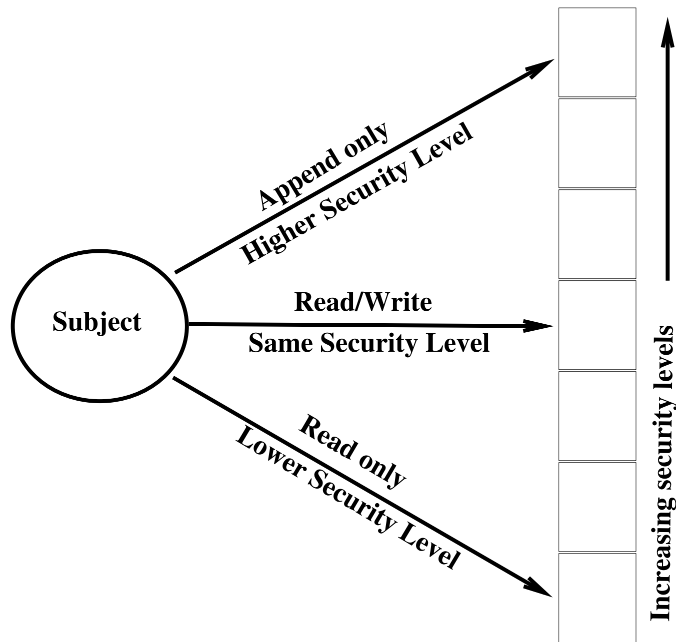

Figure 2.4: Bell LaPadula read-up/write-down

In their model, data that has to be kept consistent is contained in a module called a *constrained data item* (CDI). Modification to a CDI is limited to *transformation procedures* (TPs). There is also a function that checks the consistency of the system, called a *integrity verification procedure* (IVP). As an example, Clark and Wilson considered a simple double-entry accounting system. The data that we wish to keep consistent (CDI) is the record of the accounts. Users can only manipulate these accounts through a procedure (TP) of double entry bookkeeping. At any point we can run an audit which can be used to verify (IVP) the accounts.

Clark and Wilson went on to formulate nine requirements of a system that preserves data integrity. These rules are given below, with the C-rules being the certification rules and the E-rules being the enforcement rules.

C1 All IVPs must ensure that all CDIs are in a proper state every time that they are run.

C2 All TPs must be certified to be valid. ie that they perform their operations correctly.

E1 The system must ensure that only valid TPs modify CDIs.

E2 The system maintains a list of what objects a given TP may access on behalf of a given user.

C3 The list in E2 must meet the *separation of duty.* This means that all complete operations must involve more then one person. This is to ensure that if fraud occurs it can only occur when two or more people conspire to commit fraud.

E3 The system must be able to uniquely identify each user that calls a TP.

C4 All TPs must write to an append-only log to record their actions.

C5 No TPs must act on spurious data.

E4 Only the agent authorised to do certification can change the list E2.

2.2.4 Chinese Wall

The prevention of conflicts of interest is of some importance in a business environment. The *Chinese Wall policy* [BN89] is designed for exactly that purpose. In fact, the Chinese Wall security policy is enforced by law for the stock exchange or other corporate dealings [Cor89].

The policy partitions files into conflict classes. Two objects are in the same conflict class if knowledge of both would be construed as a conflict of interest. All access is initially legal, but as soon as a file within a conflict class is accessed, all other objects or files that belong to other members of this conflict class are rendered inaccessible.

As an example (see Figure 2.5), consider a stock brokering firm which might have conflict classes for banks and mining companies. Within the Bank class there might be three banks A, N and Z; and within the Mining class there might be mining companies B, H and P. The act of accessing files that belong to Bank A will preclude access to files from Bank N and Z. The user is still free to access files from the Mining conflict class. Once a choice has been made as to which of the mining company's files a user is going to access, the policy precludes them from accessing any other group within that conflict class.

The key feature of the Chinese Wall policy is that the information that has been accessed previously by a certain user determines future access rights to other information.

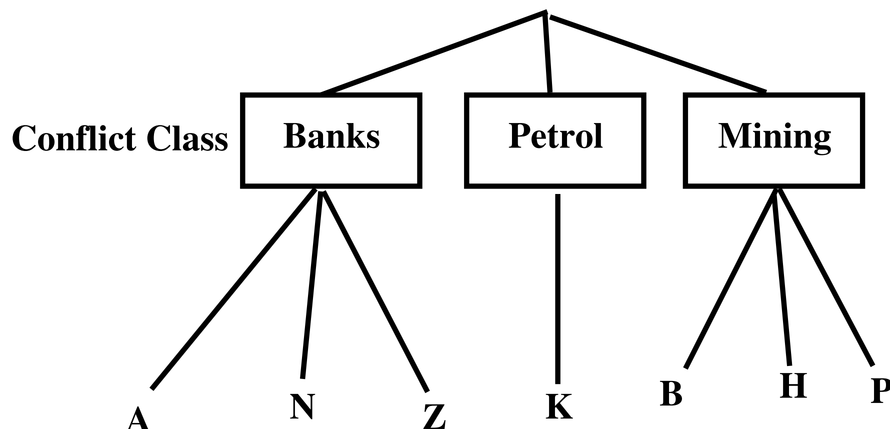

Figure 2.5: Chinese Wall partitioning

2.3 Mechanisms

The previous section presented several security models, it is now necessary to consider the mechanisms that will be used to enforce these policies. These include access control lists, capabilities, and reference monitors.

2.3.1 Access control lists

Access control lists (ACLs) are associated with objects. In a system which has protection implemented by access control lists, all objects have associated with them a list of <subject,rights> pairs, which is referred to as the access control list for that object. When a subject makes an access to an object, the object's ACL is searched to determine if the subject is making a valid access.

Access control lists have the advantage that they specify explicitly the subjects that have access to an object. They can also implement group accesses: by making subjects members of a group, group access rights in the ACL then give access rights to all members of the group. Another advantage of access control lists is that they can contain negative rights. This makes it easy to exclude certain users from accessing an object by inserting negative rights to the object for that user.

2.3.2 Capabilities

> ***Capability based discretionary access control on the other hand supports closer adherence to the principle of least privilege and allows the passing of access rights between processes and principals without mediation by the resource server.***
>
> **Dan. M. Nesbit, 1987**

Capabilities [Dv66] are tickets that grant access to a object. Capabilities are *prima facie* evidence that the holder of the capability has access to the corresponding object. Capabilities have several advantages:

- Capabilities introduce a global naming scheme [Fab74]. This allows users to reference an object using with a globally valid name, giving a mechanism that allows for simple sharing of objects.
- Capabilities provide a good adherence to the principle of least privilege. As a capability confers a set of rights on a single object, passing capabilities as arguments to procedure calls allows users to tailor their protection domain in a natural and intuitive way [Nes87].

Logically, capabilities consist of two parts. This first part is the name of the object that the capability confers rights to. This name needs to be globally valid and unique. The

second part of a capability is some indication of the actual sets of rights that the capability confers. One of the drawback of capabilities is that as capabilities are associated with subjects, it is hard to generate a list of what subjects that can access a certain object.

It should be obvious that capabilities need to be protected from modification. If subjects were allowed to construct their own capabilities, protection would be rendered useless. The issue then is: How do we protect capabilities from forgery?

There are three main ways that capabilities can be implemented [AW88]:

- tagged,
- segregated, or
- sparse.

Tagged capabilities are implemented by physically tagging the data item in memory with a *capability bit.* The hardware must either prevent modification of the data word, or a write to the word must reset the tag bit, making the capability contained in the word invalid. This method of implementing capabilities obviously requires special hardware and is thus not a very portable way of implementing capabilities. Tagging allowed early systems such as the IBM System/38 [Ber80] and later Sward [MB80], to mix data and capabilities (the advantages of which were espoused by Jones [Jon80]).

Another method of implementing capabilities is *segregation.* To protect capabilities in this way, lists of capabilities belonging to a subject are stored in a region of memory which is only accessible to the kernel. This allows traditional memory management to be used to prevent users from writing to the capability segment. Usually, in these systems, users reference capabilities using indices into the table in the kernel. Examples of systems that use segregated capabilities include Hydra [CJ75], CAP [Coo78] and Mach [RTY+88].

The third way of protecting capabilities is to employ sparseness in the capability space. This offers a probabilistic level of protection for capabilities, the strength of which can be made arbitrarily large. There are four main ways of generating secure sparse capabilities [AW88] (see Figure 2.6).

Encrypted Signature: A hash of the Object Identifier(OID) and access rights is concatenated to the capability which, is then encrypted to prevent the user deducing the hash function.

Encrypted Password: In this scheme, OIDs are chosen from a large, sparse, name space and OID and access rights are encrypted. Validity is checked by decrypting the capability and testing that the OID exists.

Amoeba: Amoeba suggested a system where the rights and a random number are encrypted together. On presentation, the encrypted word is decrypted and if the random number matches that associated with the object, the access rights are assumed to be correct [TM84].

Password: A capability consists of an OID and a large random number. This number and the associated access rights are stored with the object. The validity of a capability is determined by matching the password with the list of valid passwords for that object [APW86].

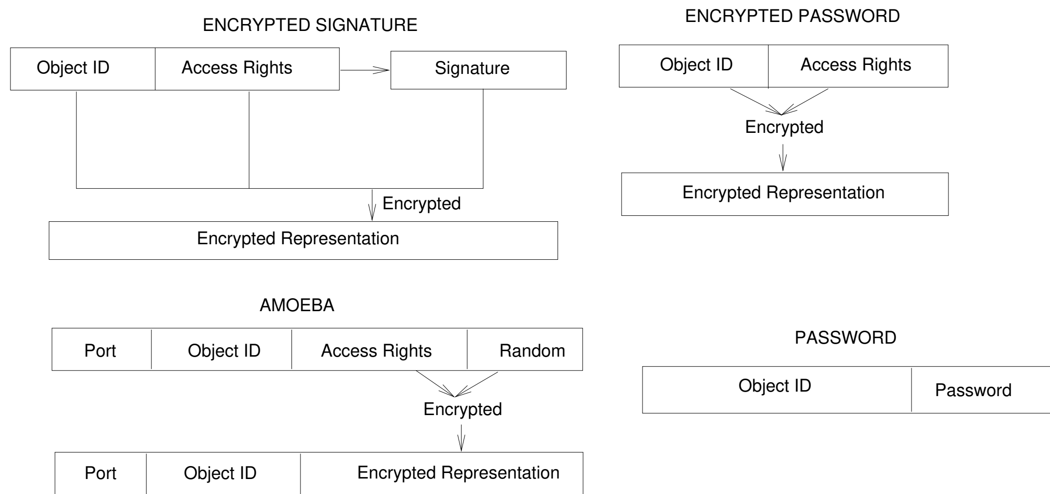

Figure 2.6: Sparse capabilities (adapted from [AW88])

.

The last two ways of implementing protection for sparse capabilities rely on the choice of a good random number. This random number should come from a search space that is large enough to prevent guessing. As the integrity of the capability relies on the inability to guess the random number, pseudo random numbers are not good enough as they reduce the search space because they are inherently predictable. Any system that requires good random numbers would need hardware support for the generation of such numbers. Wallace [Wal90] introduces a hardware device which is capable of producing 60Mbits/s of good random numbers. It is also possible to leave the generation of random numbers up to the users, who can use any means to generate these numbers such as one-way functions.

The main advantage of sparse capabilities is that they are user level objects. Users can pass capabilities from one to another, without the need for kernel intervention. Users are also able to record capabilities in arbitrary data structures.

The lack of kernel control of sparse capabilities does have a few disadvantages. Operations such as confinement and garbage collection become more difficult; although solutions such as *lockwords* [APW86] for confinement, and economic models [APW86][MT86][HLR98] for garbage collection, have been suggested.

2.3.3 Reference monitors

Reference Monitors [AGS83] actively control access from subjects to objects. When a subject wants to access an object in the system, the reference monitor is invoked. The

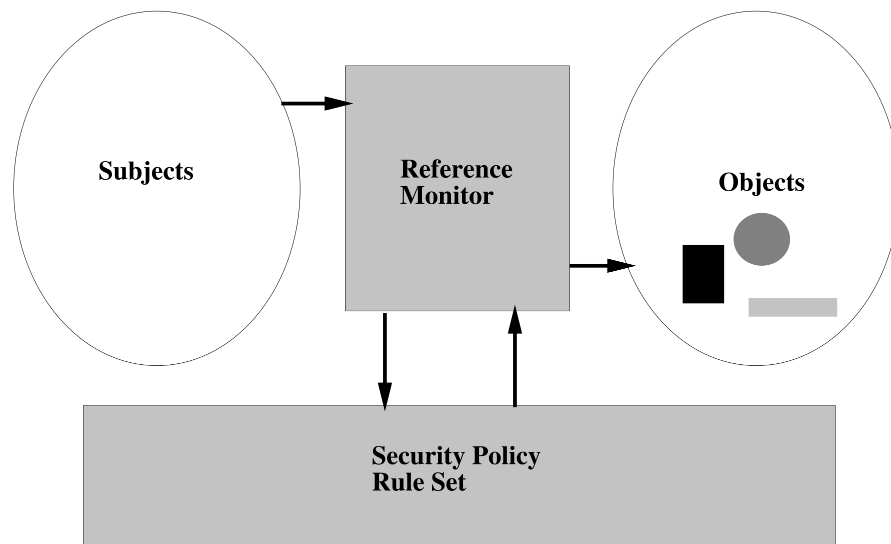

Figure 2.7: Reference monitor (adapted from Ames et. al.)

monitor will then check its rule set to determine if the access is valid or not (see Figure 2.7). The advantage of the reference monitor scheme is that by changing the rule set we can implement any security policies that we want to. Reference monitors can also be used in conjunction with capabilities or access matrices. The Bell-LaPadula security model for example can be modelled by access control lists together with a monitor which makes sure that the Read Down/Write Up policy is enforced.

2.4 Case Studies

This chapter has so far covered the area of protection and security by considering principles, models, and mechanisms. The next section will illustrate how protection and security has been implemented in previous systems.

2.4.1 Multics

The Multics [DD68, Sal74] system was designed with an emphasis on protection and security. The design of the protection system followed five main principles:

1. Protection is based on access with the default being no access.
2. Every access to every object is checked.
3. The design is not secret, as “security by obscurity” does not work.
4. The principle of least privilege is adherence to.
5. Users are able to apply protection routinely and naturally.

All data in Multics resides in a persistent storage system. This storage system also contains kernel databases and files. It is intended that the access control to the store provides all the protection in the Multics system. All information is mapped from a storage system to virtual memory. Access control lists are associated with each object in the store and each process in the system has an unforgeable user identifier associated with it, allowing arbitrary discretionary access control. The designers cite two main reasons for using Access Control Lists over capabilities: revocation is awkward and it is not possible to conduct an audit of who can touch what object when using capabilities. To allow users to implement arbitrary protection schemes, Multics provides *protected subsystems*, which are a collection of procedures and data that can only be accessed by programs within that protected subsystem. Execution of a protected subsystem can only occur at a series of predefined entry points or *gates*. Hardware support arranges protected subsystems into a hierarchy numbered from 0 to 7. The hardware allows subsystems of a certain level access only to data and procedures of a level equal or less then the current level. This nesting of protected subsystems became known as the Multics “rings of protection”.

2.4.2 Unix

Unix[1] [TR74], much like Multics, also tries to have access checks applied once only. All resources in Unix are viewed as files. It is therefore not surprising that access permissions are file based. Every process in Unix has associated with it a user id and a group id. These id’s determine the access that each process has on each file. There are three main operations that can be performed on a file, these are read, write and execute. All files in Unix also have an owner, and belong to a group. Rights to a file are specified by three sets of permissions. The first is the rights of the owner, the next set determines what rights the group members have, while the last set determines what rights all other users have.

```
unix> id
uid=10371(jerry) gid=10371(jerry)
unix> groups
jerry mungi accstaff
unix> ls -l
total 1130
-rw-------   1 jerry    jerry      19129 Oct 30 11:35 dead.letter
-rw-------   1 jerry    jerry      22190 Oct 12  1996 grub-ext2fs-floppy.gz
-rw-------   1 jerry    jerry      28400 Jul 30 09:03 in-mail
drw-r-x---   1 jerry    mungi       1024 Jul 30 09:02 mungi-src
-rw-------   1 jerry    jerry       8142 Jul 30 09:04 newgive.doc
-rw-r-----   1 jerry    mungi        322 Jul 30 08:02 proposal
```

In the above example, the user id of the process is 10371 for user jerry. User jerry also belongs to groups jerry, mungi, and accstaff. When the files in a directory are listed, the rights are represented as 9 characters following the letter indicating the file type. The first three letters indicate the rights that the owner jerry has on the file; `rw-` in the case of file `dead.letter`, corresponding to read and write permissions. The next three indicate the rights that the group has. In the case of the file `proposal`, which belongs to the group mungi, the permissions are `r--`. This means that all members of the group mungi have

[1] The Unix operating system takes it name from a play on the name Multics.

read permissions on the file. The last three letters indicate the rights that other users in the system have on the files. All of the files in the example have rights `---` in this field. This means that others users have no access to these files.

Unix provides a *protected procedure* call through the use of *set-user-id* programs. When executed, these programs run with the user id of the program's owner, usually root (the superuser to whom no access rights checks are applied). An example of this on Solaris is

```
unix> ls /usr/bin/passwd
-rwsr-xr-x   1 root     sys        23408 Oct 25  1995 /usr/bin/passwd
```

In the above example, the execute bit on the passwd file is set to s instead of x. This means that if any user executes this file it will run with the user id of root. This allows users to modify the password database, which is an operation which they are normally prevented from doing.

The Unix protection model is simple and well known. It does, however, have two main drawbacks. The first is the granularity of protection. A file can only ever have one owner and belong to one group. This prevents more than one group having access to a file. Compounding this problem is the fact that only the system administrator is able to create groups in Unix, restricting the users ability to tailor protection for their files[2]. The second problem is that set-user-id programs have been the cause of many security breaches in Unix systems, Set-user-id programs that are used to perform system duties (such as adding a file to the printer queue) usually are set to be uid root, as root is the only user that is guaranteed to be able to access the caller's file. This is in gross violation of the principle of least privilege, in that a process that only needs to have the rights to access a user file and a printer spooler actually has access to all the files in the system.

2.4.3 Hydra

Flexibility is one of the main concerns in Hydra [LCC+75, CJ75]. As a result, policy and mechanism are seen as separate issues. Hydra identified the set of protection problems discussed in Section 2.1.1, and presented solutions to these problems.

The mechanisms that Hydra provides are based on capabilities. Resources in the system are objects. Objects in Hydra have a (temporally and spatially unique) name and a type. A representation of an object contains data and may contain a capability part that contains pointers to other objects. Every instance of a procedure has associated with it a *local name space* (LNS) which defines, at that instant, the objects and respective access rights that the procedure has access to. Users can invoke other procedures by means of a **CALL** function. When a CALL is performed a new LNS is set up for the new procedure. This LNS will contain the capabilities that are specified in the capability part of the procedure. Procedures can also contain *capability templates*; these specify the type and rights of a capability that the caller must provide in order to call the procedure. Templates can be thought of as prototype capabilities. There are three types of templates in Hydra, namely:

[2]Recent implementations of Unix have added ACLs to their protection mechanisms.

1. Creation: Creates a new object to be used by the procedure.
2. Parameter: Parameter templates are the capabilities that are passed by the caller to allow the called procedure to perform an action on the specified object. If all the templates match, the capability is added to the LNS of the called procedure, but with the rights that are specified in the template.
3. Rights amplification: A capability that matches a rights amplification template has its access rights amplified in the LNS of the procedure. This allows the implementation of privileged operations.

It should be noted that any information shared between two processes has to be explicitly passed in the procedure call, forcing the users conform to the principle of least privilege.

2.4.4 CAP

> ***The intention is that each module of a program which is executed on the CAP shall have access to exactly and only that data which is required for correct functioning of the program.***
>
> **Needham and Walker, 1977**

The Cambridge CAP computer [Coo78, NW77] was intended to adhere closely to the principle of least privilege. Protection is applied to segments which are contiguous memory locations. Access to segments is controlled by capabilities. The CAP defines two types of segments, one that holds data and one that holds capabilities. There are also two types of capabilities, ones that grants access to data segments and ones that grant access to capability segments. The protection system is set up as a hierarchy, which has at the top a *master resource list* (MRL). The MRL defines the overriding base, limit and protection rights for each segment. Below this top level, each process has a *process resource list* (PRL) whose entries point to the PRL of the process above it in the hierarchy. It should be clear that all hierarchies are eventually rooted in the MRL. The hardware provides a table with 16 slots which contain pointer to capability segments. A process is allowed to have immediate access to all the capabilities that are contained in the selected capability segments. The capabilities in these capability segments point to entries in the PRL for the current process.

Protected procedures are implemented by the **enter** and **return** instructions. During a protected procedure call slots two through to six serve a special purpose. The **enter** instruction has the effect of changing slots four, five and six to point to new capability segments specified by the enter capability. Slots two and three are used to pass capabilities and parameters to and form the protected procedure. *enter* and *return* also change slots two and three; when enter is called, the reference in slot two is saved and that segment becomes inaccessible, slot three is copied to two and a new segment reference is inserted in slot three. Note that Unlike Hydra, CAP takes the approach that, in general, a list of

capabilities is passed rather than individually passing the capabilities. Both approaches offers the same levels of protection, but are tailor to different usual cases.

