THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A Thread Model for Mungi

Andrew Baumann

Bachelor of Engineering (Computer Engineering)

November 2002

Supervisor: Prof. Gernot Heiser

Abstract

This thesis presents the design and implementation of a new thread model for the Mungi operating system. Threads (or processes) form an essential part of any system, and for high performance the thread primitives provided must be lightweight. However, this is not the only consideration: the components of a thread model, especially the programming interface and access control mechanisms, must also conform to the general design philosophy of the operating system.

In addressing these requirements, a thread model was designed for Mungi which represents and identifies threads as memory objects, and enforces access control through the usual protection mechanisms of the system. A prototype of the new model has been implemented, with special care taken regarding the performance of thread creation. Results show that it outperforms the previous Mungi implementation, as well as Linux, in all common cases.

Contents

List of Figures

List of Tables

1. Introduction

Any time-sharing operating system must provide some form of thread (or process) abstraction to allow concurrent execution of user tasks. These threads form the method of execution for programs, and hence it is important that the interfaces used for threading are flexible. For good performance of a system, the provision of low-overhead threads is also desirable.

Mungi is a research operating system being developed at the University of NSW, which features a single address space, as well as capability-based protection. The threading primitives, or *thread model* provided by the current Mungi system suffer from a number of severe drawbacks, the most serious being an inflexible protection mechanism, which hinders application design and programming.

This thesis will design and implement a new thread model, which addresses the shortcomings identified in the current model, and is aimed at replacing the existing version. In particular, the following goals for the design and implementation have been identified, and will be discussed in further detail at a later point:

- high performance thread primitives;
- simplicity and flexibility of the provided model;
- security through generic access control mechanisms.

The design that will be selected is based on the primary idea of identifying and representing threads as memory objects, the main abstraction provided by Mungi. This directly results in a simple thread model, which meshes well with Mungi's other abstractions. Furthermore, an implementation of this model can be shown to have good performance in comparison with other systems.

1.1. Overview

Chapter 2 introduces the necessary background information required for an understanding of this thesis. This includes introducing thread concepts, such as kernel- and user-level threads, and the relevant features of the Mungi operating system, and provides an overview of the current Mungi thread model, which this thesis aims to replace.

Chapter 3 defines in more detail the problem to be addressed. It introduces the basic requirements of a thread model, discusses the problems experienced with the current thread model, and shows how it fails to meet those requirements. Finally, it outlines several key goals to be achieved in designing and implementing a new thread model.

Chapter 4 explores previous work in this area, in the form of the design decisions made by other systems. Relevant information regarding the thread models provided by twelve different systems is presented, and conclusions drawn which are relevant to a Mungi design.

Chapter 5 details the design process, and final design of a new thread model for Mungi. The issues raised in designing this new model are discussed at length, and the final design is presented.

Chapter 6 describes the implementation of the new thread model within the context of Mungi running on top of the L4/Alpha microkernel. The key components of the implementation are introduced, followed by a detailed look at an optimisation targeting fast intra-domain thread creation. The chapter goes on to cover other aspects of the implementation, including an overview of the structures used for kernel-level thread management, and concludes with a discussion of the implementation's portability to other versions of L4.

Chapter 7 presents the measurement and results of a number of benchmarks performed on the implementation described in Chapter 6. Microbenchmark results are presented for four systems, as well as profiles of system call costs, and a benchmark specifically illustrating the performance of local thread creation.

Chapter 8 further analyses the results presented in Chapter 7, as well as discussing L4-specific issues encountered in the design and implementation stages, and proposing solutions to these problems.

Chapter 9 draws some conclusions from the outcomes of the thesis, and discusses the extent to which the design and implementation have met the goals set out in Chapter 3. Finally, it discusses a number of avenues for future work, left unexplored by this thesis.

Appendix A contains the C-language function headers for the system call bindings implemented by this thesis, for reference purposes.

Finally, **Appendix B** documents a user-level convenience library that simplifies programming with the system call primitives introduced by the new thread model.

2. Background

2.1. Threading

Much time has passed since the days when all computer programs could be expressed as simple sequential operations, and were executed in this fashion. Many modern programs make use of concurrency, and in order to take any advantage of multiprocessor systems, some form of program parallelism is required. Typically this parallelism is provided by the operating system in the form of the *thread* abstraction.

2.1.1. Introduction to threads

A thread is the basic unit of execution, and provides a single sequential execution stream. A single program may consist of one or more threads, logically executing in parallel. In a multiprocessor system this execution really can be parallelised, with different threads executing on different processors. In a single processor system, true parallelism is impossible, but the processor is transparently multiplexed between the running threads.

Most modern, general-purpose operating systems make a clear distinction between a thread, and the other components of a complete program such as an address space (or alternatively, protection domain in a single-address-space system, as will be discussed in Section 2.2). A thread is the relatively lightweight unit of execution—it consists of some state information, a context of execution (basically the saved CPU register state), and usually a stack. One or more threads form part of a *process* (or *task*), which is a heavyweight object forming the basis of protection and control. As well as its threads, a process consists of the address space or protection domain which all the process' threads share.

2.1.2. Kernel- and user-level threads

Almost all modern operating systems support threads in some form, but as will be seen in Section 4, there are differences in what level of support for threading is actually provided by the kernel, as opposed to user-level libraries.

Some systems (notably Linux [DM02], Windows NT [Cus93], and L4 [Lie95]) provide threads as a kernel-level abstraction. The kernel's scheduler knows about all threads in the system, and makes appropriate scheduling decisions for them. This is relatively straightforward, however since the kernel must know about all threads, there is a memory overhead for each thread, and thread creation and deletion require a system call.

Fully user-level threads

The alternative is to move some or all of the thread management out of the kernel. In many older systems, the kernel supports only single-threaded processes, and multiple contexts of execution within a process are provided by a user-level library. This thread library maintains the thread-related state and switches the single kernel-provided context of execution between the threads to provide concurrency.

The advantages of such a system are primarily that it is easier to implement, and provides cheap thread creation and deletion (since the kernel does not need to be involved). There are many problems, however—if one thread blocks the process (for example, by page faulting or performing IO) no other threads can run. Furthermore, multiple threads cannot take advantage of multiple CPUs to run in parallel.

User-level threads with kernel support: scheduler activations

To address the problems of user-level threads, systems have attempted to move some support for user-level threading into the kernel without making use of full kernel-level threads. Most systems taking this approach (notably Solaris [YL98] and K42 [IBM01b], although more are discussed in Chapter 4) are based on a design called scheduler activations [ABLL92,Dre02a].

The primary problem with user-level threads which scheduler activations aim to address, is that the kernel scheduler makes decisions about scheduling processes without knowledge of what is running within processes. In this system, the kernel scheduled entities are called *virtual processors*, and a user process may have one or more virtual processors allocated to it. When a virtual processor is to be executed (i.e. the kernel scheduler has decided to give it time on a real processor), a *scheduler activation* upcall takes place. These upcalls are invocations of the user-level scheduler for each virtual processor, which selects the user-level thread that should actually be run.

If the user thread performs an action which would block in the kernel (for example, IO), instead of scheduling another virtual processor, the kernel performs another scheduler activation upcall to the user-level scheduler to allow it to pick a new user-level thread to run on the virtual processor. The four types of upcall are shown in Table 2.1. This design avoids the problem of all user-level threads being blocked when only one needs to be.

2.2. The Mungi operating system

Mungi [HEV+98, Mun02b] is a single-address-space operating system (SASOS) being developed by the Operating Systems Research Group at the University of NSW.

Add this processor (processor #)
Execute a runnable user-level thread.

Processor has been preempted (preempted activation # and its machine state)
Return to the ready list the user-level thread that was executing in the context of the preempted scheduler activation.

