Protection Domains and Threads in Mungi

Daniel Cutting
danc@cse.unsw.edu.au

Supervisor: Gernot Heiser
Assessor: Andrew Taylor

COMP4914 Computer Science Honours 1998

Abstract

The existing model of execution in Mungi is unnecessarily bound to the notion of a task. Each task has a protection domain which all threads in that task must use. However, it would be more useful if threads could run in whichever protection domain best served their access requirements. This paper proposes the removal of the Mungi task and the introduction of a more flexible binding of protection domains and threads, so as to make execution and protection orthogonal.

Protection domains are at present transient structures associated with tasks. This paper also proposes a mechanism whereby protection domains can be made persistent and distributed across the system.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is a SASOS?

SASOS (sometimes written SAOS) is an acronym for single address space operating system. Whereas in a conventional multiple address space operating system (MASOS) each process has its own address space, all data and processes in a SASOS reside in one very large virtual address space. This has only become practical with the advent of 64-bit processors (such as the MIPS R4x00, on which Mungi is presently implemented) which allows for a staggering 16 billion gigabytes of virtual memory into which *all* data objects in the system may be placed. This is large enough for a building-sized network of computers.

1.2 What is Mungi?

Mungi is the UNSW implementation of a SASOS. It is designed to be distributed over many physical nodes, with each node running its own copy of the Mungi kernel. Though many physical nodes may be part of the system, each sees the same virtual address space, with the same objects at the same locations.

The following is taken from the Mungi manifesto.

> Mungi is based on the following principles:
>
> - a single, flat address space,
> - orthogonal persistence,
> - a strong but unintrusive protection model.
>
> The single address space incorporates all memory objects on all nodes in a distributed system. There [is] . . . no file system. Any memory

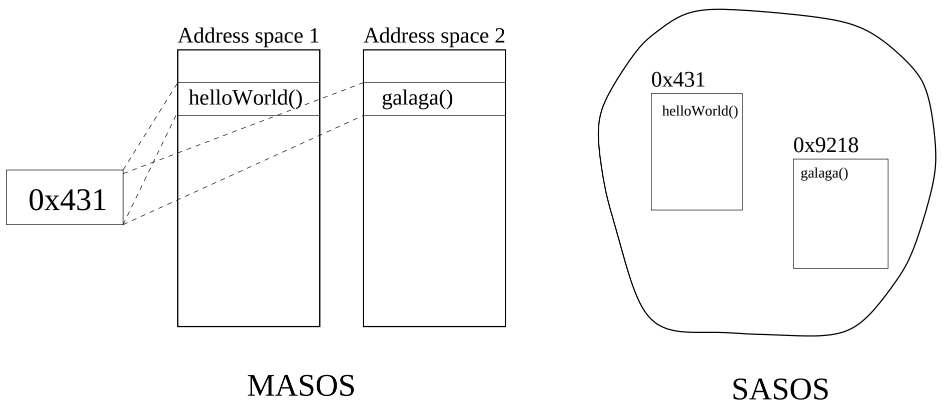

Figure 1.1: Basic difference between a MASOS and SASOS. In a MASOS, an address maps to different data depending on the context. In a SASOS, an address consistently maps to the same data.

> object is potentially persistent, i.e. can outlive its creator process. All objects are uniquely identified by their virtual address, which is a 64-bit number. . . . Sharing is trivially achieved by exchanging addresses.

Every memory object in the virtual address space is protected by a password. This means that only when an entity presents a password to a particular memory object is it granted access. Different types of access are available to the entities (such as `READ`, `WRITE`, and `EXECUTE` which are similar to the UNIX modes).

Because Mungi is a distributed operating system, the users do not know where their data physically resides or where their programs physically run.

Chapter 2

The existing execution and protection model

2.1 Objects

An *object* in Mungi is a contiguous, page-aligned region of the virtual address space. An object can contain anything from a word processing document to a game executable. There is no internal structure whatsoever assigned to objects by the kernel, although applications are free to assign types and structure if desired.

When an object is created, it is given an address in the virtual address space, and this remains constant for the life of the object, be it milliseconds, days, or years. The address of the object is its unique name and can be used by all programs in the system at all times to refer to the object. Since Mungi guarantees this consistent naming scheme, sharing an object between two programs is very simple.

Most objects that are created have an *object descriptor* added to the *object table* (OT), a system-wide data structure containing information about the objects in the virtual address space. If desired, an object can be omitted from the OT. In general, this would make the object unshareable to programs other than the object's creator and would be used for very light-weight, transient objects.

2.2 Protection

2.2.1 Capabilities and Capability Lists

Since all objects exist in the same address space, there needs to be some way of protecting them from one another. This protection is in the form of a *password capability* (or just capability for short). These are basically keys to objects and

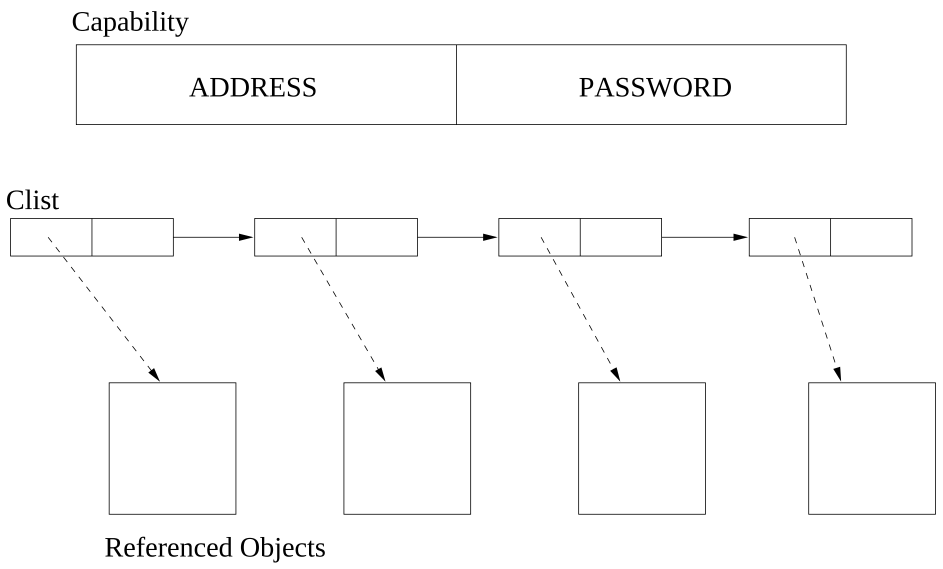

Figure 2.1: Structure of capabilities and clists.

comprise a `<base_addr,password>` tuple. To hold a capability to an object means to know the base address of that object, and to have a password which is valid for that object. All assigned capabilities are stored in the OT along with the object's descriptor.

Password capabilities are, by their nature, *sparse* capabilities. This means they are protected from forgery by the sparsity of their passwords (which in Mungi are generally represented as random 64 bit numbers). Sparse capabilities are pure data. They are not specifically protected or handled by the system — they have no `copy` bit for instance as in Hydra, [WLP75]. Thus there is no way for the system to know who holds the capabilities to a certain object as they can be passed from program to program without kernel intervention.

Many capabilities can be created for a single object, each having different types of access (and different passwords). Mungi supports several types of access to objects: specifically `READ`, `WRITE`, `EXECUTE`, `DESTROY` and `PDX`. A capability can be any combination of these types. For instance, a `READ/WRITE` capability for an object allows both examination and modification of its data. A `DESTROY` capability allows the holder to delete the object from the system. A `PDX` capability is a special type called *protection domain extension* allowing programs to execute privileged code[1].

Users are able to build up *capability lists* (clists) that hold many of these capabilities to objects. This allows the user to conveniently group similar types of capabilities. For instance, they may wish to construct a clist containing all of the

[1]Similar to the `setuid` concept in UNIX.

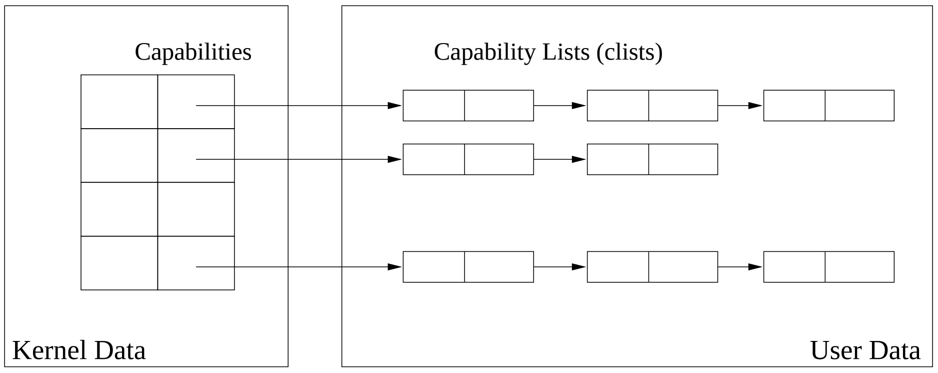

Figure 2.2: An APD and some associated clists.

capabilities to their word processing documents, and a different clist containing those to their games. Clists are just ordinary objects like everything else and as such, must also be accessed through capabilities.

2.2.2 Active Protection Domains

An *active protection domain* (APD) is a kernel data structure consisting of capabilities to some user-defined clist objects. They are associated with programs so that a user does not have to explicitly provide capabilities for objects to which the program needs access. Whenever a program tries to access a page of an object it has not accessed before (this is known as a *protection fault*), the kernel searches the program's APD to verify it is allowed to do so. It does this by traversing the user clists pointed to by the APD, and if it finds a capability to the requested object in one of these clists, access is granted to the program.

As mentioned above, clists are user-defined (and maintained) objects and as such, can be modified without kernel intervention. This means, for instance, that a user could add capabilities for extra games to their games clist without calling the kernel. When the user tried to access the new games the kernel would then be able to find them in the modified clists.

However, adding an entirely new clist to an APD can only be achieved through a system call, as it must be validated by the kernel, and a capability for it inserted into the APD. Similar operations such as removing clists must also be performed by the kernel via a system call. This is to ensure that the APD always contains valid information. Since the capabilities in the APD for the clists are not validated on a protection fault, the kernel trusts them to be accurate. This trust cannot exist unless the APDs are fully maintained by the system. If a clist is deleted by a user without removing it from the APD first, the APD will contain an invalid clist. For this reason, the APD is periodically *flushed* (revalidated) by the kernel. Thus, the kernel guarantees that there is some maximum amount of time (the time between flushes) that the APD may be inaccurate.

APDs are allocated on the kernel heap and are fully maintained by the kernel. They are not in general directly accessible by user programs or other kernels on the network.

2.3 Execution

2.3.1 Threads

Threads are the basic execution abstraction in Mungi and are scheduled by the kernel. They can basically be thought of as active programs in the system. When created, a thread begins execution at a given address and uses a (pre-allocated) object as its stack. Every thread belongs to exactly one task.