2.4.5 IBM System/38 and AS/400

The designers of the IBM System/38 [Ber80, HSH81] perceived the need for a high-level interface that allowed applications to be independent of the hardware. Internally, all memory is split into *segments*. Segments are uniquely identified by a 40-bit segment identifier, which is never re-used. Associated with each segment is a header that can only be seen by the system. The header contains information such as type, as well as a list of all the other segments that are associated with that segment. Operations are performed on *objects* which are high-level constructs consisting of one or more segments.

In order to encapsulate information, no provision is made to address bytes, except for a special object type called a *space*. Access to objects is controlled by capabilities called *protected pointers*. Capabilities can be freely stored anywhere in memory, as they are protected from forgery by a tag bit. The system supports four main types of capabilities, which are:

system pointers, which are used to address objects;

space pointers, which are used to address bytes within space objects;

data pointers that address bytes and have associated type information; and finally

instruction pointers, which are labels for code branches.

In order to resolve an object's virtual address from its name, System/38 provides an object called a *context* which contains a set of mappings from names to object addresses. An instruction is provided to resolve a name into a system pointer using a specified set of contexts.

For execution of a program, virtual machine instructions are translated into microcode instructions when a program is loaded into a program object. A program object cannot be modified after it has been loaded and can only by started at a specified entry point.

Protection domains are defined by objects called *user profiles*, which contain lists of authorities. The four types of authorities that can be stored in the user profile are listed below.

Storage resource: This authority defines the amount of space that a user profile is able to allocate. Any objects that are created by the profile are ***charged*** to the authority.

Privileged instructions: Allows access to a set of restricted virtual machine instructions, such as the instruction to create a new user profile (aptly called **create_user_profile**).

Access rights to objects: This set of authorities specifies a set of rights on objects. These rights can be grouped into three main groups:

- Object existence: These include delete, rename etc.

- Object access: Operations that grant access and grant authority.
- Access to object contents: insert, retrieve data from the object.

Special authorities: These grant authority to modify machine attributes, and provide implicit control over all objects. This is similar to root privileges in Unix.

A space pointer can be adjusted to point to any region in a space. There are instructions to adjust a space pointer, these do the bounds checking.

Object authorities are stored in a table for each user profile. This allows the system to search for user profiles that have authorities to objects and to list objects that a profile has access to. Objects may have public and owner rights associated with them to eliminate the need to search the user profile for matching capabilities.

Due to the fact that pointers are protected by the hardware, authorisation is only done on the first access: future accesses are assumed to have been authorised. Retracting authority on an object is only possible by destroying the object. Any references to the object become and remain invalid as segment addresses are not re-used.

In order to allow user access to segments in a controlled fashion, System/38 provides *adopted user profiles*. This allows users to who have appropriate authority to substitute or add another user's profile to their profile. A user can execute a *grant* instruction that will generate an authority that allows another user the use of its profile. This allows users to perform operations that they would not otherwise be authorised to perform.

The AS/400 is essentially the same as the System/38. Soltis [Sol96] cites the new AS/400 as an immediate success when it was introduced, but added "*Insiders, however, know that under the covers of every AS/400 lurks a System/38*". Discussion here is therefore limited to the System/38, except to note that the AS/400 is the only commercially successful capability-based system.

2.4.6 Eden

Eden [LLA+81, ABLN85] is an object-based system. Objects in Eden are called *ejects*. Each eject has a unique identifier. Ejects communicate with each other using invocation. The ability to invoke an eject is conferred by capabilities. Each capability has the id of the target eject and a set of access rights. The list of capabilities that belongs to an eject is stored in the kernel, with a copy kept in the user's address space. This was done instead of keeping the list of capabilities in the kernel and having users access the capabilities indirectly by means of an index into the kernel table, because:

- It eliminates the overhead of kernel calls to examine rights fields, etc.
- Capabilities can be directly compared. This is difficult when using an index into a list of capabilities due to the possibility of aliasing.
- Capabilities can be copied within a program just like other variables.

2.4.7 Amoeba

The dropping price of hardware, which allowed the building of large interconnected distributed systems, prompted the design of the Amoeba operating system [TM84, TMvR86]. Amoeba is an exclusively client-server operating system. All interaction between clients and servers is by means of IPC, which is based on unreliable datagrams (ie. no acknowledgement, no guarantee of delivery). Messages are sent to ports, and knowledge of a port number is taken as *prima facie* evidence that the sender is allowed to use the port. Amoeba chose to protect port numbers by using sparsity. Access to services is controlled by capabilities (depicted in Figure 2.8) which consist of a globally unique server id (port number), an identifier for the object which is managed by the server, the set of rights that the capability confers, and a check field which is a large random number. Once a server receives a message from a client the server is free to do whatever further authentication that it wants to do (potentially using the other fields in the capability). One scheme actually implemented encrypts the random number and the rights fields, handing out only the encrypted information. When a capability is presented it is decrypted and the random number compared against the original. If they match the rights fields are assumed to be correct.

The basic operations that the system provides are **trans** (transmit), **getreq** (get message) and **putrep** (put reply). Trans is used by the client to request a service. Getreq and putrep are used by the server to get a service request and to post a reply respectively. To prevent programs from issuing getreqs on arbitrary ports, a novel authentication scheme based on one-way hashing is used [MT86]. This scheme can either be implemented in hardware or in software. It is believed that there are no implementations of the hardware version.

Service	Object	Rights	Random

Figure 2.8: A typical Amoeba capability

2.4.8 The Monash Password Capability System

Anderson, Pose and Wallace [APW86, AW88] introduced a system that provided protection using password capabilities. Protection is based on objects, with all objects in the system having associated with them an inverted tree of passwords, which represent the set valid capabilities. Having a tree structure for the storage of passwords creates a hierarchy of capabilities. Deleting a password from the tree also removed all the passwords below that node in the tree, revoking all the associated capabilities.

When a capability is validated, many system tables have to be searched. In order to reduce the validation overhead, validation information is kept in a main-memory cache called the *Active Object Table* (AOT). The AOT is a hash table that is indexed with the object's name to allow fast access. To allow addressing of words in memory, all objects that

are currently active are mapped into a linear *intermediate address space* (IAS). The IAS is a single 32-bit address space that maintains a temporary mapping of all active objects to allow data in them to be addressed. It should be noted that the IAS is just an implementation choice and not an intrinsic part of the password capability system.

The Monash Password Capability System also introduced a novel way to solve the confinement problem. Each process has associated with it a 63–bit *lockword.* Before a capability is validated, the kernel will XOR its password with the lockword. In most cases the lockword will be zero and so all capabilities are validated as is. If a user wants to confine a task, the task is simply created with a non-zero lockword known only to the parent. The passwords of all capabilities to be passed to the child task must first be XORed with the lockword. During execution, all capabilities used by the child task will again be XORed with the lockword before being validated. As a result, the child task can only use the capabilities that the parent has explicitly passed to it. All other capabilities are invalid after they have been XORed with the lockword. This effectively confines the child task.

As with other password capability systems, it is very difficult to do garbage collection. Anderson, Pose and Wallace put forward a scheme that charges a size-dependent rent for all objects in the system, garbage is defined as objects that cannot be paid for. The economic model is built into the kernel and there are also mechanisms to charge processes for the objects that they use. The system is designed carefully to ensure that the charging system does not violate mutual suspicion and confinement [WP90].

2.4.9 Monads and Grasshopper

Monads and Grasshopper were related projects whose aims were to investigate object management issues such as orthogonal persistence and distribution. Both systems use capabilities to protect references to data in the system.

Monads

Monads [RA85, HR91] was a system designed to support a "*rational, engineering-like approach to the development of computer software*". Monads has hardware support for a large, persistent, single-level store, which is protected by capabilities. The Monads virtual memory is split into contiguous regions called *address spaces.* Each address space is uniquely identified by an address space number which is never re-used. An address space is logically divided into a collection of *segments*, which are accessed through *segment capabilities.* Segment capabilities are made up of the full virtual address of the base of the segment (address space number + offset), the length of the segment and the rights that are conferred by the capability. Monads also provides direct support for data encapsulation with *modules.* A module is a collection of segments that are all part of the same address space. Modules are accessed through the use of *module capabilities* that confer the rights to access the interface that is presented by the module. The reason for having access protection for both segments and objects was the observation that the frequency and type of access differed between small and larger objects.

Grasshopper

Unlike its predecessor Monads, Grasshopper [DdBF+94, RDH+96] was designed to be implemented on standard hardware. Grasshopper provides three main abstractions. *Containers* are the storage abstraction, they combine the features of address spaces and files of traditional systems. *Loci* are the execution abstraction, a locus is at any one time executing in a container, its host, but can jump between hosts. *Protection domains* define the access permissions of loci.

The data in a container are supplied by a *manager*. Sharing between containers is done through one of two mechanisms:

- Mapping: Grasshopper allows a region of one container to be mapped to a region in another container. This sets up shared memory between the two containers.
- Invocation: Each container is able to define an address that is its entry point. When a locus invokes a container, execution will start at that specified address. System calls are also implemented using the same invocation mechanism.

Access to operations on loci and containers is controlled through the use of segregated capabilities. Each entity in the system has associated with it two kernel data structures, namely:

- The *capability table*: contains all the capabilities that belong to the entity.
- The *permission group table*: basically contains the list of all existing capabilities for access to the entity.

A locus is able to access the capabilities from both its own capability table as well as that of the host container. Access to these capabilities is done through *capability references* (caprefs). A capref consists of a flag that specifies which capability table to look in (either the locus or the host container) and the index in the capability table of the required capability.

2.4.10 Mach

The Mach [RTY+88, RJO+89] μkernel is designed specifically to support a multiprocessor architecture, but is flexible enough to allow it to be ported to uniprocessor systems.

There are five abstractions is Mach; *tasks, threads, ports, messages* and *memory object*. The flexibility of the Mach system comes from its message-based protection and communication system. All entities can have an associated port, which is a communication end-point. Operations on objects are performed by sending messages to ports. For example a thread can be stopped by sending a stop message to the threads port. Operations on ports are protected by capabilities that are maintained by the kernel. These capabilities can be shared by sending them in messages.

Another feature of Mach is that it allows user-level memory management by introducing *external pagers* [YTR+87]. These external pagers are invoked due to page faults in memory objects and are responsible for resolving the fault.

Although Mach was intended to be a μkernel, it suffered from poor performance, which is shown to be a result of its large size[Lie93]

2.4.11 EROS

> ***. . . a capability system should be both faster and simpler than a comparable access-control-based system***
>
> **Shapiro, 1997**

The aim of the Extremely Reliable Operating System (EROS) [Sha97] is to show that capability-based systems can be fast, simple and secure. At the highest level, all storage is in *objects*. Objects are protected by capabilities that grant rights of invocation on those objects. All access is done through invocations allowing the EROS supervisor to only implement one system call, namely the invocation trap. One of the other contributions of the EROS system is that it provides a formal proof of correctness of its solution to the confinement problem [SW97].

2.5 Single-Address-Space Operating Systems

The traditional approach to protection of objects is by using disjoint address spaces. Therefore, single-address-space systems require a novel approach to protection. Recent projects have investigated several approaches to this problem.

2.5.1 Angel

Angel [WSO$^+$92, WM96] came out of City University London.[3] The base unit of protection in Angel is the object, a set of contiguous pages. Angel implements protection by user level protection server called the *object manager*. The object manager is responsible for all operations on objects, including access.

The current implementation of the Angel object manager implements a flexible system based on access control lists. When a new object is added to the domain, a bit string called a biscuit, together with a list of permissions required for the object, is passed to the object manager. The validity of the operation is determined by a simple rule set that is dependent on the objects that are already in the domain. Protection is thus based on the properties of the domain rather than that of a user, although a user could be represented by an object.

2.5.2 Opal

The University of Washington produced a single-address-space operating system called Opal [CLBHL92, CLFL94]. There are three main abstractions in Opal: *segment, thread*

[3] Angel is named after the nearby tube station [Wil96].

and *domain*. Segments are sections of the single address space. Threads execute in a domain, which defines the access rights of the threads.

Access control to segments is granted by password capabilities: A capability can be used to **attach** a segment to a domain. This means that all threads that are executing in that domain now have access to the segment. Similarly a segment can be *detached* from a domain. Segments can also be implicitly attached. To allow the system to find the correct capability, it is required to be registered with a name server using the *publish* operation.

Users with the appropriate authority can perform cross-domain calls by invoking a domain. To this purpose, owners of a domain can create *portals* for that domain. A portal has a 64-bit ID, knowledge of which grants the rights to invoke the domain through the portal. When a user invokes a domain through a portal, execution starts at a point that is specified by the portal. Restricting execution to start at an address that is specified by the portal allows a portal to implement its own security policy by applying further access checks before proceeding. This is logically similar to Amoeba server that were also able to implement their own local access policy.

2.5.3 Nemesis

An operating system project designed from the ground up to support multi-media applications was the goal of the Nemesis [LMB+96]. Units of protection in Nemesis are *stretches* [Han97], which are contiguous segments of the single-address-space. A protection domain is a set of mappings stretch → {right}, where the access rights supported in Nemesis are **read, write, execute** and **meta**. The meta right allows modification of access rights for the stretch, and allows mapping or unmapping of physical frames of the stretch. Stretches are protected by ACLs. A global domain is defined which allows global rights to be defined for a stretch. There are four main operations that can be performed on the ACLs that are associated with a stretch.

SetProt(pdom,rights) set the access rights for domain `pdom` to `rights`.

SetGlobal(rights) set the global rights to `rights`.

QueryProt(pdom) → rights Return access rights of domain `pdom`.

QueryGlobal() → rights Return global rights.

A stretch is allocated by a *stretch allocator*, of which there may be many, each managing a separate non-overlapping part of the address space. Each stretch must be allocated to a *stretch driver*, which is responsible for the backing store allocation of the stretch using **map** and **unmap** primitives (cf. ULPs).

2.6 Summary

Consideration of protection and security issues should be given equal weighting in design considerations as the other subsystems. Protection and security represent the implementa-

tion and policy side of a system that ensures the safety of data and programs in a system.

This chapter has examined a range of security issues and basic mechanisms for their support. These present the background for discussions of security issues in the following chapters. A range of notable approaches to protection with an emphasis on capability systems, was further presented. In lieu of discussions about these system at this stage, a comparison and critique is presented in discussion in Chapters 4 and 5 in parallel with the presentation of the Mungi protection system.

Chapter 3

Mungi

3.1 Introduction

Single address space systems allow all data to be named by a system-wide address; all data, transient and persistent, has a unique address within the global address space. This is different to dominant multiple address space paradigm. In this case each process executes in its own address-space. The differences in these two paradigms results in necessarily different services being provided by the operating system. This chapter will highlight the services that the Mungi kernel provides to the user.

3.2 Etymology

Mungi is an Australian Aboriginal word. In one central Australian Aboriginal language *mungi* means "message stick", which was used by the Australian Aboriginals to introduce a messenger to a new tribe (something not unlike a capability in operating systems). In the (extinct) language of the Aborigines of the Sydney region, *mungi* means lightning. The pronunciation rules [TM94] for Mungi are as follows:

- m: similar to English,
- u: as in "put" (not like the "u" sound of "but"),
- ng: similar to the "ng" of "sing" (not like the "ng" of "finger"),
- i: as in "bit" (not like the "i" sound in "bight").

3.3 Motivation

Reflecting its name, Mungi is a fast, capability-based operating system that provides a 64-bit, persistent, single address space. Mungi was designed and implemented by the Distributed Systems Group at the School of Computer Science and Engineering within the University of New South Wales, Sydney, Australia.

The design of Mungi was guided by a set of goals, which determined the direction of project.

Simplicity: Mungi should present an application with a "pure" view of the single address space. It is the premise that a huge flat address-space abstraction is sufficient to implement all the needed services, without the need to introduce other name spaces. As a result, traditional kernel services such as I/O and interprocess communication are accomplished through the single, shared, address space. The kernel and all kernel data structures are also part of this address space. Protection also has to be simple and applied in a uniform manner.

Usability: Simple intuitive interface to the services that are provided by Mungi, based on the single-address-space paradigm.

Flexibility: Mungi should provide a simple set of basic abstractions, which can be used to build arbitrary, more complex abstractions. By keeping mechanisms and policy separate, Mungi provides services that applications can use to implement policy.

Protection: As all data in a single address space can be addressed by any process, the lack of address-space boundaries might superficially be viewed as being insecure.

This perception needs to be addressed by ensuring that particular attention is paid to protection in Mungi. The protection system has to be secure and reliable as well as being intuitive and easy to use.

Performance: The performance of any system is nearly always one of the first metrics that is used to compare operating systems. It is of little use having a simple and intuitive abstraction if the costs of execution make it prohibitive.

No reliance on custom hardware: Mungi has to run on standard 64-bit hardware for a variety of reasons:

- The experience of other operating systems projects has shown that projects that involve new hardware find it difficult gaining acceptance (eg. Intel 432 [Org93] and Monads [RA85]).
- Off the shelf hardware is usually less expensive.
- Off the shelf hardware is usually more reliable.
- It is easier to show improvements in performance of an operating system when it is running on known hardware. This allows comparisons to be made with established systems.

3.4 Mungi Abstractions

The principal resource that is provided by Mungi is a 64-bit single address space. This 16 exabyte address space is intended to be large enough to contain all persistent and transient

data in a typical building sized distributed system of thousands of nodes. Operations on the address space are performed through the use of three simple abstractions, which are: *threads*, *objects* and *protection domains*.

Threads: Threads are the execution abstraction in Mungi. A thread is a lightweight instruction stream that travels through the virtual address space. Threads can be created quickly by simply supplying a starting address and a stack pointer.

Objects: Mungi objects are consecutive pages of virtual address space and represent the allocated regions of the Mungi address space. Only addresses that are contained in an object can be legally addressed. All other addresses are considered to be invalid and will cause segmentation exceptions on access. Objects in Mungi have no kernel interpreted type (similar to Unix where everything is a untyped file). There is, however, provision for assigning application specific types to objects in Mungi.

To aid in address-space allocation, Mungi partitions the address space across machines, with each machine being responsible for a region of the address space. Allocations of objects are done from a local partition, and deletes must be forwarded. Mungi places no other significant importance on partitions, in particular they have nothing to do with the actual location of the data.

Protection Domain: Whereas objects in Mungi define the regions of the address space that can be legally addressed, a protection domain defines the regions of the address space can be legally accessed. A protection domain in Mungi logically describes a set of objects and a corresponding set of access rights to those objects. A protection domain can be simply thought of as an operational, active view of the address space. All threads in Mungi execute within a protection domain, which may be shared with other threads. Naturally, any modifications that are made to a shared protection domain are visible immediately by all threads sharing the protection domain.

3.5 System Interface

The operations provided by Mungi to manipulate the three main abstractions define the system call interface. The application are presented with a small set of system calls, which can be roughly grouped into four categories (see Table 3.1).

- Address space management: This is the set of system calls that allows users to create, destroy and change the attributes of objects. Also included in this group are the set of system calls that deal with operations on Mungi virtual memory, such as the mapping and unmapping of pages.

- Protection management: Protection management is the set of system calls than manipulate the kernel data structures that are associated with protection. These system calls, and the Mungi protection system in general, are discussed in more detail in Chapter 4.

- Thread management: Thread related system calls include create, delete and wait calls.
- Miscellaneous: The last category provides support for semaphores and exception handling.

More information on the Mungi Application Programming Interface (API) can be found in [HVER97].

System Calls		
Address Space Management		
ObjCreate	ObjDelete	ObjInfo
ObjNewPager	PageCopy	PageMap
PageUnmap	PageUnalias	PageFlush
Protection Management		
ApdInsert	ApdDelete	ApdLock
ApdGet	ApdFlush	ApdLookup
ApdCreate	ApdRemove	ObjDelPasswd
ObjCrePdx	PdxCall	ObjCrePasswd
Thread Management		
ThreadResume	ThreadCreate	ThreadDelete
ThreadSleep	ThreadWait	ThreadMyId
ThreadInfo		
Miscellaneous		
SemSignal	SemCreate	SemDelete
ExcptReg		

Table 3.1: The Mungi system calls

3.6 Object Table

In order to keep track of objects, all objects in Mungi have an entry in the system wide *object table*. This table contains each object's meta-data, such as date of last access, length and associated protection information, similar to i-nodes in Unix. See Figure 7.7 for the data structure that is used to implement a Object Table node. The Object Table currently implemented as a 256-ary B^+ tree [GBY90].

3.7 Summary

The Mungi single-address-space operating system provides a single, persistent address space. Operation are performed using three simple abstractions; threads, objects, and protection domains. Threads are the execution abstraction, objects are the storage abstraction, and protection domains define the view a thread has on the objects in the system. Using these simple abstractions, applications are able to construct suitable higher-level abstractions as needed.

Chapter 4

Protection in Mungi

The design and implementation of protection in a single-address-space operating system presents a host of new challenges. This is due to the shift away from the separate address space model of protection. Traditional operating systems have relied on separate address spaces to enforce protection.

The crux of the conventional approach is to perform address translation and protection using the same mechanism, namely the memory management hardware. Each process executes in its own address space. Any data that is in the address space of a process has to be explicitly mapped into the address space by the kernel. This may be done by system calls, such as reading a file, or other I/O operations. Protection decisions can be made and applied by the kernel when such a request is made. As a result, only data that can be legitimately accessed can be addressed.

In contrast, in a single-address-space system, any data can be addressed. Single-address-space kernels can not rely on being informed explicitly when a new access is made. Instead implicit methods have to be used to enforce protection in the system. The next chapters will describe different facets of the Mungi protection and security system and the issues that had to be dealt with in the design of the protection system.