Scheduler activation has blocked (blocked activation #)
The blocked scheduler activation is no longer using its processor.

Scheduler activation has unblocked (unblocked activation # and its machine state)
Return to the ready list the user-level thread that was executing in the context of the blocked scheduler activation.

Table 2.1: Scheduler activation upcall points. Source: [ABLL92].

A SASOS [WMR$^+$95] is a system in which all threads share the same virtual address space, and all data on the system, including the kernel's, is addressable within this space. This is fundamentally different from traditional multiple-address-space operating systems (MASOSes) such as UNIX, in which each process has its own unique address space.

The following subsections introduce the main Mungi concepts required for an understanding of the thread model.

2.2.1. Capabilities

One issue where a SASOS obviously differs from a MASOS is in how protection is handled. In a MASOS, each process has access to any data in its address space, such that address spaces form the basis of protection. In a SASOS it would be insecure to allow a thread unrestricted access to the entire shared address space, so some other form of protection must be used. To address this problem, Mungi uses a capability system to restrict user threads' access to a subset of the address space, called a *protection domain*. Attempts to access virtual addresses outside the protection domain of a thread result in a protection fault.

A *capability* is represented as a 128-bit value, consisting of the 64-bit address of the object to which it confers certain privileges, and a 64-bit secret password (see Figure 2.1). Users can freely create and copy capabilities—they are protected from forgery by the randomly chosen password.

virtual address$_{64}$	password$_{64}$

Figure 2.1: Representation of a capability.

Capabilities can grant a combination of *read*, *write*, *execute*, *destroy* and *enquire* rights to an object—a capability granting all five of these is an *owner capability*.

2.2.2. Objects

Objects are the regions of the virtual address space which form the basic unit of protection and sharing in Mungi. An object consists of one or more contiguous pages, and may be created by any thread on the system (subject to a resource control mechanism which is not discussed here). The system maintains a data structure known as the *object table* (OT), which stores information about each object in the system, including the passwords registered for that object, and the rights which they confer. Users holding an owner capability to an object may register and delete passwords for it (thereby effectively creating and deleting valid capabilities).

2.2.3. Active protection domains

As mentioned previously, the set of objects to which a thread has access is called its protection domain. In Mungi, the protection domain of a thread is described by a data structure known as the *active protection domain* (APD).

An APD is a kernel-level data structure consisting of capabilities to one or more user-level *capability lists* (Clists), which is instantiated from a protection domain (PD) object, containing the same data. A Clist is an otherwise ordinary object which contains a set of capabilities stored in a predetermined format—presently there are two formats defined: unsorted and sorted Clists (using a sorted Clist allows the kernel to perform faster access validations). Figure 2.2 shows an example APD containing two Clists.

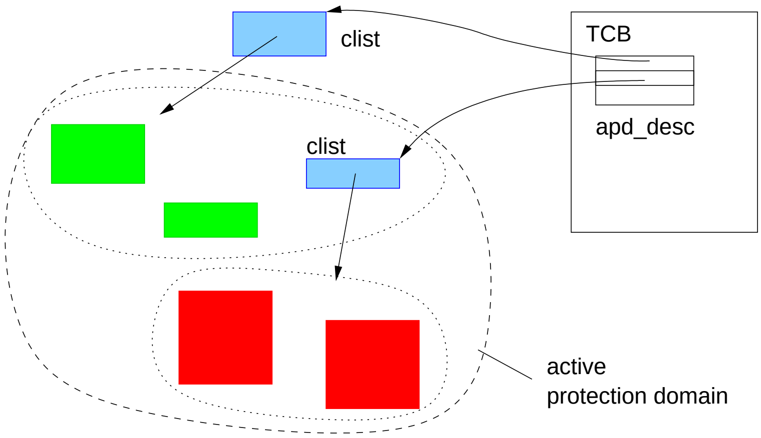

Figure 2.2: Active protection domain of a thread. Source: [Mun02b].

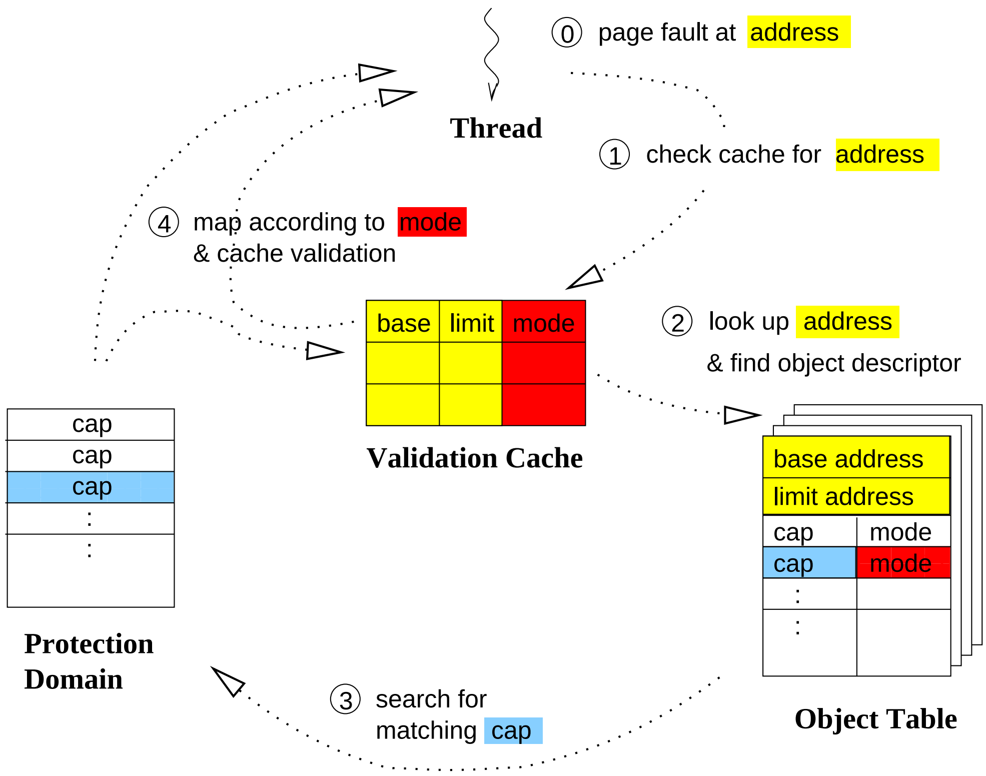

Figure 2.3: Access validation mechanism. Source: [Mun02b].

When a thread attempts to access an object, the kernel first locates the corresponding OT entry, and then searches through the Clists that make up the thread's APD for a capability with a password matching one of those registered in the OT. When a matching password is found, the appropriate rights (also stored in the OT) are granted to the object, alternatively, if the validation fails, a protection fault is raised.

To speed up the costly validation procedure, a validation cache (VCache) is used for each APD—this cache stores recently validated object accesses, and is consulted before performing the full validation. For security reasons, entries expire from the VCache after a fixed timeout. This validation and cache mechanism is depicted in Figure 2.3.

Since it is usually not necessary to run every thread in a distinct protection domain, multiple threads can share an APD (and benefit from the reduced overheads this entails). It is also possible to confine a thread so that it cannot modify its Clists or APD.