2.3.2 Tasks

A *task* is basically a set of threads. They are arranged in an hierarchical tree fashion such that, in general, each task has one parent and possibly many children. Any task can spawn children tasks which then appear beneath them in the hierarchy.

Tasks are made up of an APD, a stack, an environment and a set of threads. The APD may be either specified by the parent task (as a set of clists passed to the system call), or a copy of the parent's APD can be inherited. The stack is created by the kernel when the task is created, and is used by the initial thread in the task which is also automatically created. This thread begins execution of the instructions assigned to the task. Finally, an environment is allocated which may be in a separate object or in the first thread's stack object. Copies of the environment are inherited from parent tasks. When a task is deleted, all child tasks are also deleted (unless adopted by the task performing the deletion). A task can only be deleted by tasks above it in the task hierarchy.

This model of tasks and threads is familiar as it is the basic model used by most conventional operating systems such as UNIX.

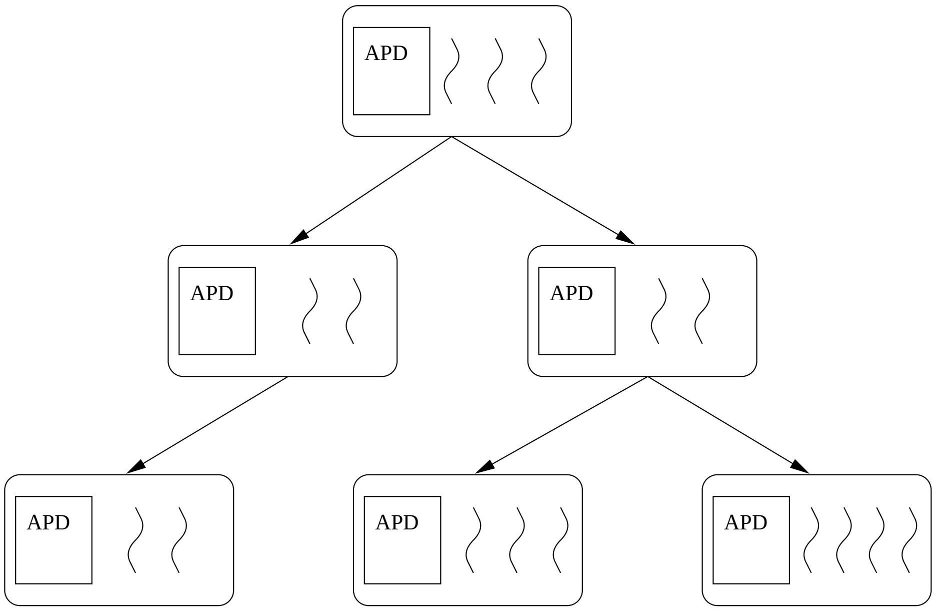

Figure 2.3: The existing logical model of tasks in Mungi — each task has a set of threads and an APD.

Chapter 3

A revised execution and protection model

3.1 Guiding design criterion

Mungi has excellent micro-benchmarking results when compared to other operating systems such as Linux, IRIX and Opal (another SASOS developed at the University of Washington). In some operations (such as task creation and deletion), Mungi outperforms these other operating systems by an order of magnitude on equivalent hardware. The primary aim of this thesis is to demonstrate that a more flexible execution and protection model can be implemented in Mungi *without significantly affecting performance*. This has been an important consideration at all times.

3.2 Impetus for change

The existing model seems unnecessarily restrictive in several ways. There are two main flaws:

1. Protection is intimately mingled with execution in the existing model, but this is unnecessary in the SASOS design. As shall be seen, Mungi allows us to fully decouple protection from execution to provide two orthogonal abstractions which will in turn allow us to implement a more flexible execution model.

2. It does not allow very effective distribution — something that is obviously at odds with the concept of a distributed operating system. Mungi tasks are

Operation	*Mungi*	*Linux*	*Irix*	*Opal*
Null system call	4.6	6.3	7.7	>88
Cross-domain call	10–20	161	450	133
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
Task create	600	N/A	5,600	650
Task delete	310	N/A	1,550	2,300
Task create + delete	910	34,163	8,150	2,950
Object create	60	N/A	N/A	315
Object delete	150	N/A	N/A	900
Object access	90	45	252	239?
Object create + access + delete	300	447	2470	2,100
Page fault/map	25	N/A	N/A	N/A

Table 3.1: Micro-benchmark timings of existing task-based model on SGI hardware (in μs).

directly mapped to L4 tasks[1]. This means the protection domains cannot be easily distributed amongst the various nodes. They are also transient, existing only as long as the task that uses them.

The first point can be rectified by restructuring threads and protection domains in the system. The second can be solved by developing a system whereby protection domains can be persistently defined and reused in a distributed manner.

3.3 Restructuring threads and protection domains

Logically, protection domains can be thought of as sets of objects. For a program to run in a protection domain means that it can access all of the objects in that set. Protection domains prevent access to objects outside the set.

The Mungi execution model (specifically the concept of tasks) has obviously been copied from a MASOS model. In a MASOS, a task (or process) is defined by its address space and the threads running within. The address space is really the way in which protection is enforced in a MASOS — programs running in different address spaces are unable to access one another's data since the same address is de-referenced differently depending on the context.

[1]Tasks are the protection abstraction of the L4 microkernel on which Mungi is constructed

Although the threads within a task in a MASOS often have similar access requirements, they are usually not identical. The reason that new tasks are not created for each thread with subtly different access requirements is that there is an enormous overhead in doing so. This cost is manifested both at creation time (when a new address space must be arranged) and when transferring data between the new and old tasks (via inter-process communication, for instance).

A SASOS allows us to forego these costs. Since an address in a SASOS refers to the same object regardless of the program that uses it, there is no need to “swizzle” data between address spaces. Creating a thread in a subtly different protection domain is also relatively inexpensive when compared with creating new address spaces.

Mungi’s existing execution model is in a sense emulating a MASOS. The APD associated with each Mungi task is emulating the address space of a MASOS task, and the threads are all executing within that “address space”. Since protection domains are no longer based on discrete address spaces, it would be more effective to tailor the protection domain of each thread rather than have it bound to the same context as its parent. Though necessary for reasons of efficiency in a MASOS, it is most definitely not necessary in a SASOS.

This argument is in favour of dispensing with the task-based model of execution in Mungi, since it imposes unnecessary (and effectively arbitrary) restrictions on the way threads and protection domains relate to one another. The real flexibility of protection domains comes from detaching these monolithic tasks from one another and instead connecting the elements that make them up (the threads and protection domains) in a more natural manner. The two abstractions should be as orthogonal as possible and this is simply not the case in the existing task-based model of execution.

After dispensing with the notion of a Mungi task, two orthogonal abstractions remain : the execution abstraction (threads) and the protection abstraction (protection domains). Without tasks to bind these together, a new way of structuring these two abstractions within Mungi is required.

Processes in UNIX are hierarchically structured so as to provide a way of cleaning up executing programs. The fact that this hierarchy also arranges the address spaces (or protection abstractions) is incidental. Similarly, it does not make sense to hierarchically arrange Mungi protection domains in such a manner. Mungi does however require some mechanism for cleaning up executing programs. Since execution is now solely based on threads, it seems clear that Mungi should provide an hierarchy of threads as opposed to its current design of an hierarchy of protection domains (the Mungi tasks).

Protection domains can easily be incorporated into this hierarchy. Each thread can be directly associated with a protection domain rather than being indirectly associated through a Mungi task.

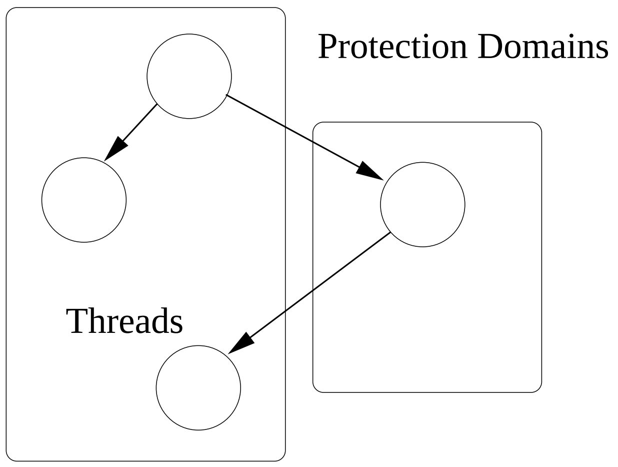

Figure 3.1: Logical view of restructured protection domains and threads.

3.4 PD objects

Mungi already has the concept of an APD well defined. This is the structure associated with a set of threads that is searched whenever one of the threads tries to access an object in the global address space. However, it is quite a transient structure, existing only as long as the threads using it exist. The fact that there is an *active* protection domain perhaps implies the idea of an *inactive* protection domain should be explored. This would be a way of defining a set of objects which could then be instantiated, or made active when a thread began execution under it. These inactive protection domains would effectively be inert, persistent sets of objects which could exist without threads using them.

This can be achieved by introducing protection domain (PD) objects. A PD object is an ordinary user-level object in the global address space comprising a collection of clists. In the same way that a clist is a set of objects, a PD object is a set of clists.

This is essentially the way APDs are currently implemented, the real difference being that PD objects are globally accessible objects — they are fully distributed, which means that any thread on any node is able to make use of them. Furthermore, since they are objects, they can be made persistent and may be used repeatedly as the protection domain for new threads. Neither of these benefits are directly possible with the current implementation of APDs, as APDs are bound to a particular node and are automatically destroyed when threads are no longer using them.

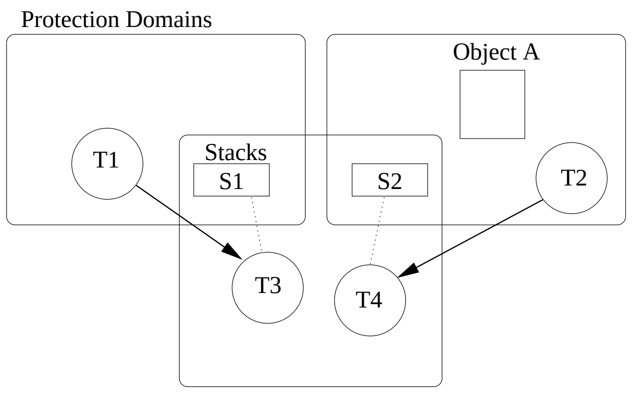

Figure 3.2: Unintentional sharing of information — T2 pushes information about Object A onto stack S2 which thread T3 can read.

3.4.1 PD objects and APDs

At first glance, it may appear that PD objects could be used directly for all object access validations. That is, each thread could be associated with a PD object that would be searched when the thread caused a protection fault. In this sense, they would basically replace the existing APDs. However, this turns out to be a less than perfect design.