4.1 Protection Philosophy

One of the most obvious advantages that single-address-space systems have over separate address space system is the ease with which sharing can be accomplished. Consequently, the design of a protection mechanisms needs to take great care that sharing is not unduly hindered; conversely we cannot ignore protection as this would render the resultant system unsuitable for use in a multiuser environment.

Ideally a protection mechanism should be unobtrusive, offer perfect safety, allow the implementation of arbitrary security policies and have no performance impact on the system. It is obvious that this is an impossible goal. In reality the aims are: a protection system that is flexible, intuitive and has as little performance impact on the user as possible. These requirements form a subset of the Mungi design requirements; in particular the following

subset provided the guiding principles for the design of the Mungi protection system.

- Flexibility: The protection mechanism should not limit the security policies that can be implemented using Mungi primitives. A limited range of security policies restricts the use of the operating system to a specific range of of environments.
- Simplicity: Protection can only be applied to entities that can be named. In the Mungi single address space there is only one name for each entity, its virtual address. This allows Mungi to implement protection simply by applying the "3 R's" (read, write and run).
- Ease of use: Good protection mechanisms don't necessarily translate to good security. If protection is too invasive or counter-intuitive users will not routinely apply it. Therefore, operating system primitives should allow the intuitive application of good security practice. Further, while a naïve user should not need to understand the protection mechanism to apply good security practice, the security conscious user, who does understand the protection mechanism, should be able to arbitrarily enact any security policy that they wish.
- Performance: The need to minimise the performance impact is obvious. Protection is not tolerated if the costs are prohibitive.

4.2 Password Capabilities in Mungi

The need to find a balance between protection and simplicity of sharing is of crucial importance in Mungi. A suitable method of implementation needs to be chosen. On an abstract level, all protection mechanisms are basically variants of access control lists (ACLs) or capabilities.

Although there is no reason why ACL-based protection cannot be implemented efficiently in a SASOS, as Nemesis [Han97] demonstrates, capabilities were chosen as they offer the following advantages.

- Close mirroring of the single-address-space paradigm: One of the features of most capability-based systems is the need for the establishment of a global name space. Fabry [Fab74] states that "*the advantage of a capability used as an address is that its interpretation is context independent*". In order to construct a global name-space, CAP implemented a *Master Resource List* which contained globally valid unique capabilities; Amoeba had globally unique port numbers and The Monash Password Capability System introduced an *Intermediate Address Space* into which all active objects were mapped. A SASOS offers a global name space without the need for another construct.
- Ease of sharing: Capabilities are compact representations of an object, also defining a set of access rights. Using capabilities as special pointers, sharing is trivially accomplished by transferring capabilities. In ACL based systems the sharing of information

would take two operations; one to pass the pointer and one to set up the protection to allow access.

- Good adherence to the principle of least privilege: Capabilities allow the construction of protection domains that contain only objects and rights necessary to perform an operation.

Recall that there are three methods of implementing capabilities: tagged, segregated and sparse. A tagged implementation of capabilities is rejected due to the need for specialised hardware; one of the Mungi objectives was that it be implemented on off-the-shelf hardware. This leaves two possible implementations, which can be classified as kernel controlled or user controlled capabilities. The kernel controlled (segregated) method needs the kernel to be invoked for every passing, comparing or storing of a capability, an unacceptable overhead in any system that has a strong commitment to providing support for sharing. It is for this reason that Mungi protection is implemented using password capabilities [VRH93]. The other implementations of sparse capability were rejected due to the need for encryption which is computationally expensive. Sparse capabilities in general, but particularly password capabilities, have the advantages that are listed below.

- Password capabilities allow the passing of capabilities and thus the sharing of objects to be performed without needing kernel intervention. Mungi also supports a user library scheme that allows for capability refinement to be achieved without the intervention of the kernel (see Section 6.2.4).

- Password capabilities allow data and capabilities to co-exist. This is of great advantage [Jon80], as it allows capabilities to be stored in arbitrary data structures.

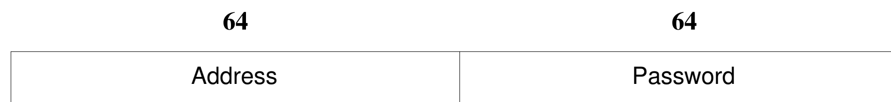

Figure 4.1: Mungi password capability

4.2.1 Capability types and access modes

Mungi capabilities (Figure 4.1) are 128-bit entities, consisting of a password and the base address of an object. Each password conveys a set of rights to the object. The access rights that are supported by Mungi are *Read, Write, Execute, Destroy* and *PDX* (for an explanation of PDX see Chapter 5). A capability that contains all of the *DRWX* rights is called an *owner capability*[1].

[1]Note that execute-only rights only make sense on CPUs that support an execute-only mode of operation. In a single-address-space, without further hardware support, if an object can be executed then it can be read and vice-versa. This is a side effect of the fact that execution of a program is as simple as doing a jump to the required start instruction and requires no kernel involvement.

One of the weaknesses of capability based systems is that they can only specify the right to access an object; capabilities normally do not have the power to explicitly reject access to an object. The Mungi protection system does provide the mechanism to explicitly deny rights by having negative versions of all of the above rights, that is, a capability that explicitly denies certain rights for an object. For these capabilities to be useful, it must be possible to force them to reside in a protection domain. Mungi provides a mechanism called *slot locking* (see Section 4.3.6) that achieves this.

4.2.2 Object table entries

All objects in Mungi have an entry in the global object table. This entry contains object information, part of which is a list of <password,rights> pairs. This list represents all of the valid passwords for the object and their associated access rights. Any valid capability for an object must consist of the base address of the object and one of the passwords in this list. The access granted by such a capability is determined by the set of rights that is associated with the password in the list (see Figure 4.2).

Any holder of an owner capability for an object is able to add or delete entries from the password list for that object. Note that it is possible to have more than one capability that grants a certain set of rights; in fact, it is possible to create a different capability for every entity that a capability is passed to, making it possible to selectively revoke access rights.

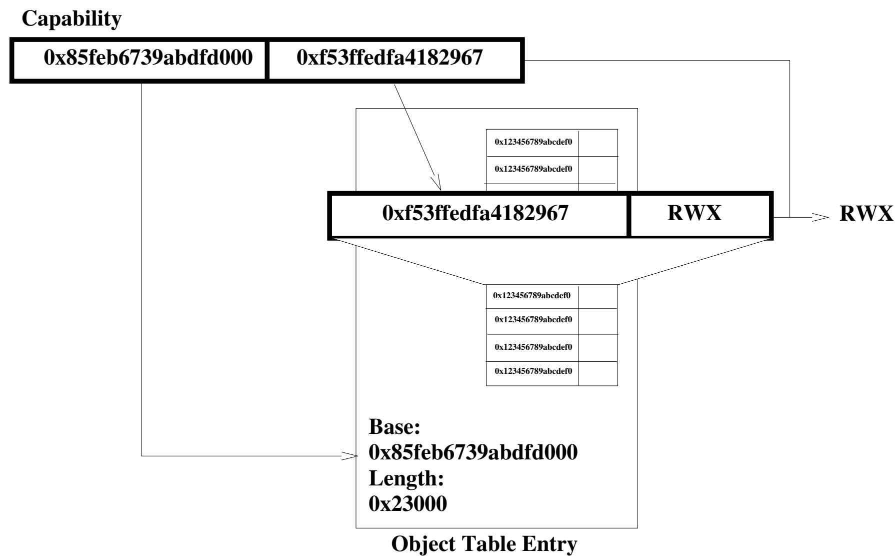

Figure 4.2: A valid RWX capability

4.2.3 Password management

When an object is created in Mungi, a password is specified which, together with the base address of the object, forms the new owner capability for that object. Using this

capability it is then possible to create new capabilities for that object (including additional owner capabilities). The system calls that implement password management are given in Table 4.1 and are described below.

Syscalls		
ObjCreate	*(size,password,info)*	→ (owner_cap)
ObjCrePasswd	*(address,password,mode)*	→ (cap)
ObjDelPasswd	*(address,password)*	

Table 4.1: Object system calls

ObjCreate: The object creation call. When a object is created the kernel allocates an entry in the object table for that object. After this, a region of the address space is reserved for the object, the base address of this region being the base address of the object. The password that is given as a parameter to the ObjCreate call is entered into the object table as representing the owner capability for that object, this capability is the return value of the system call.

ObjCrePasswd: Mungi imposes no limit on the number of valid capabilities for an object. The ObjCrePasswd system call allows any holder of an owner capability to create another capability for that object by adding it to the list of valid capabilities in the Object Table. The arguments to this system call are the password for the new capabilities together with the set of access rights that the capability is to confer.

ObjDelPassword: The opposite operation to creating a new capability is to delete or revoke a certain capability. In Mungi the only way to revoke a capability is to make a call the ObjDelPassword system call which will remove the entry for that capability from the object table. The revocation is not immediate as Mungi performs extensive caching of validations (see Sections 4.3.5 and 6.1.1). By having more then one capability that confers a certain set of access rights, a form of selective revocation can achieved.

4.3 Protection Domains

4.3.1 Implicit validation

The fact that capabilities are not maintained by the kernel but by the users themselves raises the obvious question: "How does a user present the kernel with a valid capability when an access is made?" Generally, there are two ways of approaching this problem: one is to make the user explicitly present a capability before the object that it refers to can be accessed, the second is an implicit presentation. The implicit method requires the kernel to find the required capability when an access is made. Placing the onus on the user to make the appropriate explicit validations before access is made is rejected as being too invasive.

Instead the Mungi kernel provides a mechanism that attempts to find a valid capability when an access is made, implicitly validating the access.

To provide the necessary support for implicit validation of capabilities, Mungi supports two main data structures: the *capability list* (Clist), and the *active protection domain* (APD).

4.3.2 Clists

As the name suggests, a Clist is simply a list of capabilities, usually grouped logically, for example all the capabilities that pertain to a certain project. Clists are Mungi objects that are owned by users, and are thus also able to be shared without the intervention of the system. The kernel recognises two types of Clist, ordered and unordered. The type of the Clist determines the search strategy that the kernel will employ when searching the list. Ordered lists are searched using a binary search, while unordered lists are linearly searched. As users control the Clists they can misinform the kernel by having Clists that are marked as being ordered, but are in fact not. The security of the system does not rely of the fact that this flag is correct or that the list is properly ordered. An inconsistency will only affect the users using that Clist.

4.3.3 Active protection domains

Active protection domains are the top level structure that the kernel uses to maintain a protection domain. Physically, an APD is just an array of capabilities (see Figure 4.3), which capabilities are pointers to Clists. In this way, a protection domain is logically a two-level hierarchy consisting of a list of pointers to lists of capabilities. Although an APD is just an object in the address space, the kernel makes sure that the only valid capability for any APD is the execute only capability. This capability grants the holder the permission to start threads in this protection domain, as well as the rights to modify the APD through the use of the system calls in Table 4.2.

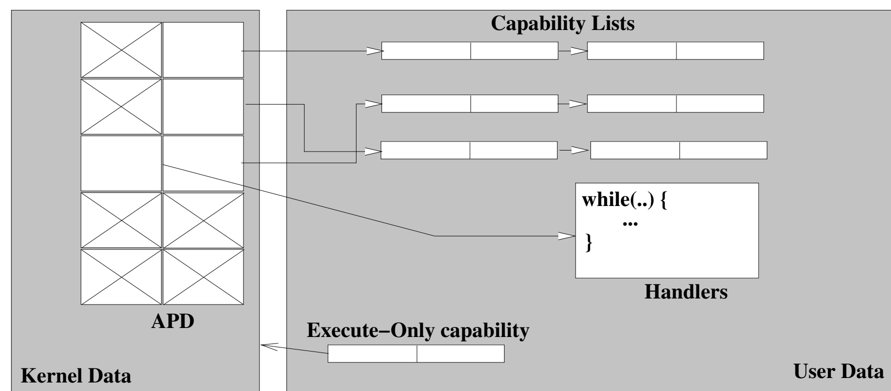

Figure 4.3: A protection domain

A suitably authorised user is able to add Clist pointers (capabilities) to an APD, but once added to the APD, only the address half of the capabilities can be retrieved. This ensures that a user who has permission to operate on a protection domain does not necessarily have the rights to access the Clists which constitute that domain.

4.3.4 Access validation

With the aid of the above two data structures, the kernel attempts to implicitly validate all accesses made by a thread. When a thread touches a page for the first time, the kernel pager is invoked (see Section 7.4.1 for more details). The pager searches the object table to find the set of valid passwords and access rights for the object that contains the faulting address. If the address does not fall within a valid object, a segmentation exception if raised. Once the pager has a list of valid passwords, the protection domain of the thread is searched for a suitable capability. This is done by searching the Clists, pointed to by the APD of the thread, one by one in the order that they appear in the APD. If a valid capability is found, the access rights of the capability are compared with the attempted access. If the rights provide sufficient privileges to perform the attempted access, the validation is complete. If the rights insufficient, the search of the protection domain continues. If no valid capability is found that confers the necessary rights, a protection exception is raised.

4.3.5 Caching access rights

To prevent the relatively expensive validation procedure from being performed with every new page that is accessed by a thread, the kernel maintains a small cache of validations for each protection domain. This cache is called the *validations cache* (vcache). The vcache is searched before the Object Table is searched and a hit in the vcache terminates the validation successfully, while a miss continues the normal validation path. When the first page of an object is accessed, the validation for the object is added to the vcache and thus all the other pages of the object are implicitly also in the validations cache. When the next page of the object is touched, only the validation cache needs to be searched. This cache is flushed at regular intervals to ensure revocations become effective eventually. The vcache represents that kernel's internal representation of a tasks protection domain.

4.3.6 Tailoring protection domains

The need to allow users to modify their protection domains is obvious. Operations like reducing a protection domain before making a call to an untrusted procedure, adding new objects and removing old objects are a few examples where modification is required. In Mungi there are two ways to make modifications to a protection domain.

- One is to add capabilities to Clists. This is mainly useful for sharing single objects and can be done without the intervention of the kernel.

- The other way is to modify the APD, which is the kernel's representation of a protection domain. A suitable authorised thread is able to modify its APD with the system calls in Table 4.2.

Syscalls		
ApdCreate	*(n_caps, handler_flag_string, ...)*	→ (*cap*)
ApdRemove	*(addr)*	
ApdInsert	*(addr, pos, h_or_cl_ptr, is_handler)*	
ApdDelete	*(addr, pos)*	
ApdGet	*(addr)*	→ ({*cptr,hptr*}[*n_apd*], *locked*)
ApdFlush	*(addr)*	
ApdLookup	*(addr, address,mode)*	→ (*addr*)
ApdLock	*(addr, pos)*	

Table 4.2: APD system calls

All the operations that are described in Table 4.2 (except ApdCreate) require the authority to execute in that domain (namely the execute-only capability for the APD itself). The argument that is passed to the functions is the base address of the APD, which uniquely identifies the APD.

ApdCreate: This is the call that will create a APD. The capabilities that are specified as arguments are inserted into a new APD, an execute only capability for which is returned to the caller. As noted above, once the Clist capabilities are inserted into the APD, only the address part of the capability can be retrieved.

ApdRemove: Destroys an APD. This is analogous to destroying the object that the APD exists in.

ApdInsert: This call adds the Clist pointer to the APD that is specified by `addr`. The capability is validated before it is inserted into the APD, as a result all of the pointers in the APD have been validated.[2] The rest of the pointers after pos are shifted down a slot.

ApdDelete: Delete a Clist or handler pointer from the APD, this has the affect of reducing the protection domain. This call will result in the entire contents of the page-table and the validation cache being flushed.

ApdGet: Returns the list of Clist and handler pointers that are currently in the APD of the calling thread. Note that only the addresses of the Clists are returned, not the actual capabilities for those lists.

ApdFlush: Flush the validation cache and revalidate the contents of the APD. This call will result in the flushing of the entire protection domain of the caller.

[2]This does not mean that they are currently valid, as they may have been revoked since.

ApdLookup: Find the first capability in the protection domain, if any, that confers at least the given rights to the address specified. The return value is the address of the capability, allowing a user with read access to the particular Clist to find the actual capability. If a valid capability is found this call also has the effect of adding the validation to the validation cache of the protection domain.

ApdLock: This system call will lock an entry in the specified APD. The result of locking an entry is that threads are no longer able to change that entry, even with the authority of an execute-only capability for the APD. The result is that the particular Clist is forced to reside in that protection domain.

4.4 Thread Creation

In Section 3.4 thread creation was briefly mentioned. There are two specific kinds of thread creation: threads can either be created in the protection domain of their parent or in a specific protection domain. For a thread to be able to start a thread in another protection domain, the parent thread has to have an execute capability on the APD.

```
ThreadCreate(void * ip, void * sp, APD_t * pd)
```

4.5 Capability Handlers

It is envisaged that a typical protection domain will consist of thousands of capabilities. The kernel is charged with the responsibility of finding the appropriate capability when an access is made. Even with a binary search, blindly traversing a task's protection domain in search of a capability can be expensive. Certain applications may have a good idea of their access patterns and the capabilities that they will need to do so. For these applications Mungi provides *capability handlers*. A capability handler can be associated with each Clist slot in the APD. During validation, when a handler slot in the APD is non-empty, the kernel will perform an up-call to the capability handler. The job of the handler is to add a valid capability for the access to the Clist that it is associated with. Once the capability handler performs, the Mungi kernel will restart the validation and expect to find a valid capability in the Clist associated with the handler. If a capability is not found in this Clist validation proceeds to the next APD slot (see Figure 4.4 for details).

A capability handler is provided with the address of the fault and the access that is required on that address. Using capability handlers, applications are free to implement more sophisticated data structures for the traversal of a protection domain.

4.6 Protection Hardware

Single-address-space operating systems are forced to apply security policy to every access to memory. Memory management hardware (MMU), in traditional operating systems, has been used to provide both protection and address translation.

```
search_apd(apd,           /* the apd to search     */
           tid,           /* thread id of faulter  */
           object,        /* the accessed object   */
           rights)        /* the rights attempted  */
{
inh = false
<start,inh> = inhandler(tid)
for i=start to napd
{
  if (apd[i]->hptr &&  !inh)       /* is there a handler  */
  {
    inhandler = <tid,i,true>   /* remember this point */
    call apd[i]->hptr(object,rights)   /* call the handler    */
  }
  inh = false;
  if (apd[i]->cptr)       /* is there a clist    */
    for j = 0 to apd[i]->cptr->ncaps
      if (validcap(apd[i]->cptr[j], object, rights)
        return true
}
return false;
}
```

Figure 4.4: Validation

In a SASOS all address translations are the same, so translation information remains valid across context switches. However, in order to be able to control access to data, a SASOS is forced to use the memory management hardware, specifically the translation look-aside buffer (TLB), to implement protection rather then to cache translations. With a tagged TLB, this has the unfortunate side affect that there might still be multiple entries for a virtual address, whose translation information is the same, but access permissions are not. This takes up valuable TLB space. With an un-tagged TLB, the situation is even worse, as translation information has to be flushed with each context switch, due to changes in protection information. Ideally, hardware support for a SASOS would include a separate protection cache. This was implemented in some of the earlier capability systems that designed their own hardware. IBM's System/38 [HSH81] has a two-way associative 64-entry look-aside buffer for caching capability translations, called a *segment look-aside buffer* (SLB). CAP's capability unit contained a slave memory that held 64 *evaluated capabilities* [NW77]. A *protection look-aside buffer* (PLB), similar to a SLB, has also recently been suggested by Koldinger et. al. [KCE92]. The PLB would cache the protection information while the TLB would contain only translation information. A PLB would not have to be as large as the TLB to be effective, as it can cache protection information over regions that are larger then a page, for instance a set of contiguous pages that represent an object. This would obviously decrease the number of entries that would need to be in the PLB to make it effective. Having a PLB would also allow the use of an untagged TLB.

4.7 Related Work

Like most other operating system projects, the ideas that are presented in Mungi are a mix of old and new. The work of other projects has contributed in two ways: while some previous ideas have been used, other have influenced Mungi to move in the opposite direction.

Mungi presents a flat protection name space. This idea was used by Multics and Unix, which applied uniform access protection to the file system. Mach also provided a uniform protection name space by performing operations on ports. Single-address-based system can also use the protection name space for addressing, allowing a simple model due to the lack of need of another abstraction. Opal, while being a single address space operating system, introduces another name space for its portals.

The password capabilities in Mungi are most like those presented by Anderson, Pose and Wallace in their password capability system, and used by Opal. The capabilities in Amoeba are also similar, but with two main differences: capability rights are interpreted by servers rather than by the system and there is no uniform set of rights. This is also true for the capabilities in Eden, with rights being interpreted by ejects. Although not truly a sparse capability system, Eden allows users to keep a shadow copy of capabilities in user-space. It is believed that the simple 3 Rs (Read, wRite and Run) philosophy of Mungi is a simpler and more consistent approach to protection.

Sparse capability system, with capabilities in user-space have to be able to pass capabilities to the kernel for validation. Mungi's capabilities are implicitly validated. Other system like Opal, Amoeba and the Monash Password Capability System all require the user to explicitly present a capability to the kernel before access. This is rejected as being far too invasive to the users of the system. Only Opal also provides supports implicit validation, by registering capabilities with a name server that is queried when there has been no explicit presentation. The disadvantage of this is that it has the effect of bringing user-level naming into the kernel. In Mungi naming is treated purely as a user-level issue, to allow greater flexibility.

Validation of capabilities can be an expensive operation, as potentially large lists have to be searched. To overcome this many systems have employed hardware support to cache validations and thus reduce the cost of validation (see Section 4.6). Hardware support is not available on current 64-bit CPUs and thus Mungi relies on software support for caching. The password capability system in [APW86] also supports the caching of validation information in software by using an active object table (AOT) which is a main memory cache of validations.