2.2.4. Protected procedure calls

For many applications, it is necessary to provide a mechanism for protected code execution (for example, set user-ID or set group-ID programs in UNIX). In Mungi, this is the protected

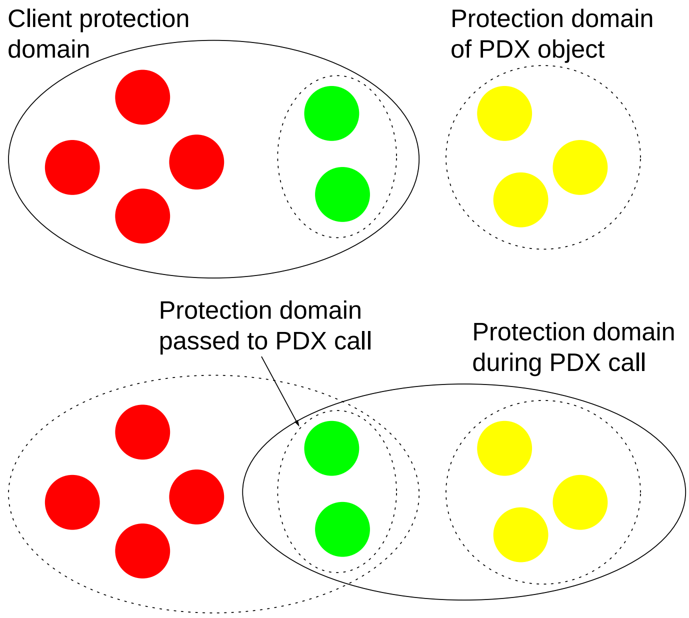

Figure 2.4: Protection domains during a PDX call. Source: [Mun02b].

procedure call mechanism, known as *PDX* [VERH96, Voc98].

As well as normal passwords, users can register PDX capabilities for objects to which they hold an owner capability. A registered PDX entry point consists of the address of the code to be executed, as well a Clist.

Users holding a capability to a PDX entry point may use the *PdxCall* system call to invoke the code, which runs in the protection domain formed by the union of the caller's APD, and the PDX's registered Clist. It is also possible to call PDX code with only a subset, or even none, of the caller's APD. Figure 2.4 illustrates the change of a thread's APD that occurs during a PDX call.

This mechanism allows for secure, temporary rights amplification in Mungi.

2.2.5. Mandatory access control

Mungi's security model, as determined by the system of capabilities, objects, and APDs presented so far, is a form of *discretionary access control*. The system's users have discretion over who they allow to access the objects they own (through the registration and transfer of password capabilities). Such discretionary control is present in all general-purpose operating systems, and is needed to allow users privacy. It also allows the security principle of least privilege, which requires that a user be given no more privilege than necessary to perform a task, to be observed.

Mandatory access control (MAC) [DoD86], on the other hand, allows the enforcement of system-wide security policies. It is generally concerned with implementing security policies that are coarser grained than individual objects, for example, ensuring on a military system that users with top secret clearance cannot leak such information to objects accessible to users with only secret clearance.

Previous work has implemented MAC support in Mungi [Edw00]. Fundamentally, the Mungi MAC implementation involves:

- associating a *domain* label with each thread and PDX entry point;
- associating a *type* label with each object and thread;
- validating each object access or thread control system call by passing the type label, domain label, and attempted operation to a global user-level policy object, which may allow or deny the access.

Obviously, querying a user-level policy object for each validation would be prohibitively expensive, so the results of these validations are cached inside the Mungi kernel. Additionally, since multiple threads share an APD, and all threads within an APD have the same security rights, the domain and type labels of each thread are actually stored with the thread's APD.

In this way, Mungi provides a combination of both discretionary and mandatory access control, since both must agree before any access is allowed.

2.2.6. Bank accounts

In order to prevent unrestricted use of virtual memory backing store by objects, Mungi has an economic resource control model based on *bank accounts* [Lam95, HLR98].

Bank accounts hold virtual *money*, which is periodically used to pay *rent* for objects associated with the account. The accounts are also periodically topped up with a *salary*, and have *taxation* applied to them (to avoid idle accounts accumulating large balances). Two threads, the *rent collector* and *pay master*, are involved in making these adjustments to bank account balances, based on their contents, and the objects associated with them.

As applied to Mungi, bank accounts are special objects (a concept discussed in the following section) with a flag set in their OT entry—those with money available for rent are considered *financial*. The system ensures that every object created is associated with a financial bank account, and that the rent collector thread is able to set overdrawn accounts as non-financial, and to remove objects whose bank accounts no longer exist.

The details of rent, salary, and taxation are discussed elsewhere, but they essentially allow flexibility in the system, and the implementation of most of the policy at user-level.

2.2.7. Special objects

Certain types of objects in Mungi are *special*, in that they are required by certain system calls, and the system restricts their creation. Currently, Mungi uses special objects for protection domains and bank accounts.

Every special object is associated with another special object of the same type called a *controlling object*. An object can only be made special if the thread has a read capability to such a controlling object. For each type of special object there must be a *master object*, which has itself as a controlling object. There are further restrictions on the creation of master objects.

The final difference between special objects and normal objects is that the execute permission has a changed meaning. An execute capability to a special object conveys permission to use the object (for example, start a thread in a protection domain), without being able to read its data.

2.2.8. Semaphores

Since Mungi provides shared memory as the basis of communication, it must also provide a primitive for concurrency control to make shared memory usable. Mungi's concurrency control mechanism is the semaphore.

Each semaphore is associated with and named by an address in the SAS, although the contents of the byte addressed are independent of the state of the semaphore. Capabilities to the object containing the semaphore's address are used for protection over the semaphore system calls.

The system calls are all fairly straightforward. *SemCreate* and *SemDelete* create and delete semaphores—*SemCreate* allows the user to specify an initial value for the semaphore, as well as (currently unimplemented) flags that control the order in which waiters are woken when the semaphore is signalled. *SemWait* and *SemSignal* provide the actual synchronisation mechanism. *SemWait* blocks until the semaphore to becomes positive, and then decrements it, while *SemSignal* increments the semaphore.

Other work [Gal00] has proposed and implemented a number of extensions to the Mungi semaphore model.

2.2.9. Persistence

Most traditional systems require data to be loaded into RAM before it can be used, but require it to be saved to stable storage (usually disk) if it is to outlive the program. A system supporting orthogonal persistence removes this requirement, so that the application does not need to concern itself with the complexities of loading and saving data. Mungi is designed to easily support orthogonal persistence, and an initial implementation of persistence has recently been completed [NBH02].

The implementation relies on performing regular *checkpoints* of threads in the system. A checkpoint involves stopping all threads, saving their state and the state of virtual memory to stable storage, and then restarting all threads. Checkpoints are taken at regular time intervals (in the order of minutes), and in order for persistence to come at an acceptable cost, the critical section when all threads are stopped and restarted must execute as quickly as possible.

2.3. The current Mungi thread model

2.3.1. Thread model

Threads in Mungi are the basic kernel-scheduled unit of execution. Each thread executes in an APD, which defines the objects it can access (see Section 2.2.3). Threads may be created inside the creator's APD, or in a new APD described by a protection domain special object.

Access control under this model uses a strict hierarchy of threads based on creator. All calls affecting an existing thread can only be used on descendants of the caller. Only threads descended from the caller can be killed, and descendants of a killed thread only survive if they are adopted by the thread higher up performing the kill.

2.3.2. System calls

The thread control system call interface under the existing model is shown in Table 2.2, and detailed over leaf.

System calls:		
ThreadCreate	(*entry_pt*, *param*, *info*, *pd*)	$\rightarrow$ (*thread_id*)
	fails: *protection_violation*, *table_overflow*, *invalid_info*, *invalid_Clist*	
ThreadDelete	(*thread_id*, *status*, *adopt*)	
	fails: *protection_violation*, *invalid_thread*	
ThreadSleep	(*thread_id*, *time*)	
	fails: *protection_violation*, *invalid_thread*	
ThreadResume	(*thread_id*)	
	fails: *protection_violation*, *invalid_thread*	
ThreadWait	(*thread_id*)	$\rightarrow$ (*thread_id*, *status*)
	fails: *protection_violation*, *invalid_thread*	
ThreadMyId	()	$\rightarrow$ (*thread_id*)
	never fails	
ThreadInfo	(*thread_id*, *thread_info*)	$\rightarrow$ (*thread_info*)
	fails: *protection_violation*, *invalid_thread*	

Table 2.2: Thread-related system calls in the current Mungi API [Mun02b].