Each thread in the system (generally) requires a stack object for it to execute. If the thread were to be started using a PD object, its stack would need to be a part of the domain defined by the object. This would mean that before every new thread was created, the caller would have to add the new thread's stack object to a clist referenced by the appropriate PD object. This is obviously not desirable because the clists would quickly grow to include many stack objects. Furthermore, since stack objects are generally transient objects (intended to exist only as long as the thread for which they were created), the clists making up the PD object would need to be stripped of the capabilities for these objects every time a thread was deleted, or suffer increased search time on protection faults as the clists grew to ridiculous lengths.

There is one final reason for not including the stacks in the PD objects and this is more of a protection issue than a practicality issue.

Threads that have been created using the same PD object from threads in different, untrusting protection domains, should not be able to access one another's stacks as this could lead to unwanted violations. Specifically, each of the threads would be able to examine the others' stacks which may contain sensitive information from their respective parents. This information, whilst part of the parents' protection domains, should not necessarily be accessible by all of the children running under the PD object. The eavesdropping threads are potentially able to

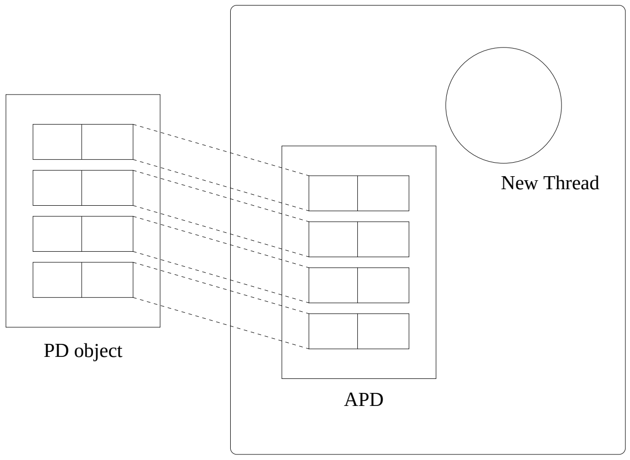

Figure 3.3: The capabilities in the PD object are copied into the new thread's APD. After this initial operation the PD object is no longer involved with the execution of the new thread.

access information they were never supposed to. However, it can be assumed that a thread created by another thread in the same protection domain can be trusted by its parent, and as such it is not necessary to hide the stack in this case.

The solution to these problems is to retain the concept of an APD. This would remain exactly as it is in the current model — it is the structure that would be searched when a thread caused a protection fault. When a new thread is created using a PD object, the clists in that object would be copied into the new thread's APD. In this sense then, PD objects are really *templates* for APDs. Since the PD object is copied into the new APD, the new thread's stack object does not need to be referenced by the clists making up the PD object itself. It can instead be inserted into the thread's APD during thread creation. This makes sense because it doesn't seem appropriate to insert a thread's (generally) private stack into a distributed PD of which others can make use. If a thread ever needed to make its stack available to other threads, it could be explicitly added to a clist referenced by a PD object.

One point to note about this design is that the PD object is copied into the thread's APD and hence any subsequent changes to the PD object do not affect the APD of any thread that has already been created.

The logical protection domain of new threads is specified by the PD object — this is the set of objects that the thread is able to access. Semantically, the thread's APD is this logical protection domain plus its own stack. Hence, it is possible to have many different APDs for different threads running in the same logical protection domain.

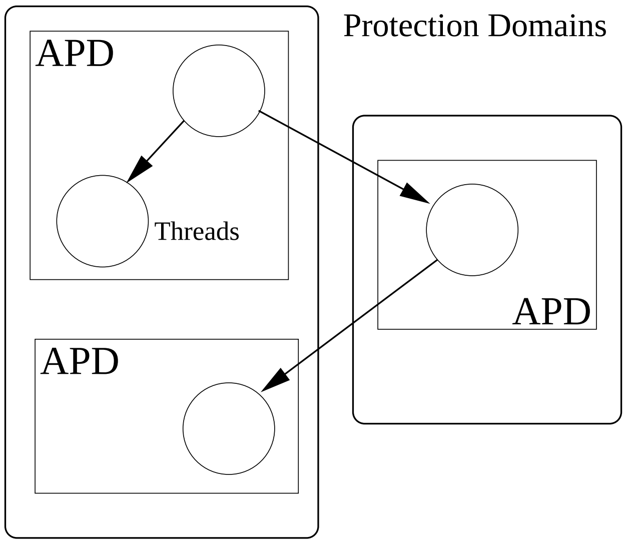

Figure 3.4: Relationship between logical protection domains and APDs.

3.4.2 Kernel support for PD objects

One of the reasons for introducing PD objects is to provide a way in which protection domains can be distributed. To this end, it would be desirable to allow a thread with access to a PD object to start new threads running within the domain *without* requiring it to have access to the clists by which it is defined. In this way, a thread could create a PD object allowing access to a set of objects and then give the ability of creating a thread within this domain to another thread. Although that thread could create new threads which could access the objects specified by the PD object. it would be unable to give away any actual capabilities for those objects.

Obviously, for this to be achieved, a thread should not be able to directly read PD objects as it would then be able to read the capabilities for the clists that make it up, and hence read the capabilities for objects contained in the clists. Hence, no thread is allowed to hold a `READ` capability for a PD object.

Although there is no reason why threads should not be able to write directly to PD objects in the revised model, there are reasons for disallowing this if the model were to be optimised. These optimisations are discussed in section 3.4.3. In anticipation that these optimisations will be included in a future version of Mungi, `WRITE` access to PD objects has been disallowed in this version.

Hence threads are not allowed to directly read or write to PD objects. For this reason, kernel support is required to modify them. Specifically, users should be able to insert clists into and remove clists from PD objects with system calls.

If a user cannot hold a `READ` or `WRITE` capability to a PD object, some other kind of capability is required which will grant the holder the ability to perform system calls to manipulate PD objects. The `EXECUTE` capability is an appropriate choice as it fits in with the idea of threads using PD objects to create new threads. Any user that holds an `EXECUTE` capability for a PD object will be allowed to modify that object by inserting and removing clists.

Since PD objects are similar to APDs, it would be useful to also perform other operations on them. This includes reading the addresses of the clists that make up the PD object, locking the PD object (or particular slots thereof), and verifying whether a particular object is accessible with a certain mode within the PD. These operations correspond directly to the APD system calls `ApdGet()`, `ApdLock()` and `ApdLookup()`.

Thus there is now a set of system calls which allow users to operate on existing PD objects to which they have access, but there is no way for them to create new PD objects. Obviously since PD objects are system-maintained (ie. not directly modifiable by users), the kernel will also need to provide ways of creating and deleting these objects. An ordinary `ObjCreate()` is inadequate as this returns an `OWNER` capability to the caller which includes both `READ` and `WRITE` access. Hence, a new system call is required which will specifically create PD objects. To complete the functionality, a way of deleting PD objects is required. There is nothing to stop the kernel from returning a capability including the `DESTROY` right when a PD object is created. This would allow the PD object to be destroyed through an ordinary `ObjDelete()` system call. However, for reasons of consistency, it seems more appropriate to provide a specific system call to delete PD objects. Furthermore, future versions of Mungi may wish to incorporate specific cleanup code when a PD object is deleted (c.f. section 4.5.1). For these reasons, an explicit PD object deletion system call is included.

Most of the above operations are required for both PD objects and APDs. The system could support them by providing two sets of system calls — one relating to PD objects, and the other relating to the caller's APD. The existing Mungi APD system calls currently perform the latter duties. However instead of having two sets of calls which are virtually identical, it is simpler to manage these two types of domains (active and inactive) through the same set of system calls. To conform to the new nomenclature, the existing APD system calls become PD system calls. They may be given an explicit PD object to operate on or, if no object is given, the caller's APD may be assumed instead. This allows threads to operate on any PD objects to which they have access, and also their current APD, as is supported in the existing version of Mungi.

There is one more thing necessary for the kernel to fully support PD objects. With just the above support, a user could supply any object to which they had `EXECUTE` access to a PD object system call and have the kernel treat it like a PD object. This is potentially dangerous as it would, for instance, allow a user with

`EXECUTE` but not `READ` access to an object to view parts of it (via the system call returning the addresses of clists that make up a PD object). The system thus needs to be certain it is operating on a PD object.

This can be achieved by including a flag in the object descriptor of the PD object in the object table. This flag would be part of the existing flags field, and would specify whether the described object is a PD object or not. This is not departing from the current "no-structure" object scheme of Mungi, as there are already several flags specifying similar object types[2]. As PD objects are system-maintained, it's not abhorrent to include a PD flag. When a thread is created with a given PD object, or when the kernel is asked to modify a PD object, the kernel can check the flag in the object descriptor. If it is set, the operation will continue, and if it is not, the operation will fail.

When a PD object is created, the system returns an `EXECUTE` capability to the caller. However, it would also be useful to create an `OWNER` capability and have the kernel retain this. This may seem unnecessary, as the kernel has access to whatever it needs without capabilities, but the `OWNER` capability would be useful in some circumstances. Mungi is a distributed system and as such, the kernels often need to talk to one another. A frequent problem with a distributed operating system is having to trust a remote kernel to be reliable and safe. The `OWNER` capability could be used to establish trust between remote kernels regarding PD objects.

There is another reason for having an `OWNER` capability however and this is related to Mungi's method of resource management. Mungi manages its virtual memory with an economic model of resource allocation ([HLR98]). Each object in the system is associated with a bank account which is used to pay for the object. At regular (or irregular) intervals, a *rent collector* visits every object in the system and charges the associated bank account for the virtual memory it uses. If the bank account is out of funds, the rent collector needs to be able to destroy objects (this is a matter of policy of course, but it should be at least supported). PD objects, being user objects are subject to the same policies. The rent collector is an ordinary user thread and as such requires capabilities to perform its duties. For the rent collector to be able to destroy a PD object, it will thus need to be able to obtain a `DESTROY` capability for it. This must be stored in the object table as this is the structure the rent collector traverses on its periodic journeys. Hence, when a PD object is created, the kernel should generate a `DESTROY` capability (or a more generic `OWNER` capability) and store it into the OT along with the `EXECUTE` capability returned to the user. The kernel does not explicitly give this capability to any threads, although the rent collector and any other thread with `READ` access to the OT will be able to see it.

[2]Mungi specifies some objects as bank accounts which are used for resource management.

3.4.3 Optimisations

Mungi guarantees that there is a maximum amount of time that invalid protection information will be accepted as valid. For instance, each Mungi task in the current version has a validation cache which caches any successful validations to objects. The next time an access is attempted, the validation cache is checked first to see whether that object has already been validated. This is obviously a problem if the capabilities to the object in question are revoked after it has already been added to a task's validation cache. Potentially the task could access the object for as long as it wanted even though its access had been revoked. For this reason, the validation cache is periodically flushed and all subsequent accesses to objects must be fully validated (until they are again added to the validation cache). The period between flushes is the maximum amount of time that invalid protection information can be accepted as valid.