4.8 Summary

The protection mechanism that is provided by Mungi is based on password capabilities, which consist of the address of an object and a 64–bit random number. The advantage of password capabilities is that they are not controlled by the kernel, this allows users to store capabilities in data structures, and supports the passing of capabilities between protection domains without the intervention of the kernel. Password capabilities are protected from forgery by the sparseness of the 64–bit password field. Validation of access is done implicitly by the kernel, removing the need for users to worry about explicit presentation.

The protection mechanism provided by Mungi allow security conscious users to arbitrarily tailor their protection domains, while providing the ordinary user with operations

that intuitively apply good security practice.

Chapter 5

Protection Domain Extension

> ***Extensible operating systems are designed around the principle that the service and performance requirements of all applications cannot be met in advance by any operating system.***
>
> **Stefan Savage and Brian Bershad, 1994**

5.1 Motivation

Extensibility and flexibility are goals that should be part of any operating system design. An operating system that is extensible is able to cope with the changing demands of applications. Extensibility is the ability to add services to a system in a controlled manner. The mechanisms for adding new services to a system can be grouped into two main classes: static and dynamic.

Mungi allows user modules to be added to the system environment [VERH96]. These modules, when invoked, have the power to temporarily extend the protection domain of the caller, hence the name protection domain extension (PDX). The primary motivation for PDX is to support extensibility. As will be discussed later, it is also the method for safe object invocation in Mungi.

5.2 Extensibility

Early operating systems recognised the need to be be adaptable. Systems like Hydra and Pilot were designed to be able to support a wide range of operating system personalities. This was done through the separation of mechanisms from policy; recognising that the system supplies the mechanism, while policy was implemented at a higher level. Using the mechanisms to add new policies involved static addition of services to the kernel. Adding

these services forces users to access the them through a strict and well defined interface, namely the kernel system call interface.

Dynamic extensibility provides a mechanism to add services to the system at any time. Hydra's amplification templates and Unix's set-user-id addressed the problem by introducing special programs that, when executed, would run with an increased set of access rights. This allows the program to perform privileged operations on behalf of users.

Another approach to extensibility is the use of μkernels and user-level servers. New services can be added via additional servers, with the main role of the kernel being to support communication between clients and servers.

The implementation of extensibility in these systems creates some of the following drawbacks:

- Adding services to the kernel makes the kernel large and error prone.
- Any additional service will execute with the same rights as the kernel, this is obviously a breach of the principle of least privilege.
- Static addition of services requires the kernel to be recompiled whenever new services are required. This is a major disincentive to add new services.
- While the above problems are avoided by mircokernels, early μkernel-based approaches suffered from performance problems blamed on the added overhead of switching between multiple address spaces [CB93].

Current research [OSD94] in extensibility focuses on the performance and protection of extensions. The goal is to execute extensions in the same domain as either the kernel or the user, which results in less context switches when the services are called. The two main approaches have been investigated in order to achieve this goal.

- User level: This approach adds services at user level, making sure that the kernel provides the mechanisms to be able to enforce some level of protection. Examples include Cache Kernel [CD94], Exokernel [EKO95], Pilot [RDH+80], L4/L3 [Lie95].
- Kernel level: Adding services to the kernel in a safe and secure way. Examples include SPIN's kernel loadable modules [BSP+95], and software fault isolation techniques [WLAG93, SESS96].

The next sections will examine these approaches in more detail.

5.2.1 Exokernel

The Exokernel [EKO95] approach is to provide as many services as possible in the user space. The design philosophy takes the approach that a kernel should not provide *any* abstractions. Instead, the only role of the kernel is to safely multiplex access to the hardware. All other services are constructed by using three main mechanisms.

- **Secure binding:** A process can bind to a machine resource. The kernel then provides a capability that can be used to access the resource.
- **Visible revocation:** When the system wishes to revoke access to a resource, a process will be notified, allowing it to cleanly release the binding.
- **Abort protocol:** If a process does not want to give up a resource, the kernel can forcibly reclaim the resource.

Essentially Exokernel brings hardware up to the user, reducing switches to the kernel, due to the fact that most code will simply run in user space.

5.2.2 SPIN

The SPIN [BSP+95] project at the University of Washington provides a kernel that is dynamically extensible by adding kernel modules. Kernel modules have to be written in a type-safe language (Modula-3 [Nel91]), and a special kernel compiler then generates code that can run in kernel mode without compromising the safety of the kernel. SPIN relies on four main features to achieve performance and extensibility.

Co-location: Communication between modules and the kernel is low cost since they reside in the same address space. Operations that need high levels of kernel communication gain the most benefit.

Enforced Modularity: Using Modula-3, enforced typechecking and module interface boundaries allow code to run safely in kernel mode. These restrictions makes sure that each module can only access authorised memory locations.

Logical Protection Domain: Each module exists within a logical protection domain. A kernel dynamic linker resolves references at run-time, to ensure the module will be restricted to its protection domain even though it is running in kernel mode.

Dynamic Call Binding: Extensions are bound to system events such as traps and page faults. This ensures that the extensions are invoked in a controlled manner and through the correct interface.

5.2.3 Software fault isolation

The second approach to add modules to the kernel is to make sure that all unauthorised references made by the module to the kernel's address space are caught and disallowed. To achieve this, each module is placed in its own *fault domain*. The software is then modified so that it cannot access memory outside its fault domain. There are two main approaches that achieve this result, both of which can be used with any programming language, removing the requirement to be restricted to a specific language such as Modula-3.

Segment matching statically determines if an instruction is going to access memory that it is not allowed to. If this is the case then it can be dealt with at compile time. Instructions

that rely on register offsets cannot be examined in this way, as they depend on runtime variables. These instructions have code placed before them to do range checking, which calls an external exception handler if an access outside the fault domain is made.

The other method is *address sand-boxing*. In this approach, all unsafe instructions have code inserted before them that sets the upper bits of the address to be the same as that of the module. In this way, it is not possible to access memory locations that are outside the fault domain. The system that is presented in [WLAG93] claims that, while the overhead of the sand-boxing approach is 4%, this is offset by the reduction in the number of context switches.

5.3 Mungi PDX

The aim of the research outlined above is to minimise the number of domain crossings. The advantages of introducing services in user space is that a context switch to the kernel might be avoided altogether. Introducing services into the kernel allows them to have access to the kernel's internal variables, but the need to protect the kernel comes at a cost; either the code for an extension is slowed down by protection checking, or it is forced to be written in a type safe language[1].

The Mungi single address space allows all resources to be addressed. The protection system uses capabilities to control access to these resources, without requiring additional compiler support or restricting the choice of languages. In Mungi the issue is how to control access to these capabilities. The approach taken in Mungi is to allow threads to extend their protection domain in controlled fashion using the PDX mechanism.

5.3.1 PDX

Mungi's PDX mechanism allows the extension of a thread's protection domain for the duration of a procedure call. A PDX module is an object containing a number of procedures and an associated Clist. This Clist is added to the caller's protection domain when invoking any of these procedures using the PDX system call. Mungi allows entry into the module only at the start address of each procedure. These entry points can be individually protected by password capabilities, allowing selective access to the procedures.

The system call

PdxCall(Cap(*proc)(Cap), Cap param, uint npd, . . .)

is used to initiate a PdxCall which executes the procedure `proc`. `param` is a capability that is passed to the procedure. If needed this capability can be used to refer to a buffer that is used to supply additional parameters to the PDX procedure. Note that the `param` parameter to the PdxCall isn't necessarily a capability, but the format allows two 64-bit words. The remaining parameters to PDX call are used to construct the desired protection

[1] Most modern systems avoid both these costs by allowing unchecked extensions to be dynamically loaded into the kernel, this is at the expense of protection.

domain. The caller can provide a set of Clist capabilities that specifies the selected subset of the caller's protection domain that is passed to the procedure (see Figure 5.1). The procedure will then run in a protection domain which is defined as the union of the Clists passed by the caller and the Clist associated with the PDX procedure. When the PDX procedure finishes, control is passed back to the caller after the caller's protection domain has been restored.

PdxCall offers three different ways of constructing the desired protection domain, depending on the `npd` parameter. If `npd` is greater than zero, the protection domain is as described above. If `npd` is -1, the caller's whole protection domain is merged with the protection domain registered for the PDX, effectively *extending* the caller's protection domain (hence the name PDX). Lastly, in `npd` is 0, the protection domain consist solely of the Clist provided by the PDX module (see Table 5.1). It is envisaged that the most common modes of operation will be either passing the whole protection domain or none.

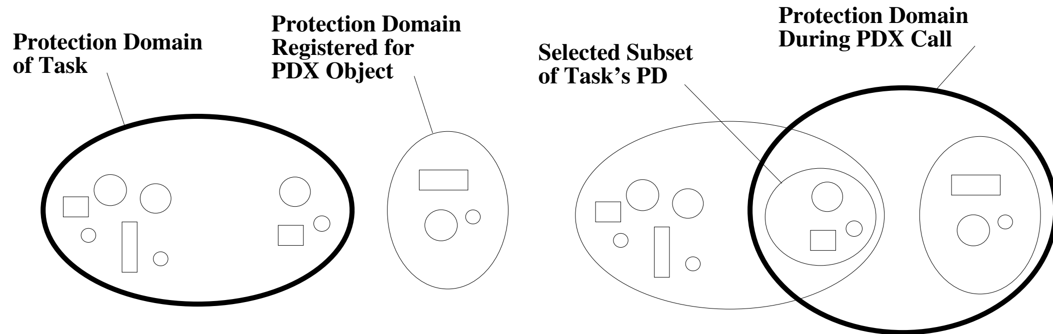

Figure 5.1: Protection domains before (left) and during (right) a PDX Call.

`npd`	Result	Use
0	empty PD passed	device drivers, page fault handlers
-1	full PD passed	"true" PDX call
> 0	selected Clists passed	untrusted PDX call

Table 5.1: PdxCall parameters

Validation of object access in the Mungi system requires the searching of two large data structures; the object table and the user's protection domain. The object table is a distributed data structure, which may add to the cost of access. To amortise some of the validation costs, much of the validation information is cached. Implementing protected procedure calls based on an extension of the caller's protection domain has two main benefits: firstly we can re-use the cached validation information from the caller's protection domain, and secondly the extension allows for the implicit sharing of large numbers of objects between the caller and the protected procedure.

5.4 Device drivers

Mungi is committed to presenting a pure view of the single address space. This is achieved by omitting from the model anything which would introduce any other forms of name spaces. For example, there is no I/O model; clients which require explicit control over I/O, such as database systems, can achieve this via virtual memory operations [ERHL96], as devices are memory mapped. The control registers of devices, as well as the contents of disk drives and physical memory, are mapped into the virtual memory. The device driver is given the capability to read and write the appropriate memory regions, and users can then safely request operations from the device driver by invoking it via a PdxCall.

The interface to a device PDX module must take into account the following two conditions.

- A capability for an operation should remain valid even when the version of the device driver changes. In particular, the address of the device driver must not change over time.

- Knowing the base address of the device driver should be sufficient to be able to deduce the addresses of all the operations that can be performed on the device.

The Mungi device model is defined by the Mungi Device Interface (MDI) standard which is similar to the Unix model. The MDI includes open, close, read, write and init primitives which are common for all devices. Device drivers have a jump table starting at the base address of the driver. Users are able to index into the table (see Figure 5.2). Following the set of common operations, there is room reserved for special operations that are specific to the device. As Mungi individually protects entry points into a PDX module, users can be individually restricted to specific operations on the device.

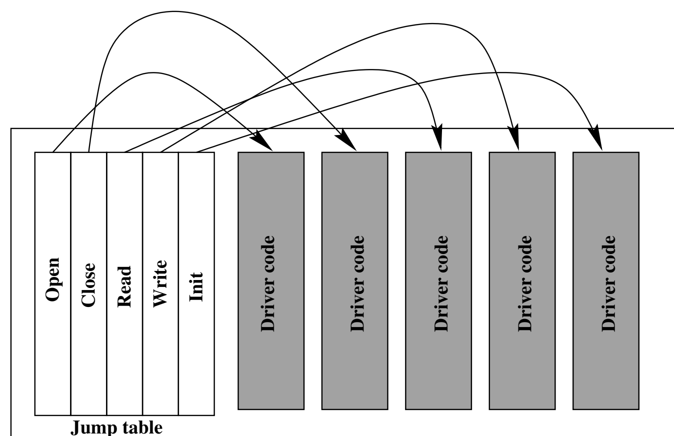

Figure 5.2: Generic Mungi device driver

5.4.1 Case Study: Serial Driver

The current version of Mungi only has one device driver implemented. This driver is the serial port driver, and is important since it the sole means of communication with Mungi at the moment.

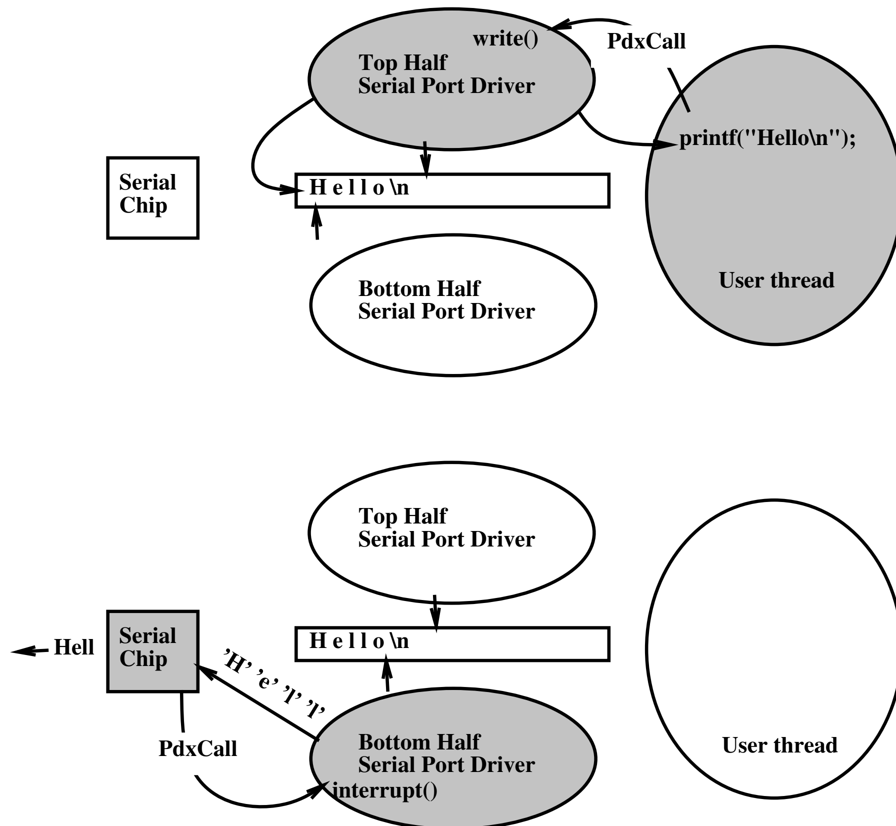

Figure 5.3: Serial port driver top/bottom half interaction

To construct the serial port device driver, a PDX object, that has access to the part of memory which maps the device driver registers, is created. In the case of the Z85230 [Zil92] serial chip on the U4600 boards used for prototyping Mungi, this is the page starting at address 0x1c800000. The serial port device driver is given exclusive read and write permissions to that object (by having the read/write capability as part of its extended protection domain). Since the driver has the only copy of this capability, users are prevented from writing directly to the serial port registers. A call to printf (see code below) will call vnsprintf to build the final buffer that needs to be printed. Printf then calls the PDX serial driver, with the capability for the print buffer as one of the arguments to the PdxCall. The serial driver, once invoked, will then copy this buffer to an output buffer and return to the caller.

```
#define WRITE 0xb08e0 /* eventually this will be resolved */
                      /* by a naming service              */

void
printf(const char *fmt, ...)
{

  char pbuffer[256];
  static Cap_t pobject = {0,0};
  va_list args;
  int r;

  /* Format the print string */

  va_start(args,fmt);
  r = vnsprintf(pbuffer,fmt,args,256);
  va_end(args);

  /* Have we already found the cap for this buffer ? */

  if (!pobject.address) {
    pobject = ApdLookup(pbuffer,R_Read,0);
    pobject.address = (void *) pbuffer;
  }

   /* Call the top half of the device driver, passing */
   /* the print buffer along as the parameter         */


  PdxCall(WRITE,pobject,0);      /* write() */
}
```

The code fragment below shows the part of the driver that deals with interrupts from the serial chip. The Z85230 chip has a 4 character output FIFO and can be programmed to send an interrupt when the FIFO is empty. When the kernel receives this interrupt it makes a PdxCall to the interrupt handler. This then reads the next four characters from the print buffer and waits for the next interrupt (for interaction between the two halves see Figure 5.3). If there are no further character to be printed the handler makes sure that the transmit interrupt is reset.

```
void
interrupt()
{

  char c;
  char intcode;
  uint32 i;


  intcode = zsccgetreg (portb, zRR2A);


  switch (intcode & INTMASK)
    {
    case zRR2_ARXA:        /* character available */

    /* deleted */

    case zRR2_ATXE:        /* transimit FIFO empty */
      if (w_buff_size > 0) /* user output data available */
        {
          for(i=0;i<4 && w_buff_size > 0;i++,w_buff_tail++,w_buff_size--)
            porta->udata = w_buffer[w_buff_tail]; /* send data to chip */
        }
      else  /* turn of transmit interrupt */
        {
          zsccputreg(porta, zWR0, zWR0_RESETTXINT);
        }

      break;
    }

}
```

Similarly, the Z85230 chip will generate an interrupt when there is an input character available. The kernel deals with this interrupt by again making a PdxCall to the serial driver. This will copy the character into a input buffer that can then be retrieved by a PdxCall to a read function.

The serial port driver currently only exports **read, write** and **initialise** operations, with the users only having access to the read and write operations.

5.5 User-level page fault handlers

User level pagers (ULPs) are essential for efficiently supporting databases and implementing persistence in Mungi [ERHL96]. A ULP is a PDX procedure which is invoked by the kernel when a page fault occurs on an object for which that ULP had been registered. ULP invocation uses an zero `npd` parameter. Hence, the ULP runs within just the protection domain which was registered for it. As the ULP has no access to either the kernel's or the faulting thread's protection domain, it does not need to be trusted. This is important, as it allows a thread to access any object without the possibility that a pager associated with that object would be able to access the threads protection domain.[2]

A single ULP can handle a large number of objects. Furthermore, as the ULP is invoked by the kernel and is passed an empty protection domain, all clients of a particular ULP can share the same cached PDX protection domain. This limits the number of ULP protection domains that need to be cached to one per ULP in actual use.

5.6 Object Support

The *NOM* object system on the IBM AS/400 [MM96] has demonstrated that it is possible to build an object oriented system on top of abstractions like those provided by Mungi. Here we show how Mungi can enforce encapsulation and support inheritance.

5.6.1 Encapsulation

Encapsulation can be enforced by the protection system if the provider of an object never hands out read, write, or execute capabilities to the object. Instead, a PDX procedure is provided which, when invoked, extends the caller's protection domain by the appropriate capabilities to the object. Clients can thus only operate on the object by invoking this procedure. The PDX procedure code can actually be part of the object, or it can be separate.

5.6.2 Inheritance

To implement inheritance, jump tables used to access virtual methods are associated with the PDX objects. Potentially, these jumps are further PDX calls to methods of other classes.

[2]Note, however, that the ULP can still interfere with the client's operation by denying service.

This can lead to a proliferation of cached PDX invocations.

A reduction of this overhead is possible if there is some trust between the classes (as there is likely to be if they are part of the same library). The derived class can then be given the capability to execute the superclass methods directly, i.e. by a normal procedure call.

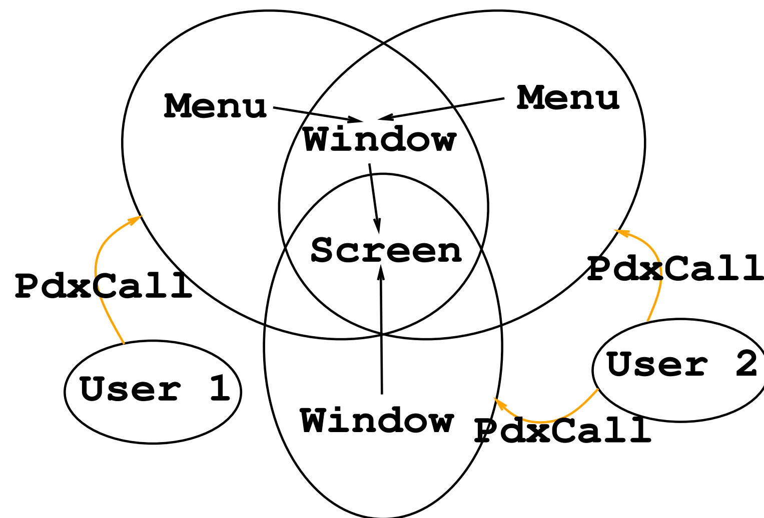

Figure 5.4: Inheritance

An example of this is given in Figure 5.4. There are three classes in this example: `Menu`, which is derived from `Window` which is derived from `Screen`. If user 1 invokes `Menu` and user 2 invokes `Menu` as well as `Window`, a total of eight PDX protection domains would need to be cached. However, if the various classes in the hierarchy trust each other, invocation of a superclass method by a subclass is by a normal procedure call, and only 3 PDX protection domains need to be cached.