ThreadCreate: Creates a new thread which starts to execute at the address given by *entry_pt*, with argument *param*. If *pd* is null, the thread will be created in the caller's APD. If *pd* is non-null, it must be a pointer to a protection domain special object to which the caller holds an execute capability—in this case the thread is created in the APD described by *pd*.

The *info* buffer specifies optional parameters for thread creation, its contents are shown in Figure 2.5. A stack pointer for the new thread may be specified in *info*, if none is given the kernel allocates a stack from the APD's system object. A number of flags may also be specified in *info*, the *detached* flag controls whether the thread will be cleaned up immediately upon deletion, or whether *ThreadWait* can be used, and the *join* flag controls whether a thread created in another APD can share the domain with other threads created from the same protection domain object (and also be joined by future threads created in the same PD).

Finally, the *info* parameter allows the specification of an *environment* buffer. This is by-value data copied by the kernel to a thread from its creator.

ThreadDelete: Kills a thread, which must be a descendant of the caller. The *status* value is delivered to threads performing a *ThreadWait* call, unless the thread was created with the *detached* flag set. The caller may opt to adopt any children of the killed thread by setting the *adopt* parameter, if not they are all killed in the same call.

ThreadSleep: Stops a thread, which must be a descendant of the caller, for a specified interval or until explicitly woken with *ThreadResume*. Also allows the caller to perform a yield operation, by specifying a zero interval.

ThreadResume: Prematurely resumes execution of a sleeping thread, which must be a descendant of the caller.

ThreadWait: Waits for a specific thread, or any thread, which must be a direct descendant of the caller, to be killed. Returns the status value given when the thread was killed, as well as its thread ID.

ThreadMyId: Returns the caller's thread ID.

ThreadInfo: Reads and sets the attributes (priority, time/memory limits) of a thread, which must be a descendant of the caller. Limits can only be restricted, not relaxed. The data in *thread_info* is shown in Figure 2.5.

$$
\begin{array}{l}
\underline{\textbf{type}}\ \mathit{ThreadInfo}\ \equiv\ \{\\
\quad \mathit{stack},\ \mathit{stack\_size},\ \mathit{environment},\ \mathit{env\_size},\\
\quad \mathit{mem\_limit},\ \mathit{time\_limit},\ \mathit{cpu\_time},\ \mathit{priority},\\
\quad \mathit{start\_time},\ \mathit{flags},\ \mathit{bank\_account}\\
\}
\end{array}
$$

Figure 2.5: Thread descriptor data structure. Source: [Mun02b].

3. Features of a Thread Model

What should be considered when designing a thread model? An example of a thread model has just been seen in the current Mungi API, but this chapter outlines more generally the components that make up a thread model, analyses the flaws in the current model that make it worth replacing, and introduces several goals in designing and implementing a replacement thread model.

3.1. Requirements

3.1.1. Thread control primitives

The most obvious parts of a thread model are the primitives for controlling threads. These may be expected to consist of:

- some unique identifier for each thread running on the system, usually in the form of a thread or process ID;
- a way (usually involving system or function calls) of starting and stopping threads;
- possibly, control over the execution context or protection domain of a thread;
- a mechanism for notification to interested parties of the exit or death of threads;
- a mechanism for obtaining status information of active threads;
- possibly, a way to temporarily stop and resume threads.

The current Mungi thread model defines the thread control system calls and *ThreadInfo* buffer (detailed in Section 2.3) as thread control primitives.

3.1.2. Access control mechanisms

Security and reliability are crucial concerns in almost any system. A thread model for a general-purpose operating system such as Mungi can be expected to include some form of access control mechanism on all thread operations—for example, to stop threads arbitrarily deleting other threads in the system.

In the current Mungi thread model, the thread hierarchy established by thread creation forms the basic access control mechanism, with thread operations only being allowed on descendants (in many cases, direct descendants) of the calling thread.

3.1.3. Resource control mechanisms

An operating system is responsible for sharing the computer's resources between the various users or threads using the system. These resources include such examples as memory, disk space, or network traffic, and in the case of threads particularly, CPU time. There are usually more threads ready to be run than processors available on any given system, so the processor time must be shared in some way between these threads, a decision usually made by the operating system's *scheduler*.

The scheduler uses some form of resource control algorithm to decide which of the ready threads in the system should be run, and for how long. Commonly a *priority* system is used, where each thread is assigned a priority and higher priority threads are favoured by the system over those with lower priority—if the scheduler always selects the highest priority thread, it is considered a *hard* priority system. As well as priorities, which control the relative allocation of CPU time between threads, limits are commonly used which control the total time allocation to one or more threads.

The availability of resource management mechanisms is crucial on a multi-user system, as it provides a method to stop possibly buggy or malicious threads tying up the system's resources to the detriment of other users.

The current Mungi thread model provides little in the way of resource control. It specifies CPU time and memory limit parameters for each thread, and restricts the lifetime of threads created within a PDX call. Related work has provided a possible design for controlling CPU time allocation within Mungi [Che01, CH02], and will be discussed in more detail in Section 5.1.5.

3.2. Problems with the current model

There are a number of problems with the existing model, which this thesis attempts to address. Most of them stem from the original design—Mungi was designed as a research system based on a single address space and password capabilities; since these were the major features of the design, the thread model was given less consideration.

3.2.1. Restrictive hierarchy

The single biggest problem with the existing model is the rigid hierarchy it imposes. This provides an access control mechanism, but one which is totally at odds with the rest of

Mungi, which uses capabilities as the basis of protection.

The current API document notes that the thread hierarchy is unsatisfactory, and needs revision. This hierarchy resulted from previous work [Cut98], which removed a hierarchy of *tasks* from the Mungi design (replacing them with APDs instantiated from PD objects). A hierarchy was maintained when removing the explicit tasks, since it provided a (simplistic) form of resource management—this thesis will argue that resource management can, and must, be provided by other means.

Such a restrictive hierarchy is too inflexible for many common requirements of threaded applications—these problems are particularly prevalent when it comes to thread deletion. Some examples are:

- A thread can only outlive its parent with the cooperation of the deleting thread.
- It is impossible to donate (or reparent) threads without deleting their parent.
- Threads can only be notified of the death of their direct descendants.
- Threads cannot provide a means for others to delete them without the cooperation of their parent. They can delete themselves, but they can't easily transfer this right.

The last point regarding rights transfer is particularly relevant. Rights transfer and delegation is a key feature of capability-based systems, and for such rights transfer to be restricted only to certain types of rights is extremely inelegant. Using capabilities for access control would solve this problem.

The Mungi thread hierarchy design is very similar to the process hierarchy used in UNIX [TR74], where it is generally accepted as a working design. UNIX, however, does not use the hierarchy for access control—it keeps a user ID (UID) for each process, and it is the UID of the process that determines access control. In fact, UIDs are the basis of the majority of access control in UNIX, whereas in Mungi this role is filled by capabilities.

3.2.2. Resource management and scheduling

There are significant problems with resource management in the current model. The primary problem relates to control over the CPU time usage of a thread. There is defined a CPU time limit for each thread (which was previously a per-task limit), but there is nothing stopping a malicious thread whose time limit is about to expire from creating a new thread with a higher limit to continue its work. Additionally, the *ThreadInfo* call does not allow these CPU and memory limits to be relaxed (and presumably scheduling priority to be increased, but this is unspecified) after a thread is started. This enforces a simple resource control policy, because a thread can call *ThreadInfo* on itself, but it is very restrictive.