The method discussed above of copying PD objects into a new APD requires every clist referenced by a PD object to be validated whenever a new thread is created using it. This is necessary because the system cannot be sure when the clist was last validated. It is possible however to design the system in such a way that the system does know when it was last validated.

When an entire PD object's contents are validated (during a thread creation), it could be added to a cache of some kind along with a timestamp. The next time a thread is created, the cache could be searched. If the object is found, the timestamp could be examined to see whether it is within some limit (similar to the period between validation cache flushes) and if so, it could be assumed that all clists in the PD object are valid. In this case, the clists could be copied directly into the new thread's APD without performing a validation on each one. If the timestamp is outdated, the entire PD could be revalidated and recached.

This optimisation would increase the speed of creating a new thread using a PD object, but only if the PD object was frequently used. If it were an infrequently used PD object, then the overhead of using the cache would impact very marginally. The slight overhead certainly seems worthwhile.

3.4.4 Alternative APD design

The above model of PD objects and APDs is a template/instance relationship. This is so the APD can contain the capability for a stack object without having to include the stack in the clists referenced by the PD object. There is however, another way to implement the APD such that the stack is still separate from the the objects referenced by the PD object.

In this method, the APD is actually a two-part entity. It is made up of a single slot in the thread control block (TCB) which contains the capability for the clist containing the thread's stack, and a pointer to the PD object with which the thread was created. In this way, the PD object is not copied but referenced dynamically as needed. In contrast to the other design, any changes made to the PD object would be visible by the threads using it.

Implementation of this type of APD is more difficult, and fairly different to the design present in the existing version of Mungi. Whether or not a shared, dynamic APD is desirable is open for debate. The disadvantage of this would be that a thread's APD could change without it being realised due to another thread modifying the PD object. Since it is not clear what use this design would have, it was decided to implement the simpler of the two designs, as described above. It may however be interesting to explore this alternative approach in future versions of Mungi.

3.5 Execution

Actual execution under the revised model remains much as it was — each thread is associated with an APD (though this association is now more direct than in the previous version) which is searched when that thread causes a protection fault. The real differences are to do with the starting and stopping of programs. Users no longer start tasks — they start only threads. There are two ways to start new threads however : within the same active protection domain as the caller, and in a different protection domain altogether. Under the revised protection model, the two forms of thread creation are termed *intra-domain* and *inter-domain* thread creation.

In both cases, the caller creates a new object for the new thread to use as its stack (or makes sure there is enough space in an existing object). The caller then creates a new thread, either intra-domain or inter-domain, by passing the *instruction pointer* (IP) and the *stack pointer* (SP) for the new thread to the system. In the case of inter-domain thread creation, extra parameters are also required, as described in section 3.5.2. It is the caller's responsibility to make sure the new thread will be able to access the code pointed to by the IP. In the case of intra-domain thread creation, the new thread must also be able to access the object pointed to by the SP (ie. the SP must point to an object that is part of the caller's APD). The new thread (whether intra-domain or inter-domain) is then added to the process table as a child of the caller.

Since the APDs are the smallest unit of protection in the model, they should be represented as L4 tasks. L4 tasks are partioned from one another so that threads operating in one task cannot access data in another. This partitioning is directly applicable to Mungi APDs.

3.5.1 Intra-domain thread creation

When a thread creates a new thread in its own active protection domain, the term intra-domain thread creation is used. Because APDs are directly mapped to L4 tasks in the revised model, this operation reduces to creating a new thread in the same L4 task as the caller. Hence, intra-domain thread creation is very light-weight because it only involves adding the new thread to the process table and having the caller's system call stub create the new thread in its own task. This is an extremely efficient operation in L4. This type of thread creation is basically identical to the `NewThread()` system call in the current version of Mungi (both in implementation and conception). It allows a thread to very efficiently create concurrent threads of execution operating in the same domain.

When a thread creates a new intra-domain thread, it must provide a pointer to an object the new thread can use as a stack. Instead of passing parameters intended for the new thread to the system call, these can be pushed directly onto the stack. If the new thread needs to access the caller's environment, a pointer to this can also be pushed onto the stack. Having the caller push only what is required onto the stack reduces the amount of time required to create a new intra-domain thread.

3.5.2 Inter-domain thread creation

Obviously, threads need to be able to create new threads in protection domains other than their own. This is achieved through an inter-domain thread creation system call.

Inter-domain thread creation is significantly more complicated (and costly) than intra-domain thread creation. This is because a new APD needs to be constructed, and consequently a new L4 task must be initiated.

Inter-domain thread creation accepts a stack pointer as does intra-domain thread creation, but it also accepts a pointer to a PD object to use as the template for the new thread's APD. Additionally, it accepts the parameters that should be passed to the new thread, and a pointer to an environment of which the new thread will receive a copy.

The following procedure occurs when a new inter-domain thread is created:

1. The given pointer to a PD object is validated to make sure the user is allowed access, and that it is in fact a PD object.

2. A new APD is constructed by copying the capabilities in the given PD object into the APD. Each capability that is copied across is first validated to be sure it has not been revoked since it was inserted into the PD object.

3. The object pointed to by the SP is validated.

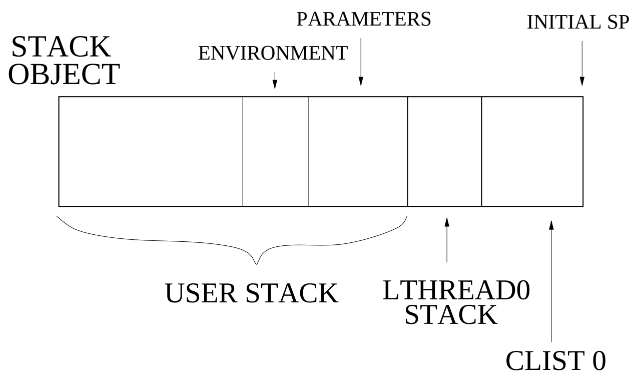

Figure 3.5: Stack object for an inter-domain thread creation.

4. A new clist is created at the top of the given stack object. This clist contains the capability for the stack object itself and is inserted into the new APD. The clist is placed at the top of the stack so that it cannot be easily (accidentally) overwritten by the user.

5. Every L4 task has a special thread associated with it. This thread, *lthread 0*, handles all of the exceptions that other threads in that L4 task raise. Since an inter-domain thread creation always results in a new L4 task being created (representing the new APD), lthread 0 must also be created at this time. The stack for lthread 0 is allocated just below the clist and is reasonably small.

6. The user stack is designated as the remainder of the given stack object. The parameters passed to the new thread are pushed onto the user stack, as is a copy of the given environment. Hence, the environment is really just another parameter to the new thread as with UNIX.

7. The thread is added to the process table (as is the new L4 task).

8. The new L4 task is started, running both lthread 0 and the new user thread (executing the specified code).

In the existing model, a new task can have its APD specified in two ways. Firstly, it may receive a copy of its parent's APD. Secondly, its parent may specifically pass a set of clists to the system from which the new APD will be constructed.

The latter method is the only way of creating a Mungi task in a different domain to its parent. This type of APD specification is not really necessary in the revised model as it is semantically equivalent to creating a new PD object and using this as the parameter for an inter-domain thread creation. The only possible reason for including the other case is to remove the need for creating (and later deleting) a PD object, which would slightly reduce the cost of starting some new threads. However, Mungi has extremely fast object creation and deletion speeds, so this argument is not particularly strong. This type of inter-domain thread creation has now been excluded, which serves to simplify the model.

The option to copy the caller's APD is not required in the revised model either. In the existing model, this option allows a task to lock its APD and use this as as the basis for a new confined task. This can now be accomplished by creating a PD object with the appropriate clists, locking it and using this as the parameter for the inter-domain thread creation. Since the other method is not necessary and complicates the model fractionally, it has also been removed from the revised model. Hence, the only way of creating a new inter-domain thread in the revised model is by specifying a PD object to which the caller has access.

Creating an inter-domain thread is in many ways the analogue of creating a Mungi task. There are however some important differences in the revised model:

- The kernel does not provide the stack for an inter-domain thread creation as it does for a new Mungi task in the existing version. This makes inter-domain thread creation more consistent with intra-domain thread creation, and reduces the likelihood of unforseen security issues — providing less avenues of achieving similar goals for the user makes it easier to enforce rigorous protection policies.

- Inter-domain threads become children of the calling thread, not children of tasks, obviously.

- There is just one mechanism for creating inter-domain threads whereas there are two for creating Mungi tasks. Again, limiting unnecessary options reduces complexity and makes security easier to enforce and reason about.

Given that the two types of thread creation are now fairly consistent, it is conceivable that they could be merged into a single call. This has not been done for several reasons.

Firstly, intra-domain thread creation allows the caller to push the parameters to the thread onto the stack before creating the thread. This is not possible in the case of inter-domain thread creation because they would be overwritten with system data such as the slot 0 clist and the lthread 0 stack. Hence, parameters to a thread created inter-domain must be passed to the kernel so that it can copy them onto the user stack at the appropriate place. A solution to this inconsistency between thread creations would be to require intra-domain threads to also have their parameters passed to the kernel. However this may slightly increase the time taken to start an intra-domain thread and may constrain any conventions higher software layers may wish to follow. Also, it is unnecessary and should therefore be excluded.

Secondly, inter-domain threads require a PD object to be passed as well as the pointer to an environment object, neither of which the intra-domain threads require.

3.5.3 The confinement problem

> Confinement is defined as allowing a borrowed program to have access to data, while ensuring that the program cannot release information. A program that cannot retain or leak information is said to be memoryless or confined. ... the problem is to eliminate every means by which an authorised subject can release any information contained in the object it has access to, to some subjects which are not authorised to access that information. [Gos91]

Confinement is an important issue in most operating systems, and Mungi is no exception. Threads must not be able to break confinement as the system would then be susceptible to Trojan Horse attacks ([Gos91]).

In the existing version of Mungi the problem of confinement has been addressed through the implementation of APDs. APDs contain clists which are searched when a thread faults on an object access. However, the capabilities for these clists do not need to be held in the clists themselves. This means that a thread can make use of clists without being able to directly read or modify them. This, when used with an APD which does not include any writable objects that are readable by other tasks means the task cannot directly leak information, or add objects to the clists referenced by its APD which would allow it to do so. Though necessary for confinement, this condition is not sufficient, as a task could insert an extra clist into its APD that does include such objects. Mungi protects the system from this by introducing locked APDs. If an APD is locked, the task is unable to insert extra clists into it and is thus unable to directly open any channels of communication to other tasks.

In general, the way a task would create a new confined task is as follows. It would first modify its own APD to include whatever rights were necessary for the untrusted program to operate (making sure not to include any `WRITE` access to objects readable by other tasks). It would then lock its APD and create a new task, which inherits its APD, running the untrusted code. In this way, the new task should not be able to betray any information it accessed, and the task would seem to be completely confined.