5.7 Other Services

PDX is intended as a replacement for the "client-server" model. In Mungi users are able to write PDX modules that allow controlled access to information that they hold. The simple procedure call interface and object granularity protection make PDX an easy to use general tool for users. Unlike IPC-based systems, users don't have to worry about the marshalling of arguments before sending a request to a server. PDX can be used for services that are usually set-user-id root in Unix, such as:

- sendmail,
- printer spoolers, and

- graphics server like X [X90] or $8\frac{1}{2}$ [Pik91].

5.7.1 Parameter passing

A **PdxCall** system call provides a simple parameter passing mechanism. PdxCall allows for 128-bits of information to be passed directly to the called PDX module. In the case where more than 128-bits need to be passed, the parameter is intended to be used as a capability pointer to a parameter buffer. This buffer can then be used to pass information back and forth between the caller and the PDX procedure. In practice, this can be done by calling a stub that stores the arguments into a new object, which is then used as a parameter buffer. In the case where the PDX to be called is simply an extension of the caller's protection domain, all we have to do is call a stub that passes a pointer to the area on the stack where the parameters are stored to the PDX module (see Pdxdummy in Figure 5.5). All the PDX module has to do is to re-create the argument list.

5.7.2 Case Study: OO1

A metric of the usability of a new protection paradigm is how easy is it to adapt an existing application to take advantage of new protection features. Many existing applications are large, complex systems with little or no internal protection. Mungi makes it relatively simple to convert such an existing non-protected system into one that has stronger, internal protection by using the PdxCall mechanism.

One of the methods that was used to measure the performance of Mungi's protection system was an adaptation of the OO1 object-database benchmark (see Section 8.2.3 for more details). This benchmark was initially implemented as a set of C functions with the client code invoking the database operations (**db_init, db_print, db_insert, db_delete, db_connect, db_lookup and db_traverse**) via ordinary function calls.

Logically, it was desired that only the database operations had access to the database data. This required the database to be implemented as a PDX module that had access to the database. The porting task was to split the database operations from the user operations, both physically into different files, and logically in protection domains. The task of the physical separation was trivial as the source code for the data-base benchmark was already split into a front end (client) and a database section. All the calls to the front end of the object database were `#defined` to use a stub which called the function via a PdxCall. This stub passes the address of the arguments to the PdxCall (see the front end section of Figure 5.5). The modifications to the database side proved a little more complex, but only due to compiler conventions. The database side required a stub that explicitly removed all the arguments from the buffer supplied by the front-end and copied them to the stack of the database function. The **db_connect_stub** in Figure 5.5 uses `marg_get` macro to achieve this.

It would be relatively simple to hide the PDX mechanism behind a C++ like interface. The compiler can specify the module addresses and allow simple access to the methods in a PDX module. Arguments that are passed from caller to a method have to be put into

a buffer rather then passed on the stack; similarly, the called method takes its arguments from the buffer rather then from the stack. This simple modification is a future project, but would allow users to take advantage of Mungi PDX without having to pay the cost of having to worry about the implementation details.

5.8 Summary

Extensibility is one of the main features of Mungi. Mungi exports a single address space that is the basis of everything else in the system. Devices are memory mapped and protected from unauthorised access in the same way that the rest of the address space is protected; that is, by the use of capabilities. Mungi provides a mechanism for users to create protected services called protection domain extension (PDX). This mechanism provides for a temporary extension of the protection domain of the caller only while executing trusted code. PDX calls are cached so as to amortise the initial cost of set-up (see Section 8.2 for performance figures). Using PDX, Mungi provides a flexible, extensible system that can meet the requirements of a large variety of applications and users.

```
/*
 *  simple stdarg-type macros, not very portable
 */

typedef int64 *m_arg;

#define marg_start(buffer,ptr) \
     (ptr = (m_arg)buffer)

#define marg_get(type,ptr) \
     ((*(type *)ptr)++) /* increment ptr by sizeof type */


/* front end side stub */

int64
Pdxdummy(uint64 address, ...)
{
  Cap_t pbuff = {0,0};

  pbuff.address = (void *) ((int)&address + sizeof(uint64)); /* addr
                                                    of params on stack
                                                    */


  return (PdxCall(address,pbuff,-1));

}

#define DB_connect(db,from,to,type,length) \
        Pdxdummy((uint64) db_connect_stub,db,from,to,type,length)


/* database side stub*/

int64
db_connect_stub(Cap_t cap)
{
  DB db;
  Id from;
  Id to;
  String type;
  Int length;
  m_arg ptr = (m_arg) cap.address;

  /* remove arguments from buffer */

  db = marg_get(DB,ptr);
  from = marg_get(Id,ptr);
  to = marg_get(Id,ptr);
  strncpy(type,marg_get(char *,ptr),10);
  length=  marg_get(Int,ptr);

  /* call real function */

  return db_connect (db,from,to,type,length);
}
```

Figure 5.5: OO1 database example for db_connect

Chapter 6

Security in Mungi

The previous chapters have introduced the Mungi protection mechanisms. It is pertinent at this point to illustrate their application. Accordingly this section will briefly sketch out the way that the Mungi protection mechanisms can be used to implement a range of security operations. Discussion is split into two sections: one covering some well know security problems, while the other introduces the security framework that is currently implemented by Mungi.

6.1 Common Security Problems

6.1.1 Hydra security requirements

The designers of Hydra introduced a set of security problems (for more information refer back to Section 2.4.3). These problems provide a good set of initial goals for security.

Mutual suspicion

Transfer of rights is important in mutual suspicion; two procedures that call each other want to ensure that only information that needs to be shared will be.

As pointed out previously, capabilities adhere well to the principle of least privilege. They allow fine grain control of information passed to a called procedure. Conversely, the Mungi PDX mechanism allows a called procedure to be able to have private objects. Using PDX and specifying a suitable restricted protection domain, a mutually suspicious procedure call can be set up.

Modification

Modification is concerned with being able guarantee that a given object will not be modified by a procedure. The Mungi protection mechanisms allow this guarantee to be made. Placing a modify restriction on an object requires a *negative write* capability, this capability; if found in a protection domain, will deny any write access to the object. To allow this capability to

be useful, one has to be able to construct a protection domain in which the following two conditions hold: the negative capability must be found before any capabilities that grant rights, and the capability must not be able to be removed from the protection domain. The first condition can be met by placing the negative capability at the start of a Clist and inserting this Clist as the first entry (ie. the first slot) in the protection domain. To prevent modification of the protection domain, and to satisfy the second condition, the first slot of the APD is locked. Recall that a locked slot is not able to be modified. Any thread that is started in this protection domain is prevented from modifying the object.

Conservation/Revocation

Mungi capabilities can be revoked by the owner of an object by simply removing the password for that capability from the object table. Any attempts at validations of the capability, after this point, will be rejected by the system. This revocation is not immediate for two reasons: validations will still exist in validation caches, and pages of the object might still be mapped into the protection domains of other threads. To aid in revocation, validation caches are periodically flushed, requiring a re-validation of all accesses. It is at this point that the revocation actually takes place. The flushing of the validation cache might leave a protection domain in an inconsistent state, with some pages of the object mapped, but without being able to access further pages of the object. It is currently left up to the application to clean up in this case. Applications need to be aware that capabilities that are shared can be revoked at any point by the owners of the object.

One of the issues for future work is to make revocation immediate. This can be done by removing the mappings for the object in all protection domains when a capability is revoked.

Capability propagation

In segregated capability systems such as Hydra it is possible to control the passing of capabilities because the kernel controls all operations on capabilities. In a sparse capability system, because capabilities are just treated as data, the issue of capability propagation is equivalent to that of confinement, the prevention of information leaking to a third party.

Confinement

As discussed in Section 2.1.1, the ability to confine an untrusted procedure is sometimes advantageous. Confining a thread is possible in Mungi, and relies on the fact that a thread can only modify its protection domain if it has either write access to any of the Clists defining the protection domain, or it has the capability to modify its APD. A user wanting to confine a thread to a certain protection domain has to create an APD that contains only the capabilities that the confined procedure needs. The caller needs to make sure that this protection domain does not contain any writable objects readable by others. The user then starts a thread in the newly crafted APD. As the confined thread does not have access to the APD (as it is not passed the capability to operate on it), it cannot add any objects to

its protection domain. The only means of communication is through the use of objects that have been supplied by the caller of the untrusted code.

The method of implementing confinement leaves it up to the caller to ensure that there are no writable shared objects. Ideally the system should be able to offer this guarantee. The issue of confinement in Mungi is an ongoing one.

6.1.2 Other policies

Reference monitors

The most general way of implementing protection is to have a reference monitor that can check the validity of all access made in the system. By changing the reference monitor, the security policy of the system can be changed. Mungi also provides support for a type of reference monitor. Recall that Mungi allows a slot in an APD contain with a Clist or a Clist/protection handler pair. In the general case, this handler is inserted by an application that knows its access patterns and is thus able to supply valid capabilities more efficiently then the kernel. The validation handlers can also be PDX modules. In this case, when a thread faults and the protection handler is called, the handler can be running a protection domain different from the faulter's. The handler thus has the potential to access capabilities that the faulting thread cannot see. The PDX protection handler can then apply its own security policy and add a valid capability to the domain of the faulter, depending on whether the handler judges the access valid or not.

The reference monitor is able to model users. This is done by associating each user with a specific APD. Recall that APDs are uniquely identified by their base address. While the reference monitor is running, it can get the base address of the APD from the APDGet call and thus use it to identify the user.

Chinese Wall

Chinese Wall is a mandatory access policy, which can be implemented by a reference monitor. The access of one member of a conflict class will result in negative capabilities for the other members being added to the protection domain of the caller.

Data Integrity

Mungi is ideally suited to implement the policy of data integrity. In the framework that is presented in Section 2.2.3, a system can easily be set up using Mungi primitives. Transformation procedures must be easy to verify and therefore should be simple. We can use Mungi PDX to implement the transformation procedures (TP) in a straightforward way. Constrained data items (CDI) are objects whose capability is only contained in the appropriate TP. This ensures that only TPs can access the CDI (rule E1). With a proper login facility (like the one described in the next section) Mungi is able to uniquely identify all users (rule E3). The PDX transformation procedure maintains a read-only list of objects

which can be accessed by each user (rule E2), with only the authorised agent being given the write access to this list (rule E4).

As the implementation of PDX is fast, this is an ideal solution for the above problem.

Unix interface

The Mungi protection framework is flexible enough to implement access control lists in general. As an example of this, an implementation of the standard Unix protection mechanism will be discussed.

Recall that all objects in Unix have associated with them a reduced access control list. This list specifies access rights for three groups of users, namely the owner, members of the group and other users. In Mungi the same can be done for any object. All that is required is that three capabilities with different passwords exist for the object. The first capability specifies the rights that the “owner” of the object has, the second will specify what rights members of the group have and the third will specify the rights that all other users of the system have.

If all capabilities which represent owner rights for a particular user have the same password, then knowledge of that password would allow access to these objects with owner rights. Similarly for group and other rights.

A logon procedure would procure a user password and logon name. With this password, all group passwords and the “other” password are conveyed to the user. Now access is based on the passwords that the user has. All that has to be done is to implement a capability handler that returns the correct capability when an access occurs.

In the example in Figure 6.1 the user would have read and write access to the object, derived from its password.

6.1.3 Language based protection

Mungi provides protection that is based on an object level. Furthermore, PDX allows protection of methods within an encapsulated PDX module. One of the disadvantages of having a protection granularity which is multiples of page sizes is that there is no support for fine grain object protection. Like Opal [CLFL94], Mungi also takes the view that fine grained protection is a programming language issue. Object-oriented programming provides support for objects that are the size of integers. It is obvious that allocating an Mungi object (which has to be at least the size of a page) would be an inefficient method of dealing with the problem. The Mungi protection mechanisms in conjunction with a pointer-safe and object-safe language would provide a good compromise of performance and object-oriented principles. Languages like Cedar [SZBH86], Pilot (mesa) [RDH+80] and Modula-3 [Nel91] are candidates to be used in this fashion.

6.1.4 Other operating systems

The ability to confine information is also provided in other systems. Hydra allows a caller to guarantee that capabilities have their modify rights masked. In this way, a procedure

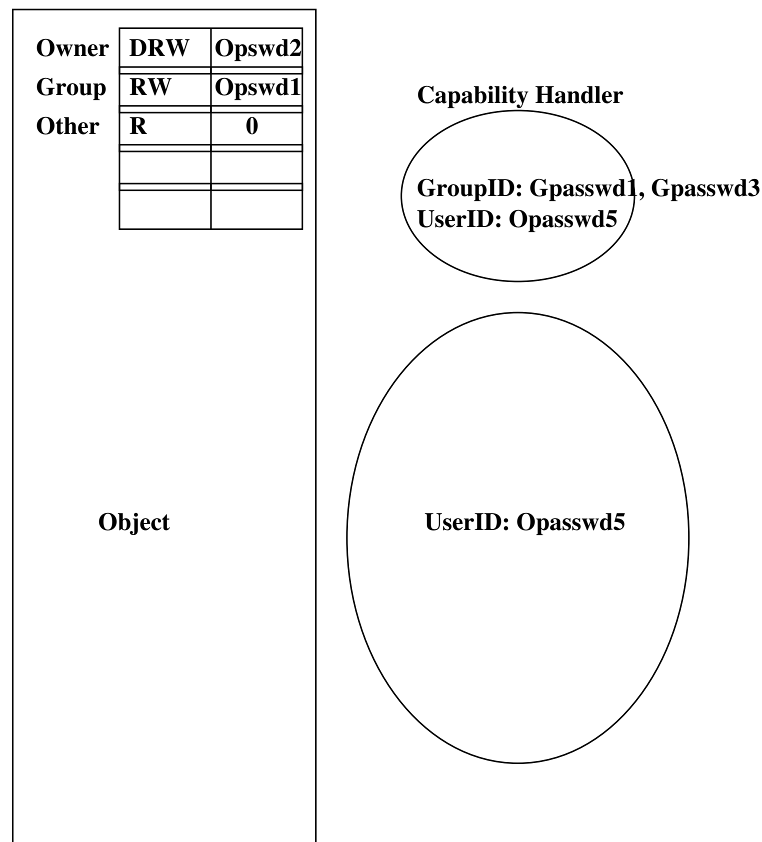

Figure 6.1: Unix-style access control

cannot write information, and thus cannot leak. The system described in [APW86] supplied a different slant on the problem of confinement in that they allowed the confined procedure to write normally, but prevented the flow of information by preventing any capabilities to be transmitted to a third party. Information that cannot be read by any other user is therefore not leaked. EROS takes a similar approach to Hydra, making sure that a confined procedure is not able to modify anything in the protection domain, therefore rendering it unable to leak information. The authors of EROS give a proof of correctness for their model [SW97]. L4 also allows confinement through the use of clans and chiefs. A task is confined by by restricting the only means of communication, namely IPC, to pass through the chief before reaching any other users [JLI98].

Reference monitors allow users to implement any security policy. The Angel single-address-space operating system implements a user level object manager which plays the role of reference monitor; all object accesses have to be authenticated by the server.

6.2 Mungi Framework

This section will describe the Mungi user interface in its current incarnation. Following this we will describe how the Mungi protection system can be used to implement other security schemes.

6.2.1 Login

Mungi currently implements a simple login based security policy, at the user level, without placing too many restrictions on the user. When the Mungi kernel has finished its initialisations, it starts the user login thread in a known protection domain. The login thread initialises its own protection domain further as required. After the initialisations are completed the thread will issue the familiar login/password challenge. Once a user has provided a login name, the login thread looks up the base address of the the user's APD in a system table[1] (cf. /etc/passwd). The user is then prompted for a password which is concatenated with the base address of the user's APD to construct a capability for the APD. The login thread then starts a new thread in the new APD, using the constructed capability. If the user supplies an incorrect password, the thread-create will fail and the login server will prompt the user for another login/password pair.

6.2.2 Group management

Single-address-space operating system create an ideal environment for sharing, due to the fact that all addresses are globally valid. In Mungi and other systems that use password capabilities, sharing is even easier as a capability, when shared, represents a pointer to an object as well as the right to use that object. Mungi allows Clists to be shared, this allows groups to be set up. Users that belong to a group are given a capability that allows read or read/write access to that Clist. By including that capability in the Clist, a user now becomes part of the group.

It is also possible to have append-only Clists in Mungi: A PDX module is created that includes a Clist. Users are given a read/PDX capability for that Clist. To add a capability to the Clist, users call the PDX procedure.

```
Clist_t clistv;

void
clist_addcap(Cap_t cap)
{

   Cap_t * ptr;

   ptr = clistv.caps[clistv.n_caps];
   *ptr = cap;
   ((Clist_t *) clistv)->n_caps++;
}
```

[1] Note that only the address of the user's APD is needed.

6.2.3 Name server

A single address space provides all objects with a unique, globally valid name: its base address. These 64-bit numbers are not a natural way for users to access objects; users are used to symbolic names for objects. To this end, the Mungi system provides name services that map symbolic names to object addresses.

Such name services can be implemented without having any special privileges, as the service would only store addresses of objects. Access to objects is still subject to the possession of a capability. Currently, the Mungi nameserver is a simple mapping from strings to addresses, but a more sophisticated naming scheme, like that used by Plan9 [PPTT90], is envisaged for Mungi. As the name service is at user-level, users can implement their own name space, while still being able to share objects by using their virtual address.

6.2.4 Capability refinement

.

Mungi, with its password capabilities, allows users to share capabilities without the intervention of the kernel. To conform with the principle of least privilege, a user might want to pass a capability with a lesser set of access rights on to another user. As only owners are able to create capabilities for an object, this would require a request to the owner. The owner of the object would have to request the kernel to create a new capability and then pass it back. This set of operations can be potentially expensive, and as such Mungi provides a library function that implements a scheme, proposed for Amoeba [MT86], that allows user to refine capabilities without calling the kernel or an owner of an object.

From the owner capability, C_{rwxd}, a new capability $C_{rwx} = f(C_{rwxd})$, where f is a well-known one-way function, can be derived, which only gives permission to read, write and execute the object. That capability can be further restricted to $C_x = f_x(C_{rwx})$, which allows only execution, and $C_{rw} = f_{rw}(C_{rwx})$, which allows only reading and writing. f_{rw} and f_x are related one-way functions, i.e. f with a constant string (s_x, s_{rw}) XOR-ed with its argument, so that $f_{rw}(s) = f(s_{rw} \oplus s)$, $f_x(s) = f(s_x \oplus s)$. The former capability can be further restricted to $C_r = f(C_{rw})$, which only allows reading. This capability hierarchy is shown in Figure 6.2. The owner of the object would need to generate all the passwords that conform to the protocol, but this would then allow users to perform their own refinement.

6.3 Summary

The Mungi protection mechanisms are sufficiently flexible to be able to address a cross section of security problems. Mungi is able to support confinement, conservation and revocation as well as being able to support such other security policies as the Unix security model.

The Mungi security framework provides a login based security system that caters specifically for the sharing of information. This is done by allowing the sharing of capabilities

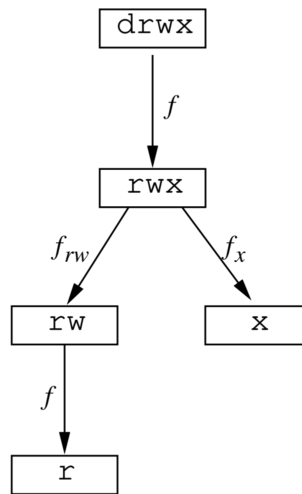

Figure 6.2: Hierarchy of derived capabilities

and the deriving of capabilities with lesser rights all without the expense of intervention by the kernel.

Chapter 7

Mungi Implementation

This thesis has so far presented the logical framework for the protection and security system in Mungi. It now remains to show that these abstraction work and can be implemented with an acceptable overhead. This chapter and the next will talk about the implementation and performance of Mungi and the protection system.

7.1 Implementation History

The initial specification of Mungi was completed in early 1993. The first attempt to implement Mungi was to modify CHOICES [CRJ87] to support a single address space. CHOICES is an object-oriented operating system, the strict object-oriented design was intended to provide the project with a rapid prototype for Mungi. This was, unfortunately not the case, the disjoint address space model was embedded too deeply into the CHOICES abstractions to allow a quick Mungi prototype to be generated. The conversion would have required changing the lowest level abstractions that CHOICES provided. In mid 1995 the CHOICES approach was abandoned and it was decided to prototype on top of the L4 [Lie92] microkernel. The L4 microkernel was chosen for a variety of reasons:

- Ease of portability. The L4 mircokernel is currently implemented in 64-bit mode on the MIPS R4x00 [EHL97] and the DEC Alpha [Sch96] (as well as the 32-bit ix86 version), this effectively results in Mungi being easily ported to both these architectures. As more 64-bit ports of L4 become available Mungi can easily by ported to these architectures.

- Minimal duplication of effort. L4 provides just the right level of support, this is evident from the close mapping between the operations provided by L4 and those supported by Mungi. (For a description of areas that do not map well see Section 7.6.) L4 has very little code that is not needed to support the implementation of Mungi and, more importantly, all the Mungi operations could be implemented by the mechanisms supplied by the L4 kernel. This almost ideal match leads to a small, fast system,

which would not be the case if Mungi had been built on top of some larger kernel such as Mach.

- Prototyping on a small fast kernel made sure that the performance of the operating system was not unduly degraded by the choice of prototyping platform. Recent work has shown that building on top of L4 leads to only a small performance penalty, which is offset by the increase in flexibility [HHL+97]. This is again in contrast to building on top of Mach [Cha95].