Secondly, the creation of threads is unrestricted—anyone may call *ThreadCreate*, which only fails when all of the system's resources are exhausted. This leads to potential denial-of-service problems, where rogue threads can create more and more threads until the system runs out of resources with which to service any other users.

The current API states that threads created during a PDX call may be cleaned up at any time after the call. This is another attempt at resource control, since these threads are children of the creator, but the creator did not have control over their creation (the PDX code did). Such a policy is extremely restrictive for Mungi applications, since it forces a synchronous, blocking model of execution on PDX calls. A number of applications, however, operate in an asynchronous manner—using a PDX call which returns directly to start some processing, and then being notified later of its completion. An example of this is the new Mungi device driver model [Les02], which relies upon the fact that PDX-created threads aren't deleted in any current implementation.

Finally, there is no consideration of the need to provide differing scheduling policies or algorithms to users (in fact, even the way that priority values are interpreted is undefined).

3.2.3. Distribution

Mungi is designed to easily support distribution, using a single global address space. In a distributed, or even multiprocessor system, threads must migrate between nodes to balance the overall load, which would require support within the thread model. However, no consideration is made in the current thread model for the needs of such multiprocessor and distributed systems.

3.3. Design and implementation goals

3.3.1. Performance

Threads form the basis of any program, their use cannot be avoided, hence it is crucial that any implementation provide high performance threading primitives.

Of course, different applications have different requirements. Some applications create large numbers of threads that execute for the lifetime of the program, and require low overheads for memory use and context switching time. Other applications continually create and destroy threads, and require extremely fast thread creation and deletion operations.

One particularly specialised class of application, device driver interrupt handlers, requires minimal interrupt handling latency [Les02]—that is, the interrupt handler code must do whatever is necessary to start handling the interrupt as quickly as possible in order not to miss interrupts. Threads are created as part of the Mungi interrupt handling routine. What is important in this particular case is not how quickly a thread starts to execute, but how quickly the thread creation system call returns to the interrupt handler routine. As will be seen in Section 6.3.2, special care has been taken to allow for this case in the implementation.

3.3.2. Simplicity and flexibility

Excessive and unnecessary complexity should be avoided in any large system, and especially software system, for reasons of reliability—a thread model is no exception. As a result, the thread model should be designed to be as simple as possible, while maintaining the flexibility to allow more complex systems to be implemented above it.

Such a stance excludes to a certain extent considerations of usability. This makes sense, since the thread model being designed is to form part of the Mungi kernel API (the introduction to which acknowledges the minimalist nature of the interfaces offered). It is anticipated that application programs will not directly use the Mungi kernel API, but rather convenience libraries implemented at user level. Such libraries would go hand-in-hand with a thread model implementation.

3.3.3. Security

Since Mungi aims to be a general-purpose operating system, any thread model design must provide access control mechanisms, and for reasons of simplicity these should operate within Mungi's protection system.

Furthermore, the access control mechanisms used should be as generic as possible to allow differing security policies at user level. This design principle of separating policy decisions from the mechanisms used to enforce them is a longstanding and accepted one within the field of operating systems, and was first published in the context of the HYDRA system [LCC+75].

4. Review of Other Systems

4.1. Commercial systems

4.1.1. POSIX

Portable Operating System Interface (POSIX) [POS90] is an IEEE standard describing a common UNIX [TR74] system interface.

Standard POSIX provides only processes, which include a single context of execution as well as an address space. Processes are created with the *fork* system call, which duplicates the caller's address space in a new process. If the new process then needs to execute different code, it can use a system call such as *exec* (which executes a file) to do so. This system is clearly inappropriate for single-address-space systems such as Mungi, where the concept of duplicating the address space is meaningless.

Process creation in a POSIX system is usually a very expensive operation, and communication between processes is difficult. To address these problems, the POSIX threads (Pthreads) standard [POS95, NBF96] was developed. It allows a process to contain multiple threads which share the address space. Creating a thread with the *pthread_create* call can be a significantly cheaper operation than *fork*, depending on the implementation of Pthreads—some implementations provide kernel threads, usually with expensive thread creation, while some only provide user-level threads with cheap creation but several other inherent problems, as discussed in Section 2.1.2.

Within a process there is no protection between individual threads (since they all share the address space, this would be useless). Between processes, POSIX uses a process hierarchy along with user IDs to control access to process manipulation system calls. Base POSIX doesn't specify any form of resource control on *fork* or *pthread_create*, although many systems allow per-user process limits to be imposed.

Since Pthreads was developed from the perspective of adding lightweight threads to POSIX, the interface is of little relevance to Mungi—as long as it can be suitably implemented on top of the Mungi primitives if required.

4.1.2. Solaris

Sun Solaris is a modern, commercial implementation of UNIX (and hence POSIX) which places particular emphasis on support for threaded applications. Solaris' kernel, known as

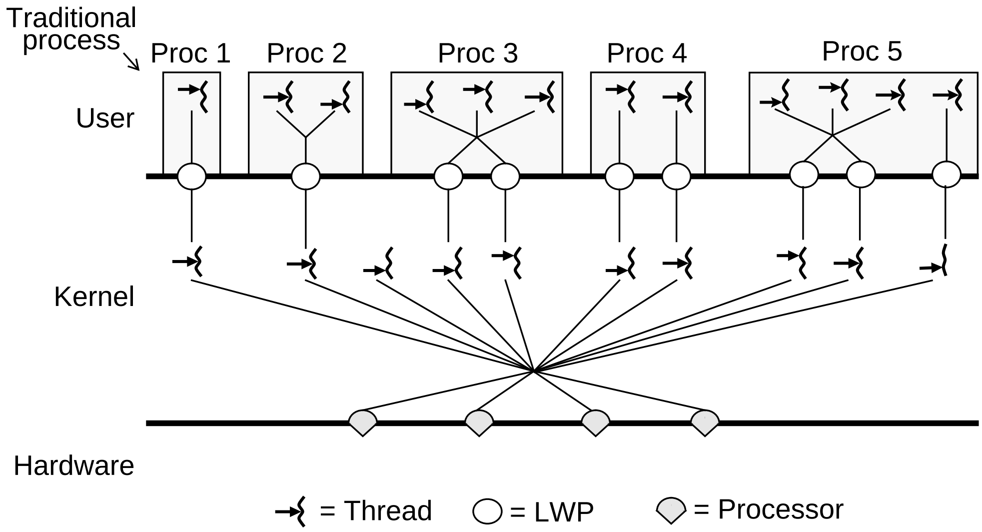

Figure 4.1: Thread structure in Solaris. Source: [EKB+92].

SunOS, underwent significant development around version 5.0 to provide proper support for multi-threading [EKB+92].

The SunOS kernel schedules entities known as *kernel threads*. A kernel thread may execute user code, or perform internal functions of the kernel, such as handling device driver interrupts. The user-level interface provided by Solaris consists of the *light-weight process* (LWP). Each LWP is bound to one kernel thread, and shares the address space, open files, and other components of a UNIX process with the other LWPs in the same process. Finally, user-level libraries implement interfaces such as Pthreads, and multiplex user-level threads onto one or more LWPs. This arrangement is shown in Figure 4.1.

The kernel scheduler schedules kernel threads, and provides several scheduling classes. A scheduling class determines a thread's global priority based on the class, and the priority of the thread within the class. The scheduler runs the thread with the highest global priority, among threads with the same global priority round-robin scheduling is used. The scheduling classes supported are system, timesharing, and real-time.

To provide support for efficient user-level threading, Solaris LWPs use an upcall mechanism [YL98] based on scheduler activations [ABLL92]. This allows for dynamic allocation of user-level threads to LWPs, resulting in better performance.

Solaris is mainly of interest to Mungi as an example of a real-world system using scheduler activations. Whether such a design could be used within Mungi is another issue, however, which will be considered in Section 4.4.