In the original version of Mungi, this was still insufficient to prevent the leaking of information however. There were two ways of creating a new task in this version of Mungi:

1. The child inherited a copy of the parent's APD.

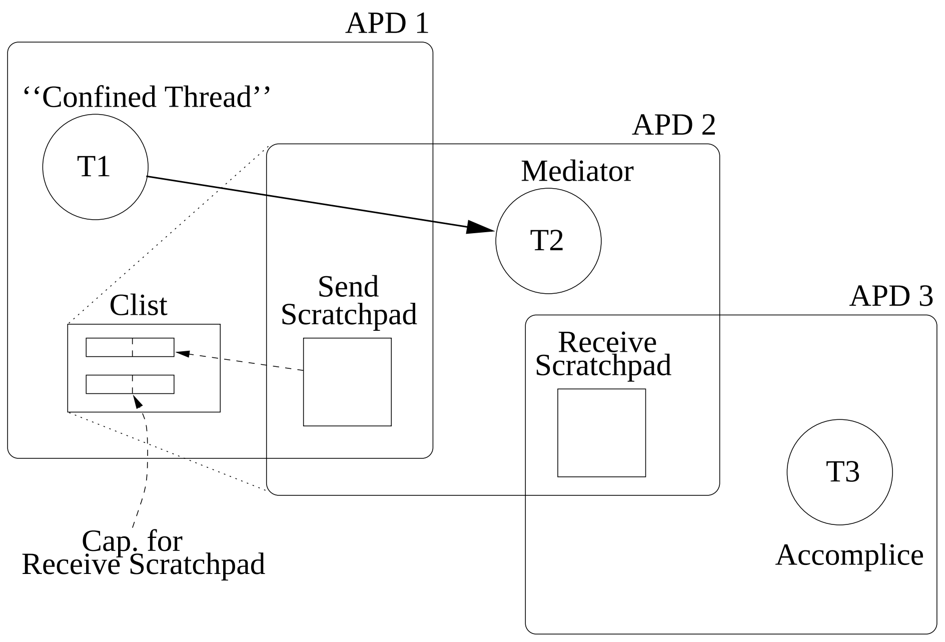

Figure 3.6: Possible confinement breach.

2. The parent supplied a set of capabilities for clists from which the kernel constructed a new APD for the child.

If a similar mechanism to the latter is allowed in the revised model (whereby a thread can create a new inter-domain thread by specifying the capabilities for some clists), a "confined thread" would be able to leak information to an accomplice via a mutual mediator in the following way:

1. The *Confined Thread* (T1) must have compiled into its code a `WRITE` capability for some object. This object, let us call it the *Receive Scratchpad*, would be a part of another protection domain in which an *Accomplice* thread (T3) would be running. The Accomplice thread would have `READ` access to the Receive Scratchpad. The Accomplice would have created the Receive Scratchpad in the first place and supplied the source code for the Confined Thread (including the hidden capability for the Scratchpad).

2. The Confined Thread would create a new *Clist* into which it would place the secreted capability.

3. The Confined Thread would then create a new object, let us call it the *Send Scratchpad*, and add a `READ` capability for it to the Clist.

4. Then, by providing a capability for the Clist to the thread creation system call, the Confined Thread would start a child thread — the *Mediator* (T2)

— running in an APD defined by the Clist. The Mediator would thus be able to read from the Send Scratchpad and write to the Receive Scratchpad.

5. To leak information, the Confined Thread would simply write whatever data it wished to leak into the Send Scratchpad. The Mediator would then copy this to the Receive Scratchpad, and the Accomplice would be able to read it.

For this reason, the existing implementation of Mungi does not allow explicit presentation of capabilities to clists when creating a new task. Instead it accepts a pointer to the clist and this is then validated in the caller's APD. This means that for the system call to succeed, the clist would need to be a part of its APD. This is impossible if the task was specifically confined as outlined above, since it could not modify its APD to include it. Hence, confinement is assured.

The mechanism of providing clists to an inter-domain thread creation has been removed from the revised model, but a similar type of confinement breach would still be possible if the explicit presentation of capabilities when creating a new inter-domain thread was allowed. Under the revised model, the only way in which an inter-domain thread can be created is by specifying a PD object to use. Hence, confinement could be breached in the following way:

1. The Confined Thread has a `WRITE` capability for the Receive Scratchpad compiled into its code as before.

2. The Confined Thread then creates a new Clist and Send Scratchpad as before, and inserts the capability for both the Send and Receive Scratchpads into the Clist.

3. Then, using the `PDCreate()` system call, the Confined Thread constructs a new PD object from the Clist.

4. The Confined Thread is then able to create the Mediator thread by passing a capability for the new PD object to the system. The Confined Thread would thus be able to pass information to its Accomplice thread as before via the Mediator thread.

This situation is really just a variation of the previous case. By comparing the similarities, it can be seen that the problem is really caused by allowing a thread to create a new inter-domain thread by passing explicit capabilities to the thread creation system call. If it passed only pointers to clists, then it would be unable to create threads in new protection domains. Hence, to ensure confinement, what is really required is to prevent any thread from calling the `ThreadCreateInter()` system call with explicit capabilities to a PD object. This is easy to implement —

when the system call is made, the kernel simply validates the address passed to be sure the referenced object is actually part of the caller's APD.

To confine an untrusted thread in the revised model, the following procedure would be followed :

1. The caller would create a new PD object and insert only the clists required by the untrusted thread to do its job. Obviously, there would be no objects referenced by the PD object which the untrusted thread could write to and other threads could read. Also, it would not include the capabilities to modify the clist objects themselves.

2. The caller would then lock the PD object and create an inter-domain thread using the PD object.

3. The untrusted thread would be unable to insert clists into its own APD since it would inherit the locked state of the PD object. It would not be able to start a new inter-domain thread either as it would need to have the capability for a PD object in its APD. Creating new PD objects would not help either as it would be unable to add these to its APD. The thread would thus be confined to the domain the caller specified.

3.5.4 PDX calls

With the PD objects come several previously infeasible features. One of these is the ability to create new threads as described above. Another is the ability to call a PDX function using a valid PD object as a parameter, instead of Mungi assuming the caller's APD. This is already possible in some sense — threads can pass specific clists to a PDX function and it will use these as the basis of the new APD, but since the system now specifically maintains groups of clists (PD objects), it would be simpler and more consistent to pass one of these instead. This idea is something that should be incorporated into Mungi as is clear from the following quote from the Mungi API ([HVER97]):

> ...the caller has the option of explicitly supplying a protection domain when calling the PDX procedure. ... This gives the caller the maximum control over which objects the PDX procedure can access.

As the facility already exists in Mungi to define a protection domain different from the caller's when executing a PDX procedure, this should be modified to use PD objects. To keep conventions consistent with thread creation, and because it is no longer necessary, the method of passing clists as parameters to the PDX system call has been removed.

Aside

It is interesting to note the similarity between PDX calls and thread creation calls. Both (potentially) create a new APD and start a new thread running within. There are however several differences:

- Thread creation can be intra-domain whereas PDX calls always change APDs. However, inter-domain thread creation also always changes APDs.
- PDX calls terminate with an *upcall* to the calling thread whereas thread creation terminates with a signal to the terminating thread.
- The major difference is that PDX calls rely on defined entry points to code. Thread creation on the other hand allows the caller to specify the exact starting point of execution. This difference however is relatively minor from an implementation stand point.

It seems possible that PDX calls and thread creation (if not intra-domain then at least inter-domain) could eventually be merged into a unified concept.

Chapter 4

Implementation in L4

The implementation of the revised model in L4 is reasonably similar to the existing model and much of the infrastructure already implemented was re-used to some extent. The code that really required an overhaul was that relating to the process table (which was mostly rewritten) and the client system call stubs and server functions for all protection and execution related system calls. Various other pieces of code were modified to fully integrate the system.

Most of the new code was written so as to smoothly merge with the existing code. The previously existing code however does not directly correspond to the latest Mungi API (with regards to system call calling conventions and so on). For this reason, the actual implementation of some features is slightly different to that specified in this document. As far as possible, the *functionality* of the code is equivalent. When the Mungi source code is reworked, these anomalies can be rectified in a manner that is consistent with the API. A list of all anomalies between this document and the implemented code can be found in section 4.6.

4.1 The process table

The reason for restructuring protection domains and threads is to provide a conceptually simpler and more flexible execution model. Since threads are now the absolute unit of execution (there are no "conglomerate" Mungi tasks), the user should be able to operate on and with these threads in a logically obvious and consistent way. It is not desirable to have the model arbitrarily constrained by implementation issues such as the fact that it is actually implemented on top of L4 tasks. In the same way that Mungi's single address space is implemented on top of multitudes of discrete address spaces in a manner that reveals none of this to the user, so too should the execution model be completely unaffected by the fact that it is really using L4 tasks to provide protection domains.

To this end, the process table needs to be carefully designed, as this is the entity which really mediates the semantic and physical structures of the revised execution and protection model. The following features are desired:

- Threads should be able to create new threads using any PD object to which they have access.
- New threads should be placed beneath their parents in the thread hierarchy.
- Threads should be able to delete their descendants (with or without adoption of subsequent children), regardless of the APDs in which they or their victims reside.

The process table requires some fairly major reworking as it is currently designed to handle Mungi tasks, and even then only in a reasonably limited way — for instance, there is no adoption of children. Since there are no Mungi tasks any more, the task control block is no longer needed in its current capacity. However, it is still required to maintain records regarding the L4 tasks that are currently running (and the APDs they represent).

In the existing model, the Mungi tasks are arranged hierarchically - ie. a task has a single parent, and possibly many children. The revised model however, requires a thread hierarchy to be maintained instead. At present, the task control block holds child and parent information. This information needs to move from the task control block to the thread control block. Each thread also needs know to which L4 task it belongs.

The task control block needs to be aware of all threads running in the L4 task it represents. This is implemented through a linked list structure. Thus, the threads in Mungi are chained together in two ways. They are linked via the thread hierarchy which crosses L4 tasks, and they are linked to the other threads within their own L4 task. When a thread is deleted, the linked list in its task control block is modified to reflect its deletion. If this list ever becomes empty, then the process table knows the L4 task is empty and is able to delete it.

At present, the process table makes assumptions about threads when they are added. One of the necessary changes to the `AddThread()` procedure is related to the fact that Mungi assumes all new threads are part of the caller's task — it sets the new thread's task by examining the task of the caller. This is unsatisfactory in the new model, as threads may now be created in different L4 tasks (corresponding to different APDs). As the new model also requires storing the calling thread as the parent of the new thread, `AddThread()` now needs two parameters where once there was one. One is the calling thread, and the other is the L4 task in which the new thread should be created. In the case of intra-domain thread creation, these can be the same, but when a thread is created across a protection domain, the two will obviously differ.