Unfortunately, L4 was still under development at the time, so its predecessor (L3) was chosen as an interim platform. This choice was motivated by L3's high degrees of similarity with L4, both in operation and functionality. During the implementation of Mungi, care was taken to only use system calls and services that would be eventually supported by L4. Mungi on PC-L3 was partially implemented when PC-L4 was released. The task of porting Mungi from L3 to L4 was relatively simple as it occurred at the early stages of Mungi implementation. While the implementation of Mungi continued on 32-bit PC-L4, a parallel effort was made to implement a 64-bit version of L4 on the MIPS R4x00 architecture. In August 1996 the first version of MIPS L4 was ready and the first port of Mungi ran on an 100 MHz SGI Indy in late 1996. Just before Christmas 1996 the first "hello world" appeared.

7.2 L4

The current version of the Mungi kernel runs as a user-level task on top of the MIPS R4x00 versions of the L4 μkernel. Naturally, as the implementation of Mungi depends heavily on the services that are supplied by L4, this section will describe the pertinent points of the L4 kernel.

The L4 μkernel and its predecessor L3 were both designed by Jochen Liedtke. The major design principle behind L4 was to construct a minimal kernel. A minimal kernel means "...a concept is tolerated inside the μ-kernel only if moving it outside the kernel would prevent the implementation of the system's required functionality."

Functionally, L4 presents three simple abstractions, *Address spaces, threads* and *inter-process communication* (IPC), which are described below.

7.2.1 L4 address spaces

Although L4 is not a single-address-space system, its flexible address-space operations can easily be used to construct a single address space. Address spaces in L4 are mappings from one virtual address space to another. Address space mappings in L4 can be arbitrarily recursed, that is any region in one address space can be mapped into another address space, creating a mapping hierarchy. The only restriction is that mappings must not be circular; all address mappings are eventually rooted in physical memory.

L4 provides four operations on address space regions as described below. The results of the operations are shown in Figure 7.1. Each bar in the figure represents an address space and the shaded boxes represent regions of the address space.

Grant: The owner of an address space can arbitrarily donate pages to another address space (It should be noted that for both the grant and map operations the receiver has to agree to receiving the page and is able to specify where in its address space the region should appear.) The page and all control of it are then removed from the address space of the granter.

Map: The map operation is similar to the grant operation, except for the fact that the page and control over it are not removed from the address space of the mapper. This allows the pages to be shared by more than one address space. The address space that performs that map is able revoke the mapping at any time by means of the two operations below.

Unmap: This call recursively removes a page from all address spaces that have received the mapping from the current address space. (Remember that mappings might be arbitrarily recursed.)

Flush: Similar to umap, except that the page is also removed from the address space of the caller.

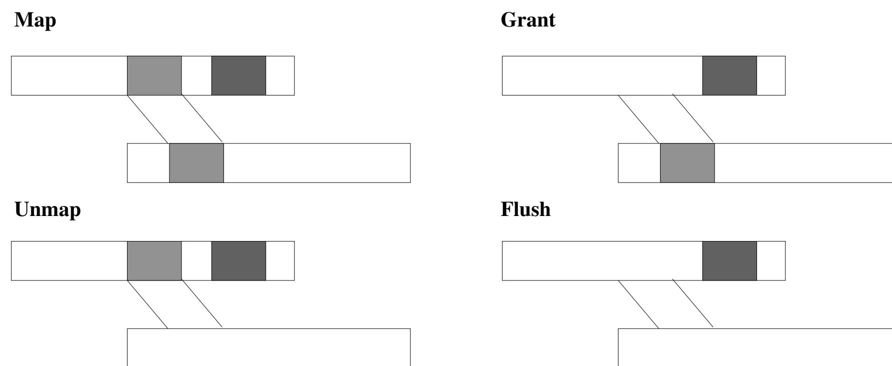

Figure 7.1: L4 address-space operations

L4 provides a default pager called σ_0 which initially contains all the mappings to physical memory. σ_0 will grant any of the pages (physical frames) in its address space to the first address space that requests it.

7.2.2 Tasks and threads

L4 provides support for threads and tasks. A task in L4 is an address space with one or one or more threads running in it. A thread is a lightweight execution context. When a task is created using the **task_new** system call, L4 will create a task with one running thread;

this thread is called lthread 0 (l_0). As well as creating l_0, L4 creates 127 inactive threads within the task. These threads can be made active ("creating a thread") by assigning a valid program counter and stack pointer to them with the **lthread_ex_reg** system call.

Creating a task in L4 implicitly grants the creator the right to operate on that task identifier. The operations that can be performed are kill and start. Having "ownership" of a task is important as task ids in L4 are limited to 2048. A thread can also create a task without running threads, this is essentially an operation that grants ownership to an (inactive) task and has the effect of reserving the right to activate that task. A mechanism also exists to release a task identifier once a thread has finished with it. The simple task and thread paradigm leads to a fast implementation of both thread and task creation; thread creation costs in the order of 1000 cycles (10 μs in L4 on a 100MHz R4700), while the creation of a task is also relatively lightweight, costing 7500–10000 cycles.

7.2.3 Interprocess communication

Communication in L4 is by means of synchronous message-passing IPC. The IPC mechanism has multiple functionality: L4 supports send, receive (from a specific thread) and wait (receive from any thread). The user can specify timeouts, allowing unsuccessful IPC to be aborted after this timeout[1]. L4 IPC [LES+97] is particularly fast if all the data being passed can fit into the available registers (on the Intel 80x86 two 32-bit words, or eight 64-bit words on the MIPS R4x00 and Alpha). This "short" IPC takes about 87 cycles on the R4700, although this figure is a little misleading as an application has to save its register set before making the IPC system call, adding at least another 30 cycles.

L4 IPC is the vehicle that allows the passing of memory from caller to receiver. A message content structure (called a message dope) regulates the type, content and amount of data that is passed during an IPC. There are three main types of data that can be passed with IPC message:

Words: Immediate data; the IPC will transfer (copy) a specified number of words from the address space of the sender to that of the receiver. Words are 64-bits long on 64-bit architectures and 32 bits on the 32-bit Intel 80x86 architectures.

Strings: Indirect data; strings allow the passing of arbitrary segments of memory by specifying a start address and a length.

Fpages: The mapping and granting operations (as described above) are implemented as a side effect of message passing. An *fpage descriptor* in the message dope specifies what page range is to be mapped or granted into the address space of the receiver. Fpages are segments of memory, of a power-of-two multiple of the page-size, aligned according to their size.[2]

[1] Sleep is implemented by receiving from an invalid thread with the appropriate time.

[2] An fpage of size 2^s has to have a 2^s aligned base address.

7.2.4 Page faults, interrupts and exceptions

External events in L4 are signalled to the user with by the L4 IPC mechanism. To facilitate this, a pager and an exceptor thread is registered for every thread.

The generation of a translation exception by the memory management hardware will invoke the L4 kernel. The kernel will then send an IPC to the pager that is registered for the thread that caused exception. This page fault IPC contains the faulting address and is crafted by the L4 kernel to appear to have originated from the faulting thread. At the same time, the L4 kernel sets up the faulting thread to wait for a mapping IPC from the pager. The pager can then supply a page, to resolve the page fault, by sending fpage IPC (see IPC above) to the faulting thread.

Exceptions that are generated by hardware are dealt with in the same way, that is when an exception occurs the L4 kernel will send an IPC on behalf of the excepting thread, to its exceptor. The exceptor is then able to handle the exception (usually by killing the excepting thread and cleaning up).

A special receive operation will associate a thread with an interrupt. Once this interrupt occurs, L4 will send IPC to the associated thread. Interrupt association happens on a first-come, first-served basis: The first thread that performs the special receive operation on an interrupt will have the interrupt associated with it. All subsequent requests to associate with that interrupt will fail, until the owner thread releases the association. This allows the flexible handling of interrupts by user level handlers.

7.2.5 Clans and chiefs

To allow users to control IPC and to allow the L4 kernel to clean up when a task is destroyed, L4 arranges all tasks into a hierarchy of *clans* and *chiefs*. When a thread creates a new task, the creating thread's task then becomes the *chief* of the new task. Threads of all tasks with the same chief, constituting a *clan*, are allowed to send IPC to each other or to their chief. When a task tries to send a message to a task that is outside the sender's clan, the IPC is redirected to l_0 of the chief of the clan. Any IPC from outside the clan directed to a thread inside the clan is also redirected to the chief's l_0. This effectively allows the chief to control all the IPC in and out of the clan (see Figure 7.2).

To allow efficient cleaning up of tasks and threads, when a task is destroyed, all the tasks with that task directly or indirectly as a chief, are also destroyed.

7.3 The Mungi Kernel Implementation

The following section will describe how the Mungi kernel system is implemented as a user task running on the L4 kernel.

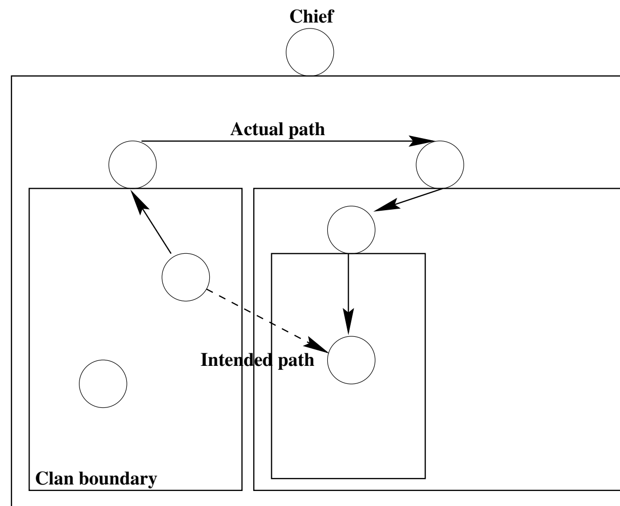

Figure 7.2: L4 clans and chiefs

7.3.1 System calls

Mungi system calls are converted (through the use of library stubs) into IPC to the Mungi kernel task. As a result, a null "system call" will consist of two IPCs: one to the kernel and one as a reply from the kernel. Consequently, the theoretical minimum overhead of a null Mungi system call is about 3 μs on the 100MHz R4700, instead, the measured overhead of a null system call is 4.3 μs[3] (see Section 8.2.1).

The logical structure of Mungi system is portrayed in Figure 7.3. The white blocks represent the physical system while the grey blocks represent the logical system; the Mungi kernel, the L4 kernel and the syscall stubs are all logically part of Mungi, although system call stubs are physically part of the application (in libraries).

7.3.2 Bootstrapping

One of the jobs-in-progress is the implementation of a PCI disk device driver. As a result, the current version of Mungi does not yet have a disk, which means that the entire environment has to be loaded into memory every time we reboot. Currently, an image that contains the L4 kernel, the Mungi kernel, a user login task and a Mungi shell is created and loaded into memory. The boot PROM starts the L4 kernel which then does all the initialisations that it needs to do and executes the Mungi kernel. The Mungi kernel, when first started, performs the following tasks:

- The initial kernel thread requests all the physical frames from σ_0 by touching all of physical memory.
- To prevent users from making calls directly to l4_task_create, the kernel will start all available (2048) tasks inactive. This essentially places all the available task identifiers

[3]The discrepancy here is probably due to cache misses.

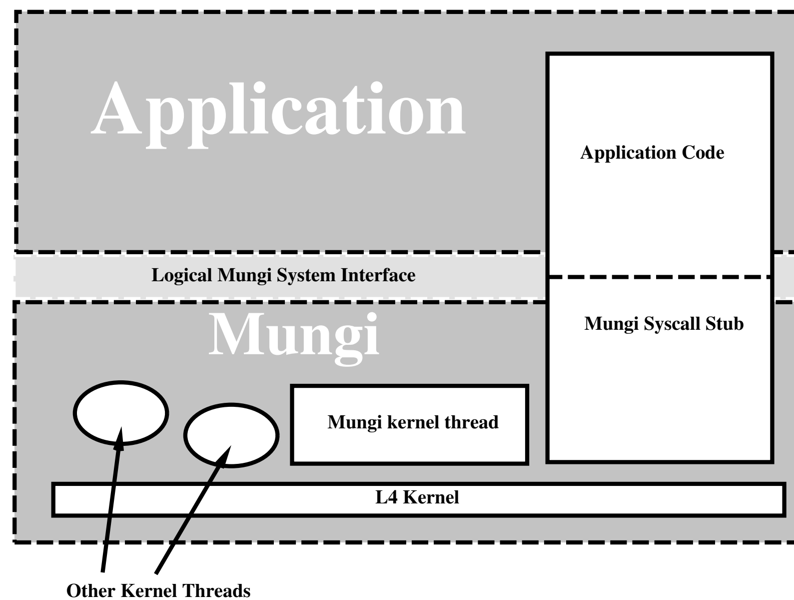

Figure 7.3: Mungi logical structure

and the ability to create and destroy tasks under the control of the Mungi kernel.

- All the other threads in the Mungi task are then started (see below for details of other threads in the Mungi kernel).
- Finally the initial kernel thread starts the first user task.

The Mungi kernel task consists of a number of threads which are listed below.

Mungi system call server: This is the thread that deals with most of the system calls.

Mungi default pager: The Mungi default pager deals with page faults from user threads. The pager checks the validity of the access (in accordance with the current protection domain) and, if the access is valid, maps the page into the address space of the faulting thread.

Serial interrupt handler: Currently the only interrupt that the Mungi kernel handles is the interrupts from the serial chip (see Section 7.4.2). To allow for more devices, will require one separate thread for each interrupt that the Mungi kernel wants to handle.

Semaphore handler: Handles system calls that relate to semaphores. This handler puts threads into a waiting queue when a semaphore is zero or negative. It also deals with the wakeup when there are threads waiting on a semaphore which has been signalled.

Timer: Provides 100ms ticks for user/kernel use.

7.3.3 Thread and task creation

Many of the Mungi system calls map directly onto L4 system calls. Thread operations fall into this category. A protection domain combining a running thread is implemented as an L4 task. Mungi threads are created using the **ThreadCreate** system call. Logically, there are two different cases of thread creation: A thread is created in the caller's protection domain, or the thread is created in another protection domain. L4 restricts the authority to manipulate(create) threads in a task to the other threads in that task. As a result, Mungi has to rely on threads that are already in a task to create other threads. For this reason, the lthread 0 in every task performs operations on the kernel's behalf.

Lthread 0 runs on the kernel's behalf[4] within the L4 task as shown in Figure 7.4. The lthread 0 in a user task is used by the kernel to handle exceptions (see Section 7.4.3). All task creation is done by the Mungi kernel server, giving the Mungi kernel the ability to kill any of the user tasks in the system. Remember that the Mungi kernel server has already preallocated all the L4 task identifiers (TID) so that even if a user thread tries to call **task_new** directly, there will be no available TIDs for a new task and therefore the operation will fail.

The creation of a thread in the same protection domain as the caller is implemented as shown on the left in Figure 7.5. When a thread makes a call to **ThreadCreate**, the system call library stub will send a message to the Mungi kernel. The Mungi kernel will allocate a new thread identifier (TID) and pass this back to the stub in the reply message. Once the stub has the new TID, it invokes the L4 **l4_ex_reg** system call which creates the new thread.

If the caller of **ThreadCreate** wants to start a thread in a different protection domain, another mechanism has to be employed. Starting a thread in a new protection domain is implemented by starting a thread in a new L4 task. The steps that are taken to start this new thread are also shown in Figure 7.5 and are as follows:

1. The **ThreadCreate** system call stub sends a message to the Mungi kernel.
2. The Mungi kernel will authenticate the use of the target protection domain. If the request if valid a new task and thread ID are allocated, and a new task is started using the L4 **task_new** system call. The creation of a task in L4 will implicitly start lthread 0 in this new task.
3. The kernel sends a message to the new task's lthread 0 containing the thread ID for the new thread.
4. Lthread 0 in the new task will call **l4_ex_reg** to start the user thread in the new task.
5. The Mungi kernel replies to the caller of the **ThreadCreate** system call, returning the new thread ID.

[4]Although the Mungi kernel relies on the fact that there is an lthread 0 running in every Mungi task, lthread itself has no special privileges. If one of the other threads in the task decides to destroy lthread 0 this would affect only the ability of the task to deal with exceptions, without affecting the kernel or the other tasks at all.

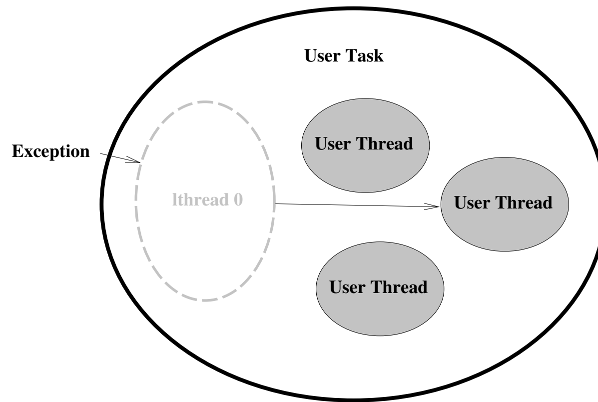

Figure 7.4: Mungi user threads in same APD (showing lthread 0)

Although logically there are only two cases in thread creation, implementation allows the creation a third class; when a thread attempts to create a thread in a protection domain that already has active threads running in it. In this case, step 2 above is omitted and the kernel IPCs the request to start the new thread to lthread 0 of the active L4 task associated with the protection domain.

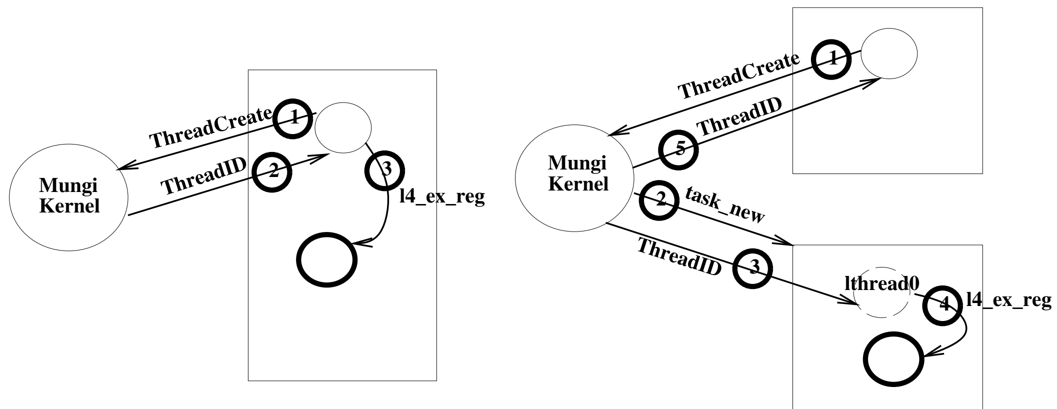

Figure 7.5: Thread creation: In local protection domain (left) and in different protection domain (right).

L4 provides no explicit way to kill a thread, short of killing a whole task and all the threads in it. To simulate killing a thread, it is forced to execute[5] a **l4_sleep** system call, with an infinite timeout. The Mungi kernel libraries provides a stub that executes the blocking wait. Thread deletion in Mungi is thus implemented in much the same way that thread creation is — by restarting the thread (at the blocking wait system call). Thread deletion is simpler when a thread wishes to delete itself. This is handled simply by the

[5]Any thread (even one that is currently executing) can be forced to start execution anywhere by specifying the new instruction pointer to **l4_ex_reg**.

kernel not replying to the **ThreadDelete** message. This will leave the thread waiting for ever on a reply from the kernel (until the kernel chooses to re-use the thread id to start a new user thread).

7.4 PDX

Setting up a new protection domain is not a lightweight operation; this is the dominant cost in a PDX call. However, the setup cost can be amortised by caching the PDX procedure's protection domain, in particular its access validations, between calls. When a thread t calls a PDX procedure p for the first time, a new L4 task t_p is created. The PDX call essentially becomes a blocking RPC call, which spawns a new thread in t_p for the duration of the PDX execution. The operation of creating a new thread and transferring control to it is very lightweight in Mungi, as it maps directly onto the corresponding L4 operations.

In the case of a true protection domain extension (i.e. `npd` = -1), t_p can inherit t's cached validations by having t_p reference t's validation cache. If, during p's execution, new objects are validated, these validations are prepended to t_p's validation cache, without affecting t's part of the validation cache. If t validates further accesses between calls to p, these are inserted into t's part of the validation cache. On the next call to p, t_p will then inherit these further validations as well.[6] This is shown in Fig 7.6. The kernel's data structure describing a protection domain contains references to all PDX tasks belonging to it, so these can be cleaned up when the last thread exits.

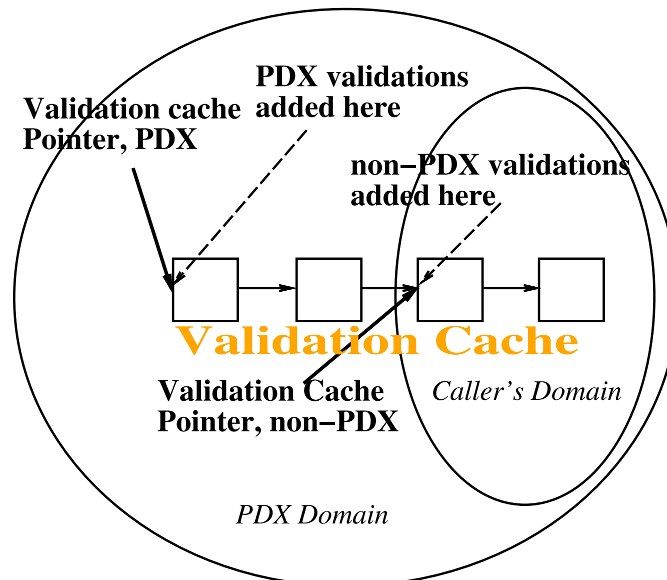

Figure 7.6: Mungi validation caches

In the case of PDX calls where the user specifies a protection domain (`npd` > 0), caching

[6]The same happens if, while t executes p, another thread in t's protection domain adds further validations — these become immediately visible to t_p.

can also be used, provided the Clist has not changed since the last call. The kernel can verify this by storing a hash of the Clist with the cached protection domain.