4.1.3. Windows NT

Windows NT [Cus93] is Microsoft's main operating system platform. It was designed to support multiple operating system environments, including Windows, POSIX and OS/2 applications—although in recent releases, only the Windows environment is present.

Due to this multiple-environment design the thread primitives provided by the Windows NT kernel (called the "executive") are fairly generic, as they must be used to support POSIX-style hierarchical processes as well as Windows processes which have no hierarchy. The executive provides process and thread objects, where a process consists of an address space as well as other information, and a thread is a single context of execution within a process. The POSIX subsystem, for example, uses a process and one thread to represent a POSIX process.

Access control is provided by Windows NT's general security mechanism, whereby each object has an *access control list* (ACL) which allows the creator of the object to specify the operations that may be performed on it by other elements of the system. This use of ACLs effectively moves the security policy out of the kernel—a feature that Mungi would do well to mirror with capabilities.

Besides limiting thread and process operations to the protected subsystems (such as Windows or POSIX), the NT executive doesn't provide anything in the way of resource management—this is left to the subsystems to deal with.

4.1.4. IBM AS/400

The IBM AS/400 [Sol97] (previously the System/38, and recently renamed again as the iSeries) is a long-standing commercial system, with a unique tightly coupled hardware platform and operating system. It is aimed at the mid-range server market (particularly for database applications), and is primarily of interest since it also uses a single address space design (although this is not visible at the user level).

The AS/400 has an unusually high-level interface to the operating system, the technology independent machine interface (TIMI, or just MI). The internal implementation of the MI uses a lightweight task structure, which is basically equivalent to our concept of a thread. However, above the MI only heavyweight processes are available, and the operating system (OS/400) extends this with the concept of jobs. The implementation of POSIX threads on the AS/400 uses these jobs, although it caches created jobs to reduce the thread creation overhead.

There is little here of interest to a Mungi design.

4.2. Single-address-space systems

4.2.1. Angel

Angel [MSS+93] is one of the earliest SASOS designs, and unlike many others it uses a form of access control list instead of capabilities for protection.

Angel's process model is based on scheduler activations [ABLL92] (introduced in Section 2.1.2), and centres on the kernel-provided *virtual processor* (VP). The VP consists of a context of execution and two data structures: its *domain descriptor list*, which describes the protection domain in which the process executes; and its *upcall list*, which describes the behaviour of the process in response to a number of kernel and user generated upcall events (this forms a method for preemption when an important event occurs).

Threaded processes aren't explicitly supported by Angel, however the kernel provides support for user-level threading, by generating an upcall whenever a thread blocks, to allow the VP to schedule another thread. Angel uses these features to provide a Pthreads interface (described in Section 4.1.1) at user-level.

4.2.2. Opal

Opal [Cha95] is a capability-based SASOS, developed for research purposes at the University of Washington, which is similar to Mungi in many ways. It provides objects (known as *segments*), threads, protection domains, protected procedure calls (through entry points called *portals*), and *resource groups* much like bank accounts.

The thread structure in Opal is provided by a user-level threads package, called *OThreads* [FCL93], also based on scheduler activations. OThreads requires user applications to label calls which might block in the kernel, and uses special mechanisms to avoid some of the traditional problems associated with user-level threads. The Opal kernel provides primitives for processor allocation to this threads library, which provides *NewThread* and *Join* (like Mungi's *ThreadWait*) operations to users. It is unclear what access or resource control methods are used on these calls.

4.2.3. Sombrero

Sombrero [SMF96] is a SASOS developed at Arizona State University, using special hardware support. The details described here are taken from the published system interface [Car96], although it is unclear to what extent this been implemented.

Sombrero runs on specialised hardware, which provides protection domains which may be changed by executing at special registered entry points—this is roughly equivalent to a Mungi PDX call, but since it is provided by hardware there is no need for a protected operating system kernel in the traditional sense. Instead, all the services associated with an OS are provided as protected entry points to objects and are executed at user level. These are the Sombrero system services.

The *SysThread* class of the system services represents a thread of execution. This class provides methods for:

- creating and deleting threads;
- reading and setting a thread's registers;
- suspending, resuming, and signalling (with abort, terminate, or interrupt signals) a thread;
- reading and setting CPU time limits;
- querying a thread's protection domain and owner, and transferring ownership to a different user;
- monitoring a thread;
- selecting scheduling type (see below);
- querying a thread's current node (in a distributed system), migrating a thread between nodes, or maintaining a list of nodes that a thread can execute on.

The three scheduling types supported in Sombrero are:

Multi: The thread executes at a specified priority, and is preempted by queue-based or cyclic threads.

Queue-based: The thread runs to completion unless a cyclic thread preempts it, there are no scheduler parameters for this type.

Cyclic: The thread runs at set intervals, but cannot take longer than a set percentage of the interval or it is aborted. This seems to be designed for real time applications.

Sombrero defines an extensive set of thread related operations, although it is unclear what form of access control is used on these operations. The interface is fairly complex—it wouldn't be wise to implement all these features in the Mungi kernel, although adequate support should be provided to allow it at user level.

4.2.4. Nemesis

Nemesis [RF97, Nem00] is an operating system designed specifically for real-time applications requiring quality of service (QoS) guarantees, such as multimedia applications. One of the main requirements of such applications is to share large amounts of data, which is facilitated by the use of a single-address-space design. However, Nemesis is not a pure SASOS in the same sense as Mungi, since it does not aim to make objects in the SAS persistent, and it does not use the SAS for distribution.

One of Nemesis' key goals is to account as much resource use as possible to the end user code, this is achieved by having an extremely small kernel and performing most functions in user-level shared library code. One of the results of this minimalist design is the lack of kernel-level thread support. Instead, the Nemesis thread system (which is related to a design developed in Fawn [Bla94]) follows an approach somewhat similar to that of Angel and Opal, which is also based on scheduler activations [ABLL92]. The kernel multiplexes the CPU to a number of virtual processors, and allows QoS guarantees to be provided for this operation. Threaded applications would then be implemented within one or more virtual processors.

The design of Nemesis attacks a fairly specific problem (multimedia applications), whereas Mungi intends to be a general-purpose SASOS. Hence the QoS guarantees, which are required for real-time applications, are not as relevant for Mungi. The use of virtual processors is a possibility, and will be analysed further in Section 4.4.

4.3. Other systems

4.3.1. L4

L4 [Lie95, Lie96] is a high-performance microkernel with implementations on a number of architectures. The current prototype of Mungi is implemented on the MIPS [EHL99] and Alpha [PWH01] versions of L4 (a port to the IA-64 architecture is in progress), and hence any successful thread model must map reasonably cleanly to the L4 primitives.

Version 2 API

The L4/MIPS and L4/Alpha implementations (as well as others) currently in use conform to version 2 of the L4 API. This version will be the target of the implementation, its details are presented here.

L4 provides *thread* and *task* abstractions. A task consists of an address space, and multiple threads which share the address space. Tasks may be started with the *task_new* system call. The *lthread_ex_regs* system call is used to start and modify threads, and only operates on threads in the same task as the caller.