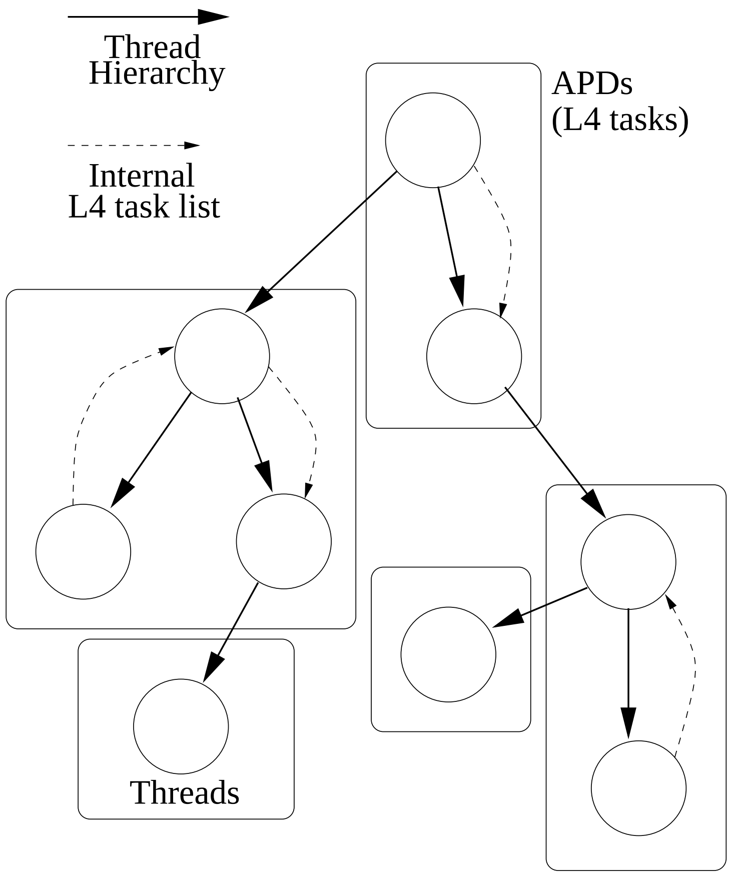

Figure 4.1: Threads are linked in two ways - the thread hierarchy and an internal L4 task list

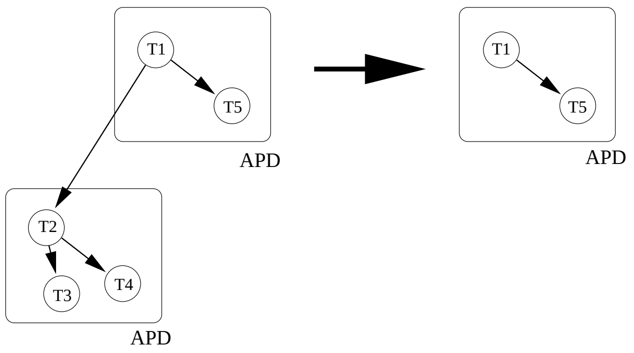

Figure 4.2: Thread deletion - T1 deletes T2 followed by cascading termination and destruction of APD.

The real changes to the process table arise with thread deletion. At present, threads can be deleted by simply removing them from the process table and checking whether they are the last one in a particular Mungi task. If they are, then their task is also deleted (which leads to cascading termination). In the revised model, a more fine-grained solution is needed since deleting a thread will more frequently result in recursively deleting other threads.

There are two ways of deleting a thread:

Without adoption When a thread is deleted, all of its child threads are also deleted. This is achieved through a recursive depth-first function. For each of the threads deleted, their task is checked to see if they are the last thread and if so, the L4 task is also deleted. After each thread is deleted, all threads waiting for it to die are woken.

With adoption When a thread is deleted, its child threads are appended to the child array of the caller. If the deleted thread is the last in its task, the task is also deleted. Supporting adoption is just a matter of shuffling some pointers around to keep the thread hierarchy consistent, and preventing the thread deletion from being recursively called.

When an L4 task is deleted, the memory allocated for the APD associated with that L4 task is reclaimed.

Here is the pseudo-code for thread deletion (including optional adoption):

```
KillThread(victim,caller,adopt)
begin
   if (victim is descendant of caller)
      if (adopt)
         for (each child in victim.childArray[]) do
            caller.childArray[] += child
```

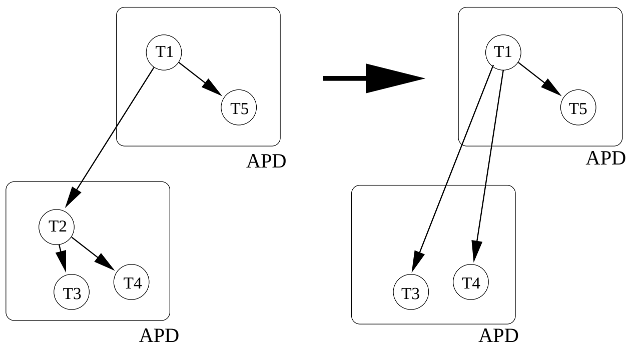

Figure 4.3: Thread deletion with adoption - T1 deletes T2 adopting T3 and T4.

```
            child.parent = caller
            victim.childArray[] -= child
         od
      fi
      Kill(victim)
   fi
end

Kill(victim)
begin
   for (each child in victim.childArray[]) do
      Kill(child)
   od
   victim.parent.childArray[] -= victim
   victim.L4task -= victim
   if (victim.L4task is empty)
      delete victim.L4task
   fi
   WakeAllThreadsWaitingFor(victim)
end
```

As Mungi tasks no longer exist, the process table no longer requires an explicit `KillTask()` function. All L4 tasks that are created are automatically destroyed when no threads are left running in them.

4.2 Kill-lists and vcaches

In the existing model of Mungi, each Mungi task maintains a *kill-list*. This is basically a list of objects which threads in that task have created and not marked persistent. When a task is deleted (either by an explicit `TaskDelete()` system call or by all of its threads terminating) all objects in the kill-list are also deleted. This is Mungi's approach to garbage collection and prevents objects from being left around after they are not needed by any threads. Although kill-lists are not actually implemented in the existing version of Mungi, the basic structure is evident in the source code. Without a task however another way of garbage collecting the transient objects created by threads must be found.

One obvious solution is to simply move the kill-list from the Mungi task to the Mungi thread. In this solution, each thread maintains its own kill-list and objects are deleted when the thread dies. Though simple and conceptually appealing, this approach places too much emphasis on the thread. Threads are supposed to be inherently light-weight and attaching a kill-list to each of them (which must be created when a new thread is created and serviced whenever a thread is deleted) seems excessive. This is especially true of intra-domain thread creation which would generally be used to perform parallel operations on the same objects.

One reason to associate kill-lists with threads is to do with migration between APDs. In this case, each thread would need to take its kill-list with it. However, the advantages of allowing such migration are dubious to say the least. If really desired, it could probably be better implemented through system libraries which recorded the state of a thread, deleted it and created a new thread in a different domain — kernel support for such an operation is not really required. Of course, threads can still 'change APDs' by calling PDX procedures.

The only other viable location for the kill-list is to have each APD maintain its own. The kill-list would be created when the APD was created, and the objects it references would be deleted when the APD terminated (ie. when all threads using it terminated). This approach seems reasonable for several reasons:

1. It removes the burden of garbage collection from the thread which means (most) thread deletion remains speedy. Obviously the deletion of the last thread in an APD would be more expensive as this is when the kill-list would be serviced. Intra-domain thread creation remains rapid as the system does not have to set up a kill-list in this case. A kill-list is however set up for every inter-domain thread creation.

2. The group of threads sharing an APD would presumably be sharing objects anyway, and most of the objects created by threads would be used by other threads in that domain regardless of whether the thread that created the object existed or not.

3. There are no other abstractions to which the kill-list can be attached, apart from PD objects which seems completely unreasonable as these are inert entities and should not participate in the actual execution of threads.

As APDs map directly to L4 tasks, the kill-list can be associated with the appropriate task control block maintained by the process table. This is in essence the way kill-lists are currently hooked into Mungi so implementing them will not require any radical reworking of source code. Kill-lists have not however actually been implemented so far.

It is also necessary to determine the location of the *validation cache* (vcache) which is at present stored with each Mungi task. The vcache stores validation information from protection faults, thus reducing the amount of time required to validate further pages of an object on subsequent protection faults. As it is a cache of validation information, it clearly relates directly to the APD. It is appropriate, at present, to store it with the Mungi task as these each have their own APD. In the new model, the vcache can be related to the L4 task representing an APD, in the same way as the kill-list. Again, this is really no different from the existing implementation — it is only a semantic shift.

4.3 System call modifications

The number, type and scope of the system calls has changed somewhat from the existing version of Mungi.

Since PD objects are now a part of the system, there needs to be some mechanisms to operate on them. Specifically, this is achieved with system calls. There exists in the current version of Mungi, a set of system calls relating to APDs and these can be extended to include PD objects also. Providing the address of a PD object to one of these calls will operate on that PD object (providing the caller has access to do so), and providing a `NULL` pointer will operate on the caller's APD.

The APD system calls have almost all been renamed as generic PD system calls. The only one that remains specifically as an APD system call is `APDFlush()`. This call flushes the validation cache of a Mungi task and revalidates all clists in the task's APD. This makes no sense in the context of PD objects, but is still required by APDs because these are associated with cached validations.

Two new PD system calls have also been added. These allow creation and deletion of PD objects. It is arguable whether the latter is redundant or not. After all, a PD object is an (almost) ordinary object — it seems plausible to use the `ObjDelete()` system call to delete PD objects. The problem with this is that a PD object is system-maintained, whereas `ObjDelete()` is designed to operate on user-maintained objects. When a user creates a PD object, they receive an `EXECUTE` capability in return. This does not confer ordinary object destruction rights necessary for calling `ObjDelete()` on the PD object. A special case could be included in the `ObjDelete()` system call (ie. check if the given object is a PD object and if the user has an `EXECUTE` capability for that object), but this would slightly lengthen the critical path taken to delete an ordinary object, and would serve no useful purpose. Besides, it seems natural to have a `PDDelete()` system call to match the `PDCreate()` system call.

Unfortunately, there is already an `ApdDelete()` system call which removes a clist from an APD. For reasons of clarity, the existing `ApdDelete()` call has been renamed `PDRemove()`, and the PD object deletion call will become `PDDelete()`. This is so as to make the names of the system calls for generic object and PD object deletion consistent.

The thread and task system calls have been varied a fair bit, mainly because there is now no concept of a task in Mungi. There are two system calls for creating new threads. One handles intra-domain thread creation and the other deals with inter-domain thread creation. It would be possible to merge the two thread creation system calls into a single call, but there would be redundant parameters involved depending on which type of thread was required. As the two calls are semantically fairly different and there is extra overhead in the implementation for inter-domain thread creation, it seems right to have two system calls instead of one.