7.4.1 Page faults

As previously mentioned, L4 page faults are translated into IPC to the thread that is registered as the pager for the faulting thread. In Mungi, the kernel pager is set as the default pager for all threads that are created. When a thread within a task touches a non-resident page, the L4 kernel will send an IPC to the Mungi pager and set up a wait for the pager on behalf of the faulting thread. The pager will then do the necessary access validations (see Section 4.3.4), and if the request is valid, send the requested page back to the faulting thread by means of the closed IPC that has been set up. In the case where the pager does not respond, the faulting thread will remain blocked on the closed wait. If the faulting address falls within an object that is managed by a user-level pager, this pager is invoked.

7.4.2 Interrupts

Like page faults, interrupts are also converted into IPC by the L4 kernel. The current version of Mungi only has to deal with one device interrupt, the interrupts that come from the serial chip. Before the Mungi kernel starts the first user task it will associate a thread with the serial chip interrupt that L4 will act on. When an interrupt occurs, the interrupt handling thread will call an appropriate interrupt handler, the address of which is stored in an interrupt vector. The serial chip interrupt handler is responsible for either receiving a character from the serial port or the sending of the next four characters (see Section 5.4.1).

7.4.3 Exceptions

L4 exceptions that are raised by a thread are sent by IPC to the default exceptor, which is a Mungi kernel thread. These exceptions are generated by the hardware (bus error, reserved instruction etc.). Mungi generated exceptions, such as protection and segmentation violations, are generated internally in Mungi. When an exception from either set occurs, the kernel will send a message to lthread 0 of the task of the excepting thread, passing the address of the exception handler for that task. Exception handlers can be set through the **ExcptReg** system call. Lthread 0 will then re-start the excepted thread (using **l4_ex_reg**) starting execution at the exception handler. Due to the fact that the handler is started in the context of the excepting thread, the exception handler can save the context of the thread before handling the exception. The exception number and the address where the exception occurred will be passed to the exception handler by lthread 0. A prototype of an exception handler is given below.

```
/* the system call to register an exception handler */
ExcptHndlr_t ExcptReg(Excpt_t Exception, ExcptHndlr_t Handler);

/* a prototype for an exception handler */
void handler(int64 Exception, int64 ip);
```

7.4.4 Capability handlers

Capability handlers are implemented much like exception handlers (see Section 7.4.3). During validation, if the default pager finds a non-empty handler pointer in the APD of the faulting task, if sends a message to l_0 of the faulting task. l_0 will then restart the faulting thread in the capability handler. Once the handler has performed its job, it can restart the faulting thread. The default pager remembers that the thread has just called a handler and will thus skip calling the handler and resume the search with the Clist associated with the handler.

7.5 Supporting Data Structures

7.5.1 Kernel information page

Mungi provides a kernel information page at a known address. This page contains information that is used by lthread 0 and the system call stubs.

```
typedef struct {
   l4_threadid_t kernel;           /* thread id of the kernel */
   l4_threadid_t pager;            /* tid of kernel pager *
   l4_threadid_t preempter;        /* pre-empter */
   l4_threadid_t semhandler;       /* tid of semaphore handler */
   l4_threadid_t timer;            /* tid of the kernel timer */
   l4_threadid_t printer;
   l4_threadid_t kexhandler;       /* tid of the kernel exception handler*/
   int64 L4_KIP;                   /* the address of the L4 info page */
   char KType[6];  /* "Mungi" */
   int version_high;
   int version_low;
   int64 ProcTable;                /* address of the process table
                                          for 'ps' like operations */
} KernelInfoPage;
```

7.5.2 Object table

Figure 7.7 shows the data structure that is used by Mungi to keep track of objects.[7] The first part of the structure is labelled the public part and can be accessed by anyone holding a read capability for the respective object through an **ObjInfo** system call. Only the first half of the ObjInfo_t is referred to the caller, unless it holds an owner capability. The flags in the object table entry are used by the kernel to indicate that the object is a valid bank account or APD.

[7] The base address and length of the object are stored in the structure of the object table and can also be accessed.

```
typedef struct {
  /* public */
  uint       extent;
  Date_t     creation;
  Date_t     modification;
  Date_t     access;
  Date_t     accounting;
  uint64     userinfo;
  uint64     acctinfo;
  uint64     length;
  /* private */
  ObjFlags_t flags;
  uint       n_caps;
  uint       n_pdx;
  CapList_t  CapList[O_MAX_CAPS];
  PdxList_t  pdxList[O_MAX_PDX];
  Cap_t      clist;  /* for PDX */
  Cap_t      account;
  Cap_t      pager;
} ObjInfo_t;
```

Figure 7.7: An object table entry

7.6 L4 Limitations

This section summarises the problems and shortcomings of the approach to build the Mungi kernel as a user task on the L4 mircokernel. In general the L4 mircokernel is well suited to support the lowest level abstractions of Mungi. There are, however, a few areas where L4 did not quite fit the bill perfectly, specifically these areas are:

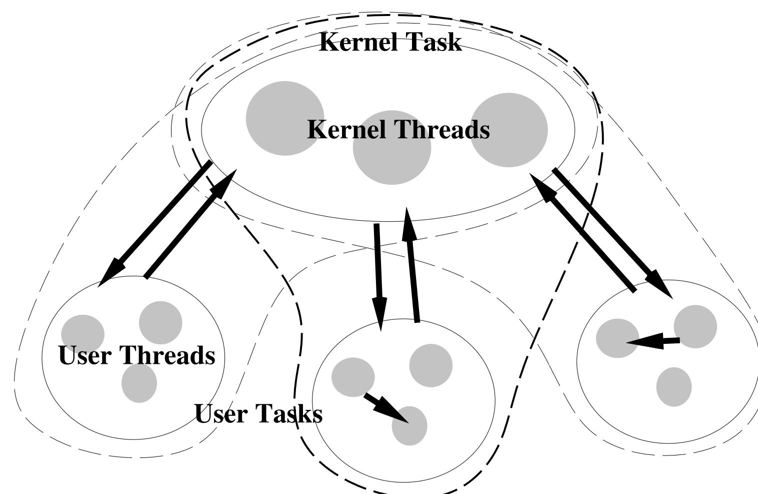

Figure 7.8: Mungi kernel with user tasks. Communication is allowed between threads in the same task and between any threads and the kernel

- Task and thread information. Mungi and L4 both have to keep information about threads. L4 needs this information for scheduling and Mungi needs the information in order to be able to allocate thread ids and maintain a thread hierarchy. The Mungi server does not have access to the L4 thread control blocks (TCBs), and as a result, some of this information is duplicated.

- Mungi system calls are implemented as interprocess communication (IPC) to the Mungi kernel. The pure IPC overhead adds to the execution time of a Mungi system call. Although an L4 IPC has an overhead of only 79 instructions [LES+97], the cost of

entering the kernel is 32 instructions, making the pure IPC overhead 47 instructions.[8] This cost would not need to be incurred on a native kernel.

- The L4's clans-and-chiefs structure allows tasks that are in the same clan to send IPC to each other without having to go through the chief. This had to modified for use by Mungi to prevent information leakage (i.e. enable confinement). The modification disallows IPC between threads of different L4 tasks except via the chief, to ensure that Mungi has control over all cross-protection-domain communication. See Figure 7.8 and Section 2.1.1. This result could also have been achieved without modifying L4, by enclosing each task in a second task, but at the cost of needing to start twice as many tasks and doubling the IPC overheads.
- L4 provides recursive mapping of address spaces. Mungi only uses one level of mappings, those from the Mungi kernel to the user threads. L4 support for recursive mappings is a disadvantage when unmapping pages or entire address spaces (APDFlush, TaskDelete) as L4 has to check each page for further mappings, which also have to be unmapped. Modifying L4 to allow only one level of mapping would make any unmapping operations a lot quicker, as whole ranges can be unmapped without having to check each individual page for further mappings.

7.7 Summary

Mungi is implemented as a user level task running on top of the L4 μkernel. L4 closely matches the needs of Mungi for lower level operations. This, together with portability and speed, are the main reasons for taking the μkernel approach. Using the flexible address-space operations that L4 provides, a single address space is constructed. Mungi system calls are implemented as stubs that send IPC to the Mungi kernel. The result is a first prototype of Mungi that is able to demonstrate the benefits of the Mungi single address space and primitives, without a significant loss of performance.

Although L4 seemed the perfect microkernel for the job, the choice of microkernel is not really important to the design of Mungi. The implementation on Mungi on L4 was only done as a shortcut to a full native implementation. Several other choices of microkernel would be possbile, with systems like EROS, with it support for capabilities built into the kernel, being a likely candidate. Building Mungi on another microkernel would be interesting from a technical perspective, but would add nothing new to Mungi itself.

[8]These figures are based on the MIPS R4600 version of L4.

Chapter 8

Mungi Performance

In this section, performance data for Mungi is presented and contrasted with two Unix operating systems: Irix 6.2, a commercial Unix operating system, and Linux version 2.1.67.[1]

It has recently been shown [HHL+97] that Linux can be run as a server on L4 with essentially unchanged performance; whether Linux runs native or as a server on L4 makes little difference in performance. It would seem a logical conclusion to claim that the figures presented for Mungi would be similar if Mungi were running native. It is, however, difficult to make direct comparisons between systems, as Mungi's superior performance is a result of the simple abstraction and the inherent advantages of the SASOS model, as well as a very performance-conscious implementation.

We also compare Mungi with Opal where possible. This is complicated by the fact that the systems are built on different kernels and different hardware platforms, and by the lack of availability of common benchmarks for comparison. However, we show that the Mungi model provides significantly higher performance for important operations such as cross-domain calls, where Mungi's PDX eliminates the need for capability validation on each call.

All the Mungi, Irix and Linux figures were obtained on an 100MHz R4600-based SGI Indy workstation with 64Mb of RAM (see Appendix A). Comparisons with Opal are based on published data [CLFL94]. Opal timings had been obtained on a DEC 3000/400 AXP (133.3 MHz Alpha CPU). According to its SPEC ratings, this machine should be roughly as fast as the Indy (within 10–20 %).[2]

8.1 Microbenchmarks

Here we present timings obtained for basic Mungi system calls. These were obtained for repeated calls (presumably hot caches), although some of the figures varied strongly between calls, obviously resulting from cache conflicts.

[1] Linux/SGI is available from http://www.linux.sgi.com.

[2] Unfortunately, no exact figure can be given, as we only have SPEC-92 ratings for the Alpha used for Opal, and SPEC-95 ratings for our Indy.

The Indy's high cache miss penalty was evident in the fact that some figures showed an extremely strong dependence on the exact location of user code and stacks.

Where possible, we are comparing our timings with those obtained for comparable operations on Irix and Linux, and for those reported for Opal. The following sections explain the figures, which are summarised in Table 8.1.

Operation	*Mungi*	*Linux*	*Irix*	*Opal*
Null system call	4.6	6.3	7.7	>88
Thread create	83/48	N/A	N/A	N/A
Thread delete	48	N/A	N/A	N/A
Thread create + delete	131/96	2,450	4,882	N/A
Thread create (new domain)	600	N/A	5,600	650
Object create	60	N/A	N/A	315
Object delete	150	N/A	N/A	900

Table 8.1: Microbenchmark timings (in μs). See text for explanations.

8.1.1 Null system call

The cost of a null system call is 4.6μs in Mungi, its closest approximation in Unix is the `getpid` call which costs 6.3μs in Linux and 7.7μs in Irix. A lower limit for the cost in Opal is that of a Mach null-RPC, 88μs. In spite of requiring two IPC operations, the Mungi version of this call is significantly faster than the corresponding call in the other systems. This is an indication of the low overhead introduced by L4.

8.1.2 Tasks, threads and IPC

Creating a new thread, in the current protection domain, in Mungi takes 83 μs, which reduces to 48 μs if an ID can be recycled from a thread which has already terminated. In a context where threads are created and deleted frequently (and where consequently this cost is most important) this should mostly be the case. Thread deletion is the same cost as thread creation with recycling, i.e., 48 μs. Thread times for Opal were published in [FCL93] for an R3000-based DECstation (create 140μs, delete 230μs). However, as no clock speed or SPEC ratings were quoted for that platform, it is hard to compare these figures. Irix and Linux do not presently have a thread interface significantly more lightweight than `fork()`, so we used fork()/wait()/exit() as an approximation.

Thread creation, in another protection domain, costs around 600 μs in Mungi (800 μs with cold caches), the corresponding fork()/exec() in Irix around 5,600 μs. In Linux we could not measure a similar operation separately for lack of a timer of sufficient accuracy. We measured the cost of fork()/exec()/wait()/exit() as the total of task creation and deletion. The surprisingly high task creation cost in Linux (given its generally good performance compared to Irix) might be a result of this version still being under development. In any case, the operations are about an order of magnitude faster in Mungi than in the Unix systems.

The equivalent of creating a thread in a new protection domain in Opal, is creation and activation of a protection domain, which takes 650 μs.

8.1.3 Objects

Object creation (which, by itself, does not allocate any backing store) costs 60 μs in Mungi. Segment creation in Opal using a recycled inode costs 315 μs.

Object deletion in Mungi takes 150 μs, compared to 900 μs in Opal. Only the combination of creation, access and deletion could easily be measured in the Unix systems. The results were about 50 % slower in Linux and eight times slower in Irix.

8.2 Protection Performance

An important attribute of a protection mechanism is the overhead that is introduced in the system due to protection. Security relies on the protection mechanisms performing with as little overhead as possible: As was pointed out in Chapter 2, security has to rely on the users being security conscious. If the penalty for security is too high in term of performance then users will not use the secure settings.

The most obvious way to ensure that the protection penalty is not too high is to pay attention to operations that are performed often. If the system is optimised for such operations, then the operations that are performed less often can afford to take a little more time.

As the protection system in Mungi is supposed to by used as a base to implement a whole variety of security policies, we have to make sure that the base operations in Mungi are as fast as possible.

8.2.1 Page faults and validation

Due to the invasive nature of explicit capability presentation, Mungi provides support for implicit validation. This means that when a thread accesses an object, Mungi will attempt to find a valid capability for this access. The steps that are taken by the protection system are outlined below.

```
Validate(address, accesstype)
{

if (vcache(address, accesstype))
   map page
   return

for (caps in APD)
   if (ishandler(cap))
      call handler
   else if (isvalid(cap,object) && grant(cap,accesstype))
      vcacheadd(object, accesstype)
      map page
      return

raise protection exception
return
}
```

When a thread tries to access a page that is non-resident, a TLB exception will cause the L4 kernel to send IPC to the Mungi kernel pager. The pager will then attempt to validate the access. This is done by calling the *Validate* function. Validate checks the validation cache to see if the object has been validated previously, if so, validate returns and the pager maps the page. If the address is not in the validation cache then the pager has to check the protection domain of the caller. In order to do this, the object table is searched to find all the valid capabilities for that object. The APD of the faulting thread is searched in an attempt to find a valid capability. If an APD contains a pointer to a handler, this is called at the appropriate time.

The validation process is performed whenever an access is attempted on a non-resident page. As a result, good performance of validation is essential. The results of benchmarking the page fault/validation process are summarised in Table 8.2.

These figures need some explanation. The four cases benchmarked were touching a page of an object; that was in the validation cache, touching an object for the first time, touching an object for the first time with a larger APD, and touching an object for the first time and invoking a capability handler. The timings that are presented in a bold type-font are actually measured figures. In order to be able to make comparisons between various modes of validation, operations common to all were taken from the "in cache" case (such timings are presented in normal font). The true times were higher due to cache misses. (Altough for the case of the "In-cache" validation the calculated figure is higher then the measured value. This is probably due to the additional space in the cache the the benchmarking code takes up, see Section A.4).The numbers presented give a more accurate reflection of the number of instructions executed, instead of including the somewhat random cache affects. In the capability handler case, after the handler is invoked it returns to the instruction that caused the first page fault. This then results in a further "first touch" page fault.

	In cache	First touch	FT (+128 caps)	cap handler
Page fault to pager	**11.5**	11.5	11.5	11.5
Search validation cache	**4.5**	4.5	4.5	4.5
Call handler	–	–	–	**104**
Search APD	–	**6.5**	**39.5**	
Add to vcache	–	**17.5**	17.5	
Map page	**12.5**	12.5	12.5	
Total	28.5 (**26.5**)	42.5 (**75**)	75.5 (**113**)	162.5 (**200**)

Table 8.2: Validation overhead (in μs)

As a comparison, we note that Unix systems require files to be opened before first accessing them and closed after the last access. In Linux, opening an empty file takes $45\,\mu$, and in Irix, $252\,\mu$. Opal similarly uses explicit attach and detach operations on segments. An attach followed by a detach takes $478\,\mu$s "best case". We assume that the cost of an attach is half this time (which is most likely erring in Opal's favour). Mungi does not feature explicit attach/detach system calls. Objects are made available to a task by inserting their

capabilities into a user-maintained Clist (an operation that occurs less frequently than subsequent accesses to the objects). The Mungi operation equivalent to an Unix open or an Opal attach is touching an object for the first time, which was measured at 75 μs. This was the only operation we found to be faster in a Unix system than in Mungi: Opening a file in Linux is almost twice as fast as validating a first access in Mungi, obviously a result of the simpler Unix protection model. Irix, however, is much slower.

Mapping a further page of a previously validated object takes only 29 μs in Mungi (no comparable data are available for the other systems).

8.2.2 PDX

PDX is the foundation of flexibility and extensibility in Mungi. PDX is used to implement many services, from device drivers to user-level pagers. It is therefore essential that PDX can be executed quickly. Recall that the implementation of PDX requires the expense of setting up a new L4 task initially, but once established PDX can be performed at the cost of four IPCs. It therefore makes sense to measure both the initial and repeated timings for PDX (these figures are summarised in Table 8.3).

PDX allows the controlled creation of new (temporary) protection domain. The thread that initiates the PDX is able to specify what parts of the caller's protection domain are mirrored in the PDX protection domain. PdxCall performance, depends on whether the kernel has to validate the new domain or not. When the caller specifies a proper extension, or specifies a null protection domain, the kernel only has to validate the Clist pointer that the PDX module itself provides. On the other hand, if the caller presents a set of Clist capabilities that represent a new protection domain, the kernel has to validate each capability before it is entered into the APD (column "new APD").

The cross-domain call mechanism in Mungi is PDX, which costs between 10 and 20 μs. The equivalent operation in other systems is an RPC, which costs around 450 μs in Irix, 160 μs in Linux and 133 μs in Opal.

	full APD	new APD (9 clists)
PDX (first call)	874	1104
PDX (repeated call)	10–20	22

Opal	Irix	Linux
133	450	160

Table 8.3: Cross domain call performance (in μs)

8.2.3 Object benchmarks

In order to establish that PDX provides a good abstraction and incurs a reasonable performance penalty, a subset of the OO1 benchmark [CS92] was implemented and run on various

operating systems. OO1 is supposed to represent a benchmark that performs "typical" operations on an object-oriented database. The database that is accessed in the benchmark should be inaccessible to the user that is performing the operation. This is an ideal situation for Mungi's PDX protected procedure calls. To achieve a similar result on Unix we had to implement a client server architecture that relied on IPC. The experiment had two stages. First the code was written so that the database resided in the same protection domain as the client of the database. This results in the same code running on each of the different operating systems. The second part of the experiment had the database in a separate protection domain.

System	*lookup*	*traversal*		*insert*	*total*
		forward	*reverse*		
Linux 32-bit	7.99	5.97	N/A	N/A	N/A
Irix 32-bit	7.44	4.77	5.13	4.76	22.10
Irix 64-bit	7.78	6.47	7.49	6.23	27.97
Mungi 64-bit	7.95	6.60	7.71	5.31	27.57

Table 8.4: OO1 benchmark times (in ms) for the single protection domain version.

System	*lookup*	*traversal*		*insert*	*total*
		forward	*reverse*		
Irix 32-bit/message passing	946	1,445	1445	208	4,047
Irix 32-bit/shared memory	949	1,409	1,411	203	3,972
Linux 32-bit/message passing	344	467	461	842	2,114
Mungi 64-bit/PDX	51	59	61	16	189
Mungi 64-bit/PDX/restricted	50	58	60	16	184

Table 8.5: OO1 benchmark times (in ms) for the multiple protection domain version.

The OO1 database consisted of 20,000 parts. *Insert* creates 100 new random elements in this database and connects each new node to three random node already in the database. This results in 400 database operations. The *forward* and *backward traverse* operations started at a random node and followed all paths connected to a depth of 7. The forward lookup performed exactly 3,280 database operations. With the backward lookup the number of operation depended on the starting node. It should be noted that all the versions of the benchmark ran with the same random numbers and thus traversed the same node in each trial.

Table 8.5 shows the results of the performance measurements of the IPC version of OO1. Mungi outperforms Irix in average by more then a factor of 20 and Linux by more than a factor of 10. Comparing the values from Tables 8.4 and 8.5 for 32-bit Irix code, it can be concluded that the cost of an RPC in Irix is around 450μs, in Linux about 160μs, while the same comparison for Mungi yields 21μs for a PDX call, which is consistent with the figures given in Table 8.3.