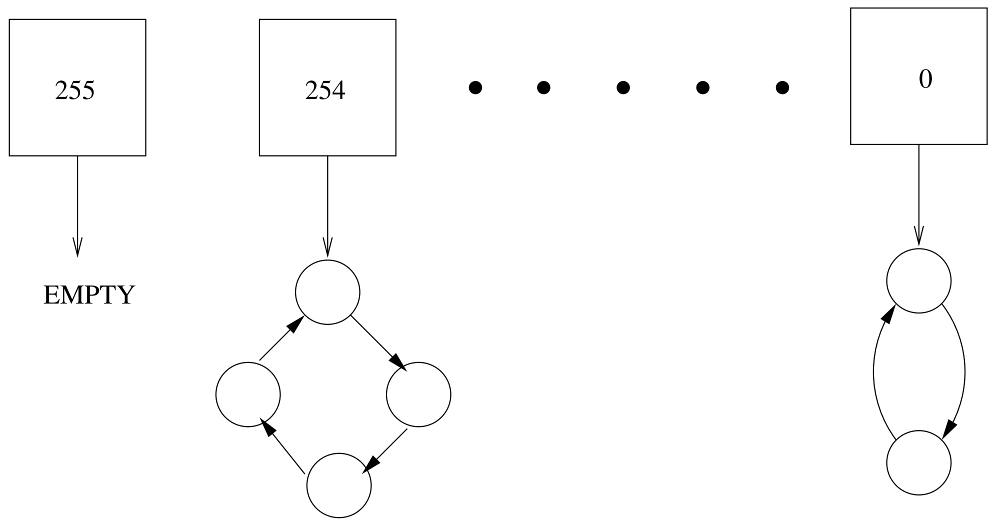

Figure 4.2: Example L4 scheduler ready queue, each circle represents a thread. Source: [AH98].

Resource control is based on a simple first-come first-served mechanism. There are a limited number of tasks, which may be created only if they do not already exist. Once a task has been created, it is associated with its creator (its "chief") in a hierarchical fashion, and cannot be used by another task until deleted by its chief. To allow tasks to be allocated without using resources, they may be created in an inactive state, with no running threads. The task hierarchy also forms the basis of most access control in the system.

Thread state information can be queried and set along with scheduler parameters using the *thread_schedule* system call. Access control on this call is based on another property of a thread: the *maximum controlled priority* (MCP), which is set at task creation time and controls the maximum priority that a thread may give itself or others. This call is effective only on threads with an MCP value less than or equal to the caller's. It can be used to set the priority and timeslice length of a thread, and also access thread state information.

L4 provides a 256-level hard priority scheduler, with round-robin scheduling within each level, and variable time-slice lengths. An example ready queue is shown in Figure 4.2. A more complex user-level scheduler can be implemented in L4 as a thread running at a higher priority than the threads it schedules (so it is always executed by L4), which uses the *thread_switch* system call to donate its time-slice to the thread to be scheduled. Winwood [Win00] provides a good discussion of the various scheduling issues under L4.

Version 4 API

An update to the L4 API, version 4, has been specified [L4K01]. A cross-platform implementation of this interface, known as Pistachio, is in progress. Mungi is in the process of being ported to Pistachio (as of this writing, a preliminary version boots on the IA-64 architecture), so any implementation must consider portability to the version 4 API.

The primary changes in version 4 which affect a thread model are:

- Resource control has been changed. Under the new model, the first task started on the system is privileged, in the sense that only it can allocate new tasks and threads. This is an improvement over the first-come first-served model of version 2, but places several restrictions on the design of an L4-based operating system, as will be discussed in Section 8.2.3.
- There is no longer a fixed number of threads and tasks—thread identifiers are under the full control of the privileged task.
- Access control over thread scheduling parameters is not based on an MCP value. Rather, each thread has a scheduler thread, set by the privileged task, which may control its scheduling parameters.

Conclusions

Obviously, one of the key considerations for a Mungi thread model is its ability to be implemented on top of L4, which aims to provide the minimal kernel abstractions required to implement arbitrary systems at higher levels. A Mungi design should be easily prototyped using L4, but should not necessarily mirror the L4 design, which is inconvenient for general application programming. The fixed number of threads and tasks may pose an implementation problem, since Mungi aims to support distribution, and an arbitrary small limit on the number of threads would be prohibitive.

4.3.2. EROS

The Extremely Reliable Operating System (EROS) [Sha98, SSF99, Sha99] is a persistent capability system based on some of the original ideas developed in the now-defunct KeyKOS system [Har85]. Unlike Mungi, it is a multiple-address-space OS, and uses segregated capabilities (meaning that user code is not allowed direct access to its capabilities) rather than passwords.

EROS provides the abstraction of a *process* to application code along with two kinds of capabilities:

Process capabilities: provide operations to manipulate a process;

Entry capabilities: allow the holder to invoke the services provided by a particular process.

Unlike other systems, EROS doesn't separate the concepts of thread and process. Every process has exactly one context of execution, and although multiple processes can share an address space by referencing the same page table structure (which is used for implementation of multi-threaded services), they all hold distinct sets of capabilities.

Although EROS is a MASOS, there are some points of interest to a Mungi design. In particular, the use of capabilities to control access to process manipulation operations is a useful and flexible design. The use of entry capabilities seems to be closer to that of PDX capabilities in the Mungi model.

4.3.3. Grasshopper

Grasshopper [DdBF+94] is a capability-based persistent system developed primarily at the Universities of Sydney and St Andrews (Scotland). Grasshopper uses the concepts of *containers* and *loci*. Containers are address spaces (Grasshopper is not a SASOS, although it uses several SASOS-like concepts), and like L4, recursive mappings can be established between them. Loci are contexts of execution (i.e. threads), they execute within a container, and can change containers through registered invocation points (similar to Mungi PDX calls, although invocations don't necessarily return).

Grasshopper uses segregated capabilities to provide protection on the facilities used by containers and loci. Sadly, Grasshopper has no published APIs for the locus model, and the project is now dead.

4.3.4. K42

K42 [IBM01a, IBM01b] is a microkernel-based system being developed by IBM for large shared-memory multiprocessor systems. This emphasis on scalability is of interest here, because Mungi also aims to support large clustered systems in the future.

In K42 the kernel-level unit of execution is the *dispatcher*. Each process consists of one or more dispatchers which are scheduled by the kernel and themselves run user-level threads. Each dispatcher runs in a *resource domain*, which holds the rights to the system's CPU resources. Each resource domain belongs to one of five resource classes scheduled by the system:

1. system and hard-real-time
2. gang-scheduled
3. soft-real-time
4. general-purpose
5. background

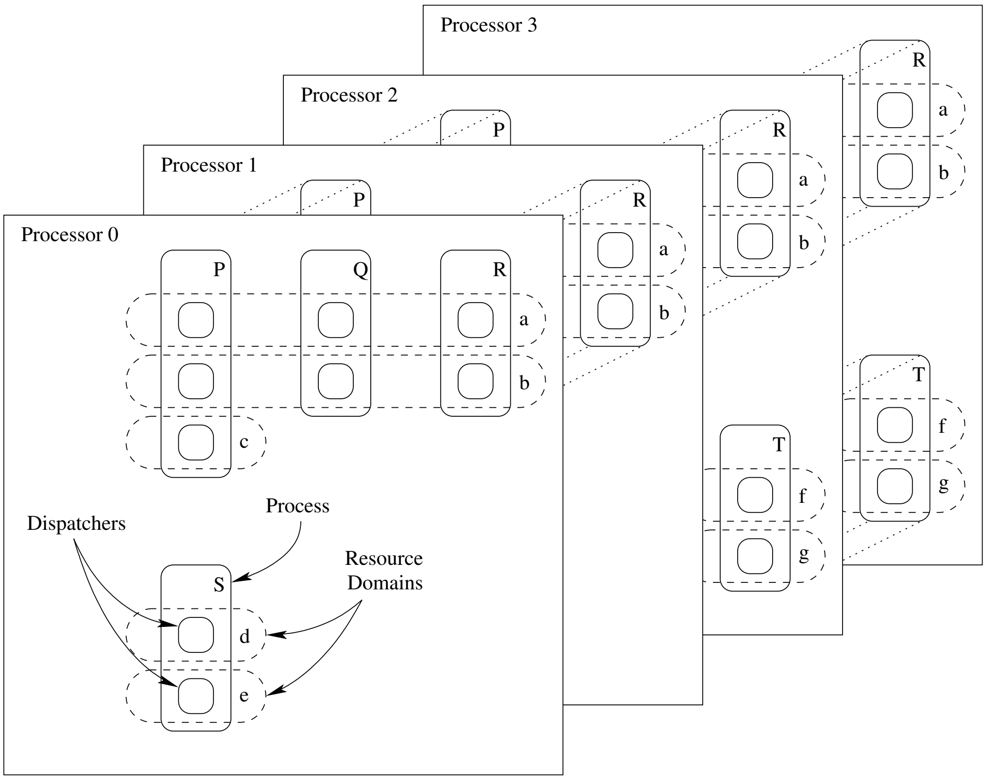

Figure 4.3: Processes, resource domains, and dispatchers in K42. Source: [IBM01b].