`ThreadCreate()` remains as the call to create a new thread in the same active protection domain as the caller. `TaskCreate()` has now become `ThreadCreateInter()`, but retains most of the parameters of its previous incarnation (lacking only the clists used for implicit APD creation), and gains a parameter allowing a pointer to a PD object to be passed. The `stack_size` parameter has become a pointer to a pre-allocated stack object. `ThreadDelete()` gains an extra parameter also, `adopt`, which specifies whether the children of the deleted thread should be adopted by the caller.

`TaskDelete()`, `TaskWait()`, `TaskMyId()` and `TaskInfo()` have all been removed as they no longer fit into the model. All other thread related system calls have been left untouched.

The `PdxCall()` system call has been modified to accept an explicit PD object as opposed to a set of clists passed as parameters.

The code for the server side does not need to be significantly modified. At present, it assumes the parameters passed are the clists to use. Since a PD object is really just an array of clist capabilities, the system simply needs to set a pointer to those in the PD object and then validate them one by one. This can be achieved by patching in a small block of code at the beginning of the `PdxCall()` server-side code.

There are potential race conditions with the above procedure. If a thread is inserting a new clist into the PD object while another is using it to perform a PDX call, synchronisation problems could occur. For this reason, any invalid capabilities encountered while a PDX call is accessing a PD object are ignored.

If a `NULL` capability is passed as the PD object, then the caller's APD will be extended instead. To pass an empty protection domain, simply pass an empty PD object to the PDX system call. The call will then operate in just the domain specified for that PDX procedure.

The overall number of system calls is now three less than in the previous version of Mungi, and this could conceivably be reduced further by integrating `ObjDelete()` with `PDDelete()`, and both of the thread creation calls with the PDX call (though it is doubtful what advantages this would have).

4.3.1 Protection domain system calls

PDCreate Creates a new PD object and returns an `EXECUTE` capability made up of the object's address and the supplied password. The specified clists are validated and inserted into the PD object. Invalid clists are ignored.

PDDelete Deletes the specified PD object providing the caller has an `EXECUTE` capability for it in their APD.

PDInsert Inserts the specified clist into the specified position of the given PD object. If no PD object is specified, the caller's APD is assumed.

PDRemove Removes a clist from the specified PD object. If no PD object is supplied, the caller's APD is used instead.

PDGet Returns the list of clist and capability handler pointers in the APD, as well as the locking status of each slot. Unassigned clist or handler slots are returned as zero. For clist capabilities, only the address part is returned; passwords are not.

PDLookup Performs an explicit validation of an access of type *mode* to *address*. If successful, returns the first capability granting at least those rights, and caches the validation. If it is locked, no passwords are returned. If no PD object is specified, assumes caller's APD.

PDLock Locks a specific slot or an entire PD object. If none is supplied, the caller's APD is assumed instead.

ApdFlush Flushes the APDs validation cache and revalidates all clist capabilities in the caller's APD.

PdxCall Calls a PDX object via specified entry point. If a PD object is passed, this will be extended with the domain registered for the PDX call. If no PD object is specified then the caller's APD is assumed. To pass an empty domain, pass an empty PD object. Otherwise operates like the previous version of PDX.

4.3.2 Thread system calls

ThreadCreate Creates a new thread in the caller's APD. The new thread begins execution at the instruction pointer and uses the specified object for its stack. Parameters for the new thread should be pushed onto the stack object prior to making the system call.

ThreadCreateInter Creates a new thread in a new APD. The new APD is constructed from the referenced PD object providing the caller has that object in their APD. Supplying no PD object will fail. Uses a pre-allocated stack object, but accepts an environment pointer and a list of arguments to pass to the new thread. Parameters are pushed onto the stack by the kernel.

ThreadDelete Deletes the specified thread providing it is a descendant of the caller, passing the supplied signal to all threads waiting for the victim. If `adopt` is set will add victim's children to caller's set of children, otherwise all descendant threads will be recursively deleted.

4.4 Miscellaneous modifications

An extra flag has been added to the object descriptor. This is the `O_PD` flag and represents whether or not an object is a PD object.

In addition to the original `Validate()` function (found in the `Security.c` file) which takes a faulting address and checks whether a given thread's APD allows access to the object residing there, another validate function was written. This function validates whether a thread has access to a particular PD object, and then verifies that it is in fact a PD object. This is easily achieved by calling `Validate()` with an `EXECUTE` access mode on the supplied object, and then checking the `O_PD` flag in the returned object descriptor. This `ValidPD()` function simplifies the code in the `mungi.c` file.

Many of the `#defines` for system call "magic numbers" were modified to reflect the changed system calls. The actual names of server side and client system call stubs were also modified for this reason.

4.5 Issues with the alternative APD design

Depending on the model of APD chosen, different implementation issues arise. Previous sections have dealt with implementing the preferred method of copying a PD object into an APD structure on thread creation. This section deals with the alternative method of implementing the APD as a two-part entity — a pointer to a PD object and an extra slot in the thread control block containing the capability for the clist referencing the thread's stack object (c.f. section 3.4.4). This method proves slightly more complicated to implement as it is introduces distribution problems and extra object access validation complexity.

First, and most obviously, this implementation means that a single PD object may be in use by many threads around the network simultaneously. This means that when a thread changes the PD object, all threads using it will have their protection domain affected. Though this may or may not be a problem semantically, it could result in slight inconsistencies as changes to the object are propagated between nodes. This would probably not prove to be a problem however. More interesting is the problem of when the PD object is deleted.

4.5.1 PD object deletion

If the PD object is simply a template for creating APDs, then deleting a PD object is not a problem. As it is entirely independent of the thread once it has been created, the system does not have to worry about maintaining consistency between the executing threads and the PD objects — once the thread has its APD, it does not matter whether the PD object exists or not.

If however a PD object is actively associated with a thread via a pointer from the thread control block, then a problem is more apparent. Since the PD object could be in use by many threads on many nodes, there needs to be some mechanism for cleaning up when a PD object is deleted. Obviously, the thread deleting the PD object calls the kernel on which it resides, so Mungi is able to easily destroy the appropriate L4 tasks and threads on that node which rely on the PD object. Threads using the PD object that reside on remote nodes however, need to be informed of the destruction. This can be achieved in two ways:

1. Broadcast a signal to the other kernels telling them explicitly that the PD object no longer exists. When a kernel receives this signal, it can examine whether it is running any threads which make use of the PD object and if so, delete them.

2. Alternatively, the PD object can be entered into some kind of globally accessible stale list (residing at a well known address) which the remote kernels may examine periodically. This lazy option is preferred because it would mean less strain on the system when a PD object was deleted. This method however requires the guarantee that the kernel will check the stale-list and act accordingly within some maximum length of time. An extra method that could be employed would be to delete any thread that requires a validation on a non-existent PD object. The kernel could also delete the other threads

in the same L4 task as these too would be making use of the PD object. Cached validations will continue to work for some time but the kernel already guarantees that the caches are flushed within some maximum period. If during one of these flushes the kernel discovers the PD object no longer exists, all threads using it could be deleted.

Process table addendum

The process table in the existing version of Mungi supplies a `TaskDelete()` function. This allows Mungi tasks to be deleted from the process table and initiates a cascading termination of children tasks. In the revised model, there is no longer a specific task deletion operation. L4 tasks are implicitly deleted by deleting the threads that make them up — when the last thread is deleted, the L4 task is deleted also.

With the alternative APD implementation, the process table may again require a task deletion function. If the PD object is dynamically accessed by the kernel, then all L4 tasks associated with it must be deleted if and when the PD object is deleted. This can easily be achieved by reintroducing the `TaskDelete()` process table function and have the kernel call this when a PD object associated with a particular L4 task is deleted. This cleanup code could be a part of the `PDDelete()` system call.

4.5.2 Object access validation

At present, everything the kernel needs to know about a thread's protection domain resides in a kernel data structure — the APD. The APD is simply searched from top to bottom whenever a protection fault occurs (ie. whenever a thread tries to access an object). To save time on subsequent validations, these validations are cached in the validation cache associated with the task. The basic existing algorithm for validation is as follows:

```
if (object in vcache)
   succeed
else
   if (object in object table)
      for (each clist in APD) do
         for (each cap in clist) do
            if (cap is valid for this object)
               add cap to vcache
               succeed
            fi
         od
      od
   fi
fi
raise prot_fault
```

With the preferred new APD model, object validation is unaffected since the APDs are used in precisely the same way as in the existing model. If however, the alternative APD model were implemented, an extended object validation procedure would need to be applied:

```
if (object in vcache)
   succeed
else
   if (object in object table)
      for (each cap in clist 0) do
         if (cap is valid for this object)
            add cap to vcache
            succeed
         fi
      od
      for (each clist in PD object) do
         for (each cap in clist) do
            if (cap is valid for this object)
               add cap to vcache
               succeed
            fi
         od
      od
   fi
fi
raise prot_fault
```

There are potential race conditions with the above validation procedure. If a thread is inserting a new clist into the PD object while another is using it to validate access to an object, synchronisation problems could occur. For this reason, any invalid capabilities encountered while searching the APD should be ignored as they are in the present implementation ([HVER97]).

4.6 Implementation anomalies and known bugs

Though the revised model of protection and execution has been designed in theory, not all of the features are currently implemented in the new code. A list and description of the anomalies (and bugs) follow, in order of importance:

1. Inter-domain thread creation is not implemented exactly as described above. Instead of the user providing a stack pointer for the new thread, the system creates a new stack object of a specified size. It also creates a separate object for the clist containing the stack object. This code remains from the existing version of Mungi. It should not be a major problem to change this to the revised design, as the server-side code will in fact be simplified. Once this code is modified, creating an inter-domain thread should be faster than it is currently, as there will be two fewer object allocations.

2. Deleting a PD object does not work very well at present. This is because the code is mostly the same as that for deleting an ordinary object, and this code is incomplete. Specifically, the caller's vcache is completely destroyed instead of merely being cleared of just the deleted object. When the code for deleting an ordinary object is implemented correctly, it should also be propogated to the PD object deletion routine.

3. There is no real error checking on most of the new and modified system calls. Passing invalid parameters will frequently result in the system hanging, crashing or intentionally bailing out.

4. The system does not recover from exceptions very well, although this seems to have always been a problem. After an exception, the system is very unstable and may hang at any time.

5. All system calls relating to PD objects (and APDs) accept capabilities for PD objects as a parameter. This is so as to remain consistent with the existing system calls which also accept capabilities when dealing with objects. Future versions of the code should accept only pointers to PD objects for reasons of confinment (c.f. section 3.5.3). Although an entire capability is passed to the client stub, only the address part is actually sent to the server, so modifying this code to accept addresses instead of capabilities will be simple.