The observation that Irix shared memory IPC does not perform better than System-V

message passing is explained by the fact that the amount of actual data passed is very small (around two dozen bytes), so that the cost is dominated by the system call and context switching overhead.

The last line in Table 8.5 (marked "restricted") was obtained from running Mungi on top of a modified L4 kernel, implementing IPC restrictions. In this version it is impossible to send IPC directly between Mungi user tasks, this version therefore fully supports confinement. As can be seen this is achieved without any run-time penalty, but at the cost of a small modification to the kernel.

8.3 Summary

The benchmarks show that Mungi clearly outperforms Unix operating systems on some of the most important basic operations, as well as on an IPC-intensive benchmark of database operations. This shows that the single-address-space approach is not intrinsically less efficient than traditional operating systems, and has a significant edge for certain classes of applications.

The microbenchmarks also clearly outperform Opal's published results. Obviously, Opal's performance was partly a result of the platform chosen for the implementation of the prototype. However, we have clearly demonstrated that the PDX mechanism can be implemented with very high performance, and is an inherent advantage of our model, compared to the approach taken by Opal.

The most significant performance advantage of a SASOS, however, will come in areas where the single address space can be used to avoid cross-domain calls or other operating system intervention altogether. This can, naturally, not be demonstrated on small benchmarks, including OO1. The DiSy group is therefore working on a port of a full-blown object-oriented database system to Mungi, where the potential of taking advantage of the model is particularly great.

Chapter 9

Conclusion

One of the traditional impediments to sharing is the fact that all processes execute in seperate address spaces. Getting data from one address space to another requires the use of additional operating system services such as IPC or the file system. Single-address-space operating systems, on the other hand, provide the ideal environment in which to share data. The fact that there are system-wide names for all data in the system allows applications to pass globally valid references, to any data, to other applications.

It is important that the clean sharing benefits of a SASOS are not negated by an intrusive or ill-matched protection mechanism; it is no use tearing down one set of walls only to put up new ones. Providing Mungi with a protection mechanism that is well matched to the concept of a SASOS and that is fast and powerful is the purpose of this thesis. This disertation has described a protection mechanism based on password capabilities that complements the single-address-space paradigm. Below are the main achievements.

- Mungi demonstrates that software implementations of capabilities and protected procedure calls can be made efficient. Careful design and extensive caching of address mappings allow, for example, repeated PDX calls to be performed in 10-20 μs on a 100MHz R4600 CPU. Similarly good performance is shown for capability validation, thread creation, and object creation.
- The Mungi operating system was implemented on top of the L4 μkernel. L4 proved to be an excellent choice, as it provided just the right level of support: its interface made the development project simpler, the abstractions L4 provided were well matched to the task of building an new class of operating system, and the emphasis on performance in L4 motivated similar concerns for Mungi.
- This thesis has presented a password-capability-based protection system. It has shown the flexibility of a capability based approach. Most modern operating systems implement protection by using access control lists. One of the problems that frequently arises in these systems is that a process that is running under a certain user id is subverted to do something else. With an ACL based system this process has all the rights and privileges that the user has. The amount of damage that can be done by

the errant process in not contained. With a capability-based system every process can be given exactly the rights that it needs to be able to perform its job. Thus if this process is subverted, then the only damage that it can do is limited to those rights that have been given to the process. This thesis has shown that capability based systems have no inherent performance penalty as compared to ACL based ones.

Having a small set of abstractions also supports more intuitive protection. A user sees only one level of protection, that is, protection of an object by password capabilities. This is unlike traditional systems, where there is protection at the filesystem level as well as the memory level. A capability-based single-address-space operating system makes it easier to think about protection implications for a user's application.

- Mungi provides a protected procedure call mechanism, called PDX, that temporarily extends the protection domain of the caller, allowing the caller to perform privileged operations in a controlled way. Mungi's PDX mechanism has been demonstrated to be particularly flexible and suitable for a wide range of applications, including device drivers and display managers. Mungi supports protection at an object granularity, where an object is a contiguous set of pages, allowing users to share information selectively. Mungi also provides mechanisms to ensure that information shared can not be leaked to a third party.

In summary, this thesis has presented the design, implementation and performance of the Mungi protection system. With its simple abstractions and careful attention to performance design, protection in Mungi is shown to be flexible, powerful and fast.

Mungi introduces two abstractions that are not commonly used: single-address-space and capability-based protection. A single address space removes the address-space obstacles to sharing. The benefits that are derived from this freedom have to be realised at the higher layers.

Convincing people to embrace a new paradigm is difficult. Mungi presents a whole set of abstractions that are different to what people are used to. Having a single address space removes the need for the programmer to think about run-time issues in communcation between applications, information is just passsed in situ. Likewise the protection system based on capabilities will look and feel different to the dominant protection paradigm, ACL based protection. Capabilities and a SASOS are well matched, both provide a global naming scheme for objects. Capabilities in Mungi are powerful pointers that not only give the globally valid location of data, but also provide the rights to access the data. Using capabilities instead of pointers in Mungi will allow users to apply the principle of least privilege naturally. With the right interface to the protection system, users will apply good protection principles naturally.

Both single-address-space systems and capability-based systems will have to overcome the natural resistance to move from the commonly accepted ways of doing things. There are substantial benefits in the adoption of the ideas presented in this thesis. Only time will tell if either abstraction will be adopted by the mainstream.

Appendix A

Benchmarking Details

A.1 Environment

Testing and debugging was done on two platforms

- SGI Indy.
- U4600 (developed at UNSW).

Both systems have a 100MHz R4x00 single issue CPU and have 64Mb of RAM. Both systems have two-way set associative 16kB instruction and data caches. Figures are given in μs, which translate to about 100 instructions on the 100MHz processors.

A.2 Software

- 64-bit compiler: gcc 2.7.2.3, modified to produce 64-bit code that could be assembled by the assembler. Compiler flags: -O2 -G 0 -mips3
- Assembler: IRIX 6.2 assembler
- Linker: IRIX 6.2, ld. Linker flags: -n -64 -non_shared

The PROM on both architectures are set up to only load elf-32 files. A utility was written which would pack an arbitrary number of elf-64 object files into one elf-32 file.

A.3 Measurement Details

This section will describe the methodology that was used to produce the figures in Tables 8.1 and 8.2.

Time measurements are done by writing a timestamp into memory before and after each operation. At the conclusion of the benchmark the timestamps are printed to the terminal and the differences between timestamps are calculated. The overhead of the timestamp

operations was measured and varied between 4.5 and 7 μs, this is probably due to cache effects (See Section A.4 below). Obviously if the time of an operation was near this order of magnitude we could only get accurate figures by measuring the total time of multiple iterations.

A.3.1 Thread measurements

The **ThreadCreate** and **ThreadDelete** figures were run 20 times with the average figures presented. The **ThreadCreate** timings are generated by using the code fragment below. A timestamp is taken before the call to ThreadCreate. The newly created thread (running D_Dummy) then sends a message back to the parent thead. When this message is received another timestamp is taken. The difference between the two timestamps is taken as the time for thread creation.

The average as reported was measured to be 48 μs with a standard deviation of 0.9μs.

```
/* from benchmarking code */

  for(k=0;k<B_NewThread_Iter;k++) {
    tstamp(4*k+4);
    Thread = ThreadCreate(D_Dummy,&D_Stack[B_NewThread_SSize]);
    l4_mips_ipc_wait(&next,L4_IPC_SHORT_MSG,
                     &mipsmsg,
                     L4_IPC_NEVER,&dope);
    tstamp(4*k+5);
    ThreadDelete(Thread,1);
  }


char D_Stack[B_NewThread_SSize];

void
D_Dummy(void)
{
  l4_threadid_t parent = {0x20080000000a0801}; /* hardwired  TID */
  l4_msgdope_t dope;
  l4_ipc_reg_msg_t mipsmsg;

  l4_mips_ipc_call(parent,L4_IPC_SHORT_MSG,&mipsmsg,
                   L4_IPC_SHORT_MSG,&mipsmsg,
                   L4_IPC_NEVER,&dope);

}
```

A similar methodology is used to measure the cost of thread creation in a different protection domain. The average is the reported 600 μs but the standard deviation is significant at 70μs. The reason for this large variation can be attributed to the fact that a thread in a new protection domain must first allow the pager to map its pages into the protection domain. In the benchmark tests, a thread in a new protection domain takes six residency faults before it was able to execute the IPC code in order to notify the parent that the thread had started.

A.3.2 Other benchmarks

The null system call and repeated PDX are both calls that are expected to be in the same order of magniture as the time taken to timestamp. As a result the figure for both PDX

and the null system call are the result of one million iterations. The object creation and deletion benchmarks were run 1000 times with the total time taken for the 1000 operations taken.

As the validation code measured some of the internal timings of the kernel, the benchmarks were generated by inserting timestamping code in the kernel. The overall pagefault timings were generated by timing the time taken to access a memory location that was known to be unmapped. The "In-cache" benchmark was run 30 times with the average of 26.5μs and a standard deviation of 3.4μs.

A.4 Cache effects

Both machines have a 2-way set-associative instruction and data cache. Each cache is 16Kb, giving 8Kb for each level of associativity. The cache is organised as 256 lines of 32 bytes for each of the two levels of associativity. There was a significant difference in the cache refill times on the U4600 and the SGI Indy. On the U4600 it took on average about 14 cycles to load the first 4 bytes after a cache miss. This overhead was measured as about double on in the SGI, on average 28 cycles for the same result [Elp98].

Due to small size of the L4 μkernel and Mungi, there was a significant variance in benchmark timings depending on the cache alignment of user data and code relative to the kernel. As an example, the times taken for repeated PDX varied from 12-20 μs depending on the alignment of code and data. This indicates that further performance gains are possible by carefully aligning Mungi's code and data relative to L4's to avoid conflict between them. This would free up the second level of associativity for user code and thereby eliminating conflict misses on system calls.

Bibliography

[ABLN85] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[AGS83] Stanley R. Ames, Morrie Gasser, and Roger R. Schell. Security kernel design and implementation: An introduction. *IEEE Computer*, 16(7):14–22, July 1983.

[Alg95] Algorithmics, 3 Drayton Park, London N5 1NU, UK. *P–4000i User's Manual*, 1.10 edition, May 1995.

[APW86] M. Anderson, Ronald Pose, and Chris S. Wallace. A password-capability system. *The Computer Journal*, 29:1–8, 1986.

[AW88] M. Anderson and Chris S. Wallace. Some comments on the implementation of capabilities. *The Australian Computer Journal*, 20(3):122–33, 1988.

[Ber80] Viktors Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–250. ACM/IEEE, May 1980.

[BL76] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., March 1976.

[BN89] David F.C Brewer and Michael J. Nash. The Chinese Wall security policy. In *Proceedings of the Symposium on Security and Privacy*, May 1989.

[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 OS Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995. ACM.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 120–133, Asheville, NC, USA, December 1993.

[CD94] David R. Cheriton and K. Duda. A caching model of operating system functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 14–17, Monterey, CA, USA, November 1994. USENIX/ACM/IEEE.

[Cha95] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures.* PhD thesis, University of Washington, 1995.

[CJ75] E. Cohen and D. Jefferson. Protection in the HYDRA operating system. In *Proceedings of the 5th ACM Symposium on OS Principles*, pages 141–59, 1975.

[CLBHL92] Jeff S. Chase, Hank M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 80–85, Key Biscayne, FL, USA, 1992. IEEE.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[Coo78] Douglas Cook. The cost of using the CAP computer's protection facilities. *Operating Systems Review*, 12(2):26–30, April 1978.

[Cor89] Corporations law s.1002. *Commonwealth of Australia, Consolidated Acts*, 1989.

[CRJ87] R.H. Campbell, V. Russo, and G. Johnston. Choices: The design of a multiprocessor operating system. In *USENIX C++ Workshop*, pages 109–23, Santa Fe, NM, USA, 1987.

[CS92] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17:1–31, 1992.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the Symposium on Security and Privacy*, pages 184–194, April 1987.

[DD68] R.C. Daley and J.B. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, May 1968.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, David Hulse, Anders Lindström, Stephen Norris, John Rosenberg, and Francis Vaughan. Protection in the Grasshopper operating system. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, Workshops in Computing, pages 60–78, Tarascon, France, September 1994. Springer-Verlag.

[Dig92] Digital Equipment Corp., Maynard, MA, USA. *Alpha Architecture Handbook*, 1992.

[DoD85] DoD National Computer Security Center. *Department of Defence Trusted Computer System Evaluation Criteria*, December 1985. Standard DOD 5200.28-STD.

[Dv66] J.B. Dennis and E.C. van Horn. Programming semantics for multiprogrammed computers. *Communications of the ACM*, 9(3):145–155, 1966.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. L4 reference manual — MIPS R4x00 — Version 1.0. Technical Report UNSW-CSE-TR-9709, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. Latest version available from http://www.cse.unsw.edu.au/~disy/.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 251–266, Copper Mountain Resort, Co, USA, December 1995. ACM.

[Elp98] Kevin Elphinstone. Private Communication, 1998.

[ERHL96] Kevin Elphinstone, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Supporting persistent object systems in a single address space. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, pages 111–119, Cape May, NJ, USA, May 1996. Morgan Kaufmann.

[Fab74] R.S. Fabry. Capability-based addressing. *Communications of the ACM*, 17:403–412, 1974.

[FCL93] Michael J. Feely, Jeffrey S. Chase, and Edward D. Lazowska. User-level threads and interprocess communication. Technical Report 93-02-03, Department of Computer Science and Engineering, University of Washington, 1993.

[GBY90] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 2nd edition, 1990.

[Han97] Steven Hand. ESPRIT LTR 21917 (Pegasus II): Deliverable 2.3.1: Virtual address management. Technical report, University of Cambridge Computer Laboratory, July 1997. Available from http://www.cl.cam.ac.uk/Research/SRG/pegasus/papers/vam.ps.gz.

[Hei91] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems, Inc., Sunnyvale, CA, USA, 1st edition, 1991.

[HERV94] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single-address-space operating system. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 271–80, Christchurch, New Zealand, January 1994.

[HHL+97] Herrmann 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 OS Principles*, pages 66–77, St. Malo, France, October 1997. ACM.

[HLR98] Gernot Heiser, Fondy Lam, and Stephen Russell. Resource management in the Mungi single-address-space operating system. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 417–428, Perth, Australia, February 1998. Springer-Verlag. Also available as UNSW-CSE-TR-9705 from http://www.cse.unsw.edu.au/school/research/tr.html.

[Hog88] Carole. B. Hogan. Protection imperfect: The security of some computing environments. *Operating Systems Review*, 22(3):7–27, 1988.

[HR91] Frans A. Henskens and John Rosenberg. A capability based distributed shared memory. *Australasian Computer Science Conference*, 14(29):1–12, 1991.

[HSH81] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Symposium on Computer Architecture*, pages 341–348. ACM/IEEE, May 1981.

[HVER97] Gernot Heiser, Jerry Vochteloo, Kevin Elphinstone, and Stephen Russell. The Mungi kernel API/Release 1.0. Technical Report UNSW-CSE-TR-9701, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 1997. Latest version available from http://www.cse.unsw.edu.au/~disy/.

[JLI98] Trent Jaeger, Jochen Liedtke, and Nayeem Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 143–158, Berkeley, January26–29 1998. Usenix Association.

[Jon80] A.K. Jones. Capability architecture revisited. *Operating Systems Review*, 14(3):33–5, 1980.

[KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single-address-space operating systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–86, 1992.

[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 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.

[LCC+75] R. Levin, E.S. Cohen, W.M. Corwin, F.J. Pollack, and W.A. Wulf. Policy/mechanism separation in HYDRA. In *ACM Symposium on OS Principles*, pages 132–40, 1975.

[Lee89] Ruby B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for efficiency). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997. IEEE.

[Lie92] Jochen Liedtke. Fast thread management and communication without continuations. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 213–221, Seattle, WA, USA, April 1992.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–88, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[Lip75] Steven. B. Lipner. A comment on the confinement problem. In *ACM Symposium on OS Principles*, pages 192–196. ACM, 1975.

[LLA+81] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fisher, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the 8th ACM Symposium on OS Principles*, pages 148–159. ACM, 1981.

[LMB+96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, 1996.

[MB80] G.J. Myers and B.R.S. Buckingham. A hardware implementation of capability based architecture. *Operating Systems Review*, 14(4):13–25, 1980.

[MM96] Ashok Malhotra and Steven J. Munroe. Schema evolution in persistent object systems. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, pages 194–204, Cape May, NJ, USA, May 1996. Morgan Kaufmann.

[MT86] Sape J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–299, 1986.

[Nel91] G. Nelson. *Programming in Modula-3*. Prentice Hall, 1991.

[Nes87] Dan M. Nesbit. Factors affecting distributed system security. *IEEE Transactions on Software Engineering*, 13:233–248, 1987.

[NT95] Windows NT platform gets C2 evaluation. URL, 1995. http://www.microsoft.com/syspro/technet/boes/winnt/prodfact/c2.htm.

[NW77] R.M. Needham and R.D.H. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on OS Principles*, pages 1–10. ACM, November 1977.

[Org93] E.I. Organic. *A programmer's view of the Intel 432 System*. McGraw-Hill, 1993.

[OSD94] Radical operating system structures for extensibility: A panel session. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 195–199, November 1994.

[Pea98] Conrad Parker and et al. U4600 user's manual. Technical Report UNSW-CSE-TR-9803, School of Computer Science and Engineering, University of New South Wales, 1998.

[Pik91] Rob Pike. $8\frac{1}{2}$, the Plan 9 window system. In *Proceedings of the 1991 Summer USENIX Technical Conference*, pages 257–65, Nashville, TN, USA, 1991.

[PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *UKUUG*, pages 1–9, London, UK, July 1990.

[RA85] John Rosenberg and David Abramson. MONADS-PC—a capability-based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, volume 1, pages 222–31. IEEE, 1985.

[RDH+80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, Willian C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23:81–92, 1980.

[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.

[RJO+89] R.F. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: a system software kernel. *Spring COMPCON*, pages 176–8, 1989.

[RSE+92] Stephen Russell, Alan Skea, Kevin Elphinstone, Gernot Heiser, Keith Burston, Ian Gorton, and Graham Hellestrand. Distribution + persistence = global virtual memory. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 96–99, Dourdan, France, September 1992. IEEE.

[RTY+88] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, C-37:896–908, 1988.

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17:388–402, 1974.

[Sch96] S. Schönberg. The L4 microkernel on Alpha - design and implementation. Technical Report 407, Cambridge University, 1996.

[SESS96] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–228, November 1996.

[Sha97] J.S. Shapiro. EROS: A capability system. Technical Report MS-CIS-97-04, Department of Computer and Information Science, University of Pennsylvania, 1997.

[SLM90] Michael Scott, Thomas LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the 2nd Conference on Principles and Practice of Parallel Programming*, pages 70–78. ACM, March 1990.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, CO, USA, 1996.

[Sun] Sun Microsystems Inc, 901 San Antonio Road, Palo Alto, CA 94303 USA. *SunSHIELD Basic Security Module Guide*.

[SW97] J.S. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS-97-26, Department of Computer and Information Science, University of Pennsylvania, 1997.

[SZBH86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8:419–490, 1986.

[TM84] Andrew S. Tanenbaum and Sape Mullender. The design of a capability-based distributed operating system. Technical Report IR-88, Vrije Universiteit, November 1984.

[TM94] Nick Thieberger and William McGregor, editors. *Macquarie Aboriginal Words*. Macquarie Dictionary, Macquarie University, NSW 2109, Australia, 1994.

[TMvR86] Andrew S. Tanenbaum, Sape J. Mullender, and Robert van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 558–563. IEEE, May 1986.

[TR74] Ken Thompson and Dennis M. Ritchie. The UNIX time-sharing system. *Communications of the ACM*, 17:365–375, 1974.

[VERH96] Jerry Vochteloo, Kevin Elphinstone, Stephen Russell, and Gernot Heiser. Protection domain extensions in Mungi. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, pages 161–165, Seattle, WA, USA, October 1996. IEEE.

[VRH93] Jerry Vochteloo, Stephen Russell, and Gernot Heiser. Capability-based protection in the Mungi operating system. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 108–15, Asheville, NC, USA, December 1993. IEEE.

[Wal90] Chris S. Wallace. Physically random generator. *Computer Systems Science & Engineering*, 5:82–88, 1990.

[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.

[Wil96] Tim Wilkinson. Private Communication, 1996.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, December 1993. ACM Press.

[WM96] Tim Wilkinson and Kevin Murray. Evaluation of a distributed single address space operating system. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 494–501, Hong Kong, May 1996. IEEE.

[WMR+95] Tim Wilkinson, Kevin Murray, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Single address space operating systems. Technical Report UNSW-CSE-TR-9504, University of NSW, University of NSW, Sydney 2052, Australia, November 1995.

[WP90] Chris S. Wallace and Ronald D. Pose. Charging in a secure environment. In J. Rosenberg and J.L. Keedy, editors, *Security and Persistence*, pages pp 85–97. Springer-Verlag, 1990.

[WSO+92] Tim Wilkinson, Tom Stiemerling, Peter E. Osmon, Ashley Saulsbury, and Paul Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, pages 316–319, Barcelona, Spain, 1992.

[X90] *X Protocol Reference Manual*, 1990.

[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 OS Principles*, pages 63–76. ACM, 1987.

[Zil92] Zilog Inc., Campbell, CA. *SCC User's Manual*, 1992.