6. `PDLookup()` fails if the given address to look up is within a PD object. The cause for this bug is unknown. It appears to try to run a capability handler (which crashes), so it is probably just a matter of initialising some clist entries, for instance.

System Calls:		
PDCreate	(password, n_pd, ...) *raises* no_memory	$\rightarrow$ (execute_cap)
PDDelete	(pd_addr) *raises* prot_fault, invalid_pd	
PDInsert	(pd_addr, pos, clist_addr, hndlr_addr) *raises* prot_fault, inv_pos, overflow, apd_locked, invalid_pd	
PDRemove	(pd_addr, pos) *raises* inv_pos, invalid_pd	
PDGet	(pd_addr) *raises* invalid_pd	$\rightarrow$ ({cap, handler, locked}[n.slots])
PDLookup	(pd_addr, addr, mode) *raises* invalid_pd	$\rightarrow$ (addr)
PDLock	(pd_addr, pos) *raises* invalid_pd	
ApdFlush	()	
PdxCall	(addr, param, pd_addr, nargs, ...) *raises* prot_fault, invalid_entry, apd_locked, invalid_pd	$\rightarrow$ (any)

Table 4.1: Modified system calls relating to protection domains.

System Calls:		
ThreadCreate	(start_addr, stack) *raises* prot_fault, no_threads	→ (thread_id)
ThreadCreateInter	(start_addr, stack, env, pd_addr, nargs, ...) *raises* prot_fault, no_threads, invalid_pd	→ (thread_id)
ThreadDelete	(thread_id, status, adopt) *raises* invalid_thread	

Table 4.2: Modified system calls relating to threads.

Chapter 5

Performance Results

The guiding constraint of this thesis was that no change to the system should make execution significantly less efficient than the existing model. To test whether this was the case, a series of micro-benchmarks was arranged and run under both the existing and revised models. These micro-benchmarks were essentially the same as those published previously by the DiSy group[1] which compared the times of thread and task creation, object access and other operations with other operating systems including Linux, IRIX and Opal[2].

The original micro-benchmarks tested (among others) the following operations in Mungi:

1. Thread creation
2. Thread deletion
3. Task creation
4. Task deletion

Obviously now that tasks are not part of the Mungi execution model, some of the tests need to be modified. Roughly equivalent operations are (respectively):

1. Intra-domain thread creation
2. Thread deletion (of a thread at the bottom of the thread hierarchy)
3. Inter-domain thread creation (using an existing PD object)
4. Thread deletion (of the last thread in an APD)

[1]The Distributed Systems group at UNSW.
[2]The original results can be found in Table 3.1.

The revised model also requires a few more tests. These are the system calls for creating and deleting PD objects, and other operations which PD objects affect:

1. PD object creation
2. PD object deletion
3. Creating a new PD object and starting a new inter-domain thread with it
4. PDX call using a PD object

Operation	*Existing Mungi*	*Revised Mungi*
Null system call	4.4	5.3
PDX call (new APD of 9 clists)	45.4	66.9
PDX call (new APD of 0 clists)	42.0	55.6
PDX call (extend caller's APD)	42.1	45.0
(Intra-domain) thread create	26.1	31.6
(Intra-domain) thread delete	24.0	27.4
Task/inter-domain thread create	317.1	477.9 (?)
Task/inter-domain thread delete	127.1	147.9
Object create	33.7	36.1
PD object creation	-	99.6
PD object deletion	-	53.2 (?)

Table 5.1: Micro-benchmark timings on custom MIPS R4700 hardware (in μs).

The results show a clear bias in favour of the existing model of Mungi. Each operation is noticeably faster in the existing model. There is also an extreme difference between the results in Table 5.1 and those published previously for the same version of Mungi. This is presumably due to the fact that the previous results were recorded on a different architecture. Hence, results in this paper cannot be directly compared to previously published results. However, the benchmarks in Table 5.1 were both run on the same machine so they are open to comparison.

It is interesting to compare the null system call and object creation system calls. In each case, the existing model outperforms the revised model by 10%-20%, *even though the revised model did not alter the code for these operations in any way*. This anomaly can be explained in two ways: either the revised code has some internal conflicts with existing code, or the existing version has been optimised to give the absolute best results. The revised version, on the other hand, *has not* been fully optimised.

Since these two system calls show a fairly consistent difference, it would not be unreasonable to assume a similar bias in the other benchmarks, and indeed it can be seen that most of the revised tests are outperformed by 10%-20%. For instance, the PDX call with an extended APD is 7% slower in the revised version, thread creation is 18% slower, task deletion is 14% slower, and so on. With appropriate optimisations, the operations in the revised model may be able to perform as well as the existing model.

There are however some obvious problems in the revised version. Of particular note is inter-domain thread creation which is substantially slower than its analogue (task creation) in the existing model. This is surprising, as the only substantial difference in the code is the fact that inter-domain thread creation must validate a PD object prior to validating each clist that makes it up. Task creation in the existing model follows a similar procedure, the only real difference being that it does not have to validate a PD object before validating some clists.

Another large difference is noticeable with PDX calls using a new APD. In the existing model, a set of clists is passed to the call and the new APD is constructed from these. In the revised model, a PD object is used as the container for these clists. These two tests would seem to indicate that the PD object is something of an overhead, either touching it, or validating it appears to be slowing down the system. Whether this difference is significant enough to cause concern is debatable.

Although the results for the revised model were not as good as hoped, they are not terrible. The aim of the thesis has been met — a more flexible execution and protection model can be implemented in Mungi without overly reducing its inherent speed. Although execution is affected in some cases, it is not significantly bad enough in *any* case to prevent the model from being adopted. Certain optimisations are possible (as discussed elsewhere) which should be able to reduce the differences, and bring the revised model into closer range of the existing model. Since the revised model still outperforms other operating systems by a large amount[3], the flexibility the revised model brings seems affordable.

[3]This has not been verified, but is based on a comparison of existing and revised models, and results published previously.

Chapter 6

Related Work

Opal ([CLFL94]) is a SASOS developed by Chase et al. at the University of Washington. It has a similar execution and protection model to that proposed here; ie. it completely decouples execution from protection by using the two orthogonal concepts of threads and protection domains.

> Protection in Opal is independent of the single address space; each Opal thread executes within a protection domain that defines which virtual pages it has the right to access. The rights to access a page can be easily transmitted from one process to another. The result is a much more flexible protection structure, permitting different (and dynamically changing) protection options depending on the trust relationship between cooperating parties. We believe that this organization can improve both the structure and performance of complex, cooperating applications.

The protection domains of Opal allow overlapping segments of memory that can be used to share data between programs.

However, the protection model which Opal uses is a little different from Mungi. Opal uses password capabilities (like Mungi), but to construct a protection domain, one must specifically *attach* and *detach* individual capabilities. Mungi's analagous method is to *insert* or *remove* entire clists. Clists can be built up of arbitrarily many capabilities (and can be modified without kernel intervention). This results in both fewer system calls, and a simpler way of confining untrusted threads.

Another difference is Opal's notion of *portals*. A protection domain can be arranged with portals which control the access of other threads that wish to use them. Any thread that knows the 64-bit ID of a portal can use it (they are effectively sparse capabilities), but once a thread enters a portal, it begins executing code specified by that portal. This is semantically equivalent to the PDX mechanism in Mungi. It is supposed to allow threads to call other routines which require more access rights. The disadvantage of this method as opposed to Mungi's more general PDX system is that access to a portal cannot be selectively revoked.

Grasshopper ([DdBF$^+$94]) is an operating system that is designed to support orthogonal persistence:

> Grasshopper relies upon three powerful and orthogonal abstractions: containers, loci and capabilities. Containers provide the only abstraction over storage, loci are the agents of change (processes/threads), and capabilities are the means of access and protection in the system.

Protection in the system is facilitated by capabilities. These control access over containers and loci. Containers hold sets of data, and any number of loci may execute in them. In this sense, a container is like a Mungi protection domain and a locus is like a thread. An important difference in the Grasshopper system is that a locus may change containers throughout its lifetime, whereas in Mungi, a thread is constrained to its APD for its duration (discounting PDX calls). However, since APDs are dynamically modifiable, this is not a drawback.

Chapter 7

Further Work

There are several features that could yet be implemented. The first of these would be the exploration of the alternative APD design (c.f. section 3.4.4). It would be interesting to analyse the consequences of this design on how programs executed, and how (if at all) performance was impacted.

The PD object cache (c.f. section 3.4.3) would be a worthwhile feature to implement if the template-based APD design was retained. This optimisation would probably have a fairly significant impact on reducing the cost of repeatedly creating inter-domain threads using a specific PD object. This optimisation is similar to the way that PDX calls are cached to avoid future re-validation.

Although PD objects are now potentially persistent, it would be desirable to eventually provide a level of persistence beyond this. The original topic of this thesis, “Persistent tasks in Mungi”, evoked the thought of persistent user sessions whereby a user could log on and commence from the exact point at which they left their account. The work for this thesis quickly diverged from the original topic, but this type of session persistence should be more easily accommodated now that at least protection domains are in some way persistent.

Chapter 8

Conclusion

The main aim of the thesis was to demonstrate that a more flexible execution and protection model, based on decoupling threads and protection domains in the existing task-based hierarchy, could be implemented without significantly affecting the large speed advantages Mungi has over several other operating systems. A secondary goal was to design a system whereby protection domains could be stored persistently, distributed around the network, and shared between users.

These goals have been achieved.

Mungi tasks have been removed from the model. Threads are arranged in a hierarchy such that each thread has a single parent, and possibly many children. Semantically, each thread belongs to a logical protection domain which is the set of objects it may access. Physically, each thread belongs to an APD which is an instantiation of a PD object, plus a “private” stack object.

PD objects are persistent objects which threads may use to create new threads in protection domains different to their own. They are basically templates for APDs, and can be used by any thread on the network that has access to them. The ability to start threads with a PD object can be passed between threads so as to make them shareable.

Although all relevant system call operations are slightly slower in the revised model (including those that remained unchanged from the existing version), this seems mainly due to the fact that they have not yet been fully optimised. Some operations are significantly slower in the revised model, and this is a cause for concern. However, these operations do not negate the inherent speed of Mungi when compared to other operating systems, and it is likely that after a more intensive study, they will be optimised.

In any case, this thesis was a proof of concept and an exploration of alternative models as opposed to a final solution. The work outlined here should provide a strong basis for further development of the Mungi execution and protection model.

Bibliography

[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.

[DdBF$^+$94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

[Gos91] Andrzej Goscinski. *Distributed Operating Systems — The Logical Design*. Addison-Wesley, 1991.

[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.

[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/.

[WLP75] W. Wulf, R. Levin, and C. Pierson. Overview of the HYDRA operating system development. In *Proceedings of the 5th ACM Symposium on OS Principles*, pages 122–131. ACM, 1975.