Processes may use more than one dispatcher to take advantage of multiple processors, or to run in different resource domains. Within a resource domain, the dispatchers are run in round-robin fashion. The arrangement of processes, resource domains and dispatchers is illustrated in Figure 4.3.

The K42 system uses a number of techniques to minimise problems arising from the use of user-level threads within a dispatcher. Dispatchers are given the opportunity to perform their own preemption, and avoid saving unnecessary state. They can also be signalled to handle kernel events, such as exceptions, or a thread becoming blocked. Dispatchers are very similar to the scheduler activation design used in Angel, Opal and Nemesis.

At the user-level, native K42 and POSIX threads interfaces are provided.

4.4. Conclusions

It can be seen that beyond accepted standards such as Pthreads, the variety in threading models supported by existing systems is great. In many cases, it is difficult to find details for the user-level API provided by these systems, although some useful points have been noted.

Of interest is the large number of systems that rely on, or provide explicit support for, user-level threading (Solaris, Angel, Opal, Nemesis and K42 are all examples of this approach), usually through scheduler activations. Results cited suggest that such a design can offer improved performance over pure kernel-level threads.

However, since Mungi itself is already running on a microkernel system which provides lightweight threads, the performance benefits from a scheduler activations design would be less significant. Furthermore, due to the design of L4, the overhead of an implementation of scheduler activations on top of native L4 threads would most likely outweigh the performance gains made by moving thread management to user level—other work attempting to implement scheduler activations on L4 has reached similar conclusions [Sch02].

For these reasons, a reliance on user-level support for threading, through a mechanism such as scheduler activations, is ill-suited to Mungi. Furthermore, implementing user-level threads securely would require separate kernel-level threads for each protection domain, and since threads in Mungi change protection domains often through the use of PDX calls, this would lead to a complex implementation and reduced performance. Logically, providing a 1-on-1 relationship between kernel- and user-level threads results in clean orthogonality to the SASOS.

Hence, we conclude that fully kernel-scheduled threads are the right model for Mungi. Of course, regardless of the final design, applications could still use a user-level threads package if desired, but they would encounter the traditional problems of pure user-level threads (discussed in Section 2.1.2)—for some classes of application this is an acceptable tradeoff.

5. Design

5.1. Design decisions

5.1.1. Capabilities for protection

As discussed in Section 3.2.1, which also considers solutions to this issue, the single biggest problem with the current Mungi thread model is the inflexible and overly restrictive hierarchical access control model it enforces. Since Mungi is a capability system, and every other part of Mungi uses password capabilities as the basis of protection, it makes sense to also use some form of capability to control access to thread operations rather than a (fairly arbitrary) hierarchy. This mirrors the design decisions made by Windows NT and EROS (described in Sections 4.1.3 and 4.3.2 respectively) to use the same access control mechanism as the rest of the system in controlling access to thread operations.

5.1.2. User-level TCB

Having decided to use capabilities as the basis of protection, the next issue to be resolved is what form these capabilities should take. It would be possible to add a new type of *thread capability* to Mungi with a different set of permission types (for example, query parameters, set parameters, sleep/resume, delete, etc.). Alternatively, Mungi's regular capabilities, which control access to memory objects, can be used.

Deciding between these two options leads us to an essential principle of Mungi, which was designed to present the SASOS concept in a very pure form. Mungi has in the past avoided typed capabilities (where capabilities exist that convey different permissions to different types of object), since it views everything as an untyped memory object in the SAS. This avoidance of typed capabilities leads us to choose to normal object capabilities for controlling access to thread operations.

Since regular object capabilities will be used, each thread must somehow be associated with an address in virtual memory. This poses two alternatives: either the identifying address for each thread could store some data related to the thread, or the address could be purely an identifier with no relevance to the data in memory. The latter option is similar to the design of semaphores in Mungi—each semaphore is identified by a virtual address, and protection for semaphores is based on capabilities to the object containing the semaphore's address, but there is no direct correlation between a semaphore and the data stored at its identifying address.

A decision was made to take the first option, since storing thread-related data at the identifying address of each thread would be convenient for the user, replacing the role formerly filled by the *ThreadInfo* system call. Thus, the idea of a *user-level thread control block* (TCB) was born. User-level TCBs reflect the thread's state, but the design could also be extended to allow the setting of thread parameters by writing to TCBs—this has several potential ramifications, which must now be considered.

Updates to user-level TCBs

If user-level TCBs are to be both readable and writable, with certain thread properties set by writing to a thread's TCB, several issues arise. The primary one is how the Mungi kernel finds out about changed parameters in order for them to take effect. Several methods were considered:

- The kernel could check TCB fields lazily. In particular, fields that are only relevant in the context of an event (for example, the thread performing a certain system call) could be read at that time and their values used. Since not all thread properties fall into this category, lazy updates don't address the whole problem.
- The kernel could actively check for changes. This would require the kernel to contain one or more threads that polled the TCBs of all threads in the system, compared the values with kernel-saved copies, and took action if a field was found to have changed. Such a design was rejected, as it would require continuous processor overhead on the part of the kernel, require the kernel to keep a copy of TCBs in order to check for updates, and allow a period of time between a value being changed and the change taking effect. This design also leads to problems with users needing to atomically update fields in a TCB, and would require some sort of lock on each TCB that is obeyed by the kernel (i.e. updates do not take place while the lock is held).
- The system's page-fault handler could be modified to notify the kernel when a TCB is written to. This would require extensive modifications to the Mungi paging system, require TCBs to be mostly mapped read-only, and yet still incur the same problems with atomic updates. It was rejected mainly due to complexity reasons.
- Each TCB could have associated with it a special semaphore, and when this semaphore was signalled, the kernel would check the TCB for any updates. Such a design could work, but was rejected for several reasons. Logically, it would require a thread blocked in the kernel on each TCB semaphore to check for updates—obviously an implementation would not do this, but it was considered unclean and not in line with the general Mungi design principle of avoiding unnecessary server threads.

- A system call could be used to explicitly notify the kernel of TCB changes. One possibility would be to use the existing *PageFlush* system call, which is part of Mungi's page fault handler mechanism—a *PageFlush* call specifying a page containing TCBs would cause the kernel to update changed values in any of those TCBs. Two problems with this are that TCBs are reasonably small (many can fit in a single page, resulting in much unnecessary work on every call), and that it overloads the semantics of a system call with previously straightforward behaviour.

  To address these problems, it would be possible to have a special system call purely for notifying the kernel of TCB updates. This avoids many of the previously mentioned problems, but is very similar to *ThreadInfo* in the old model, and still requires the kernel to keep a copy of each TCB to check for changed fields.

These methods were all considered, but each had certain problems making it unsuitable for use in Mungi. In the end, a decision was reached to use the lazy method as much as possible, and for changes to TCB fields which must take immediate effect, to use a special system call for that purpose. In the current design (see Section 5.2), the only system call that falls into this category is *ThreadSetPriority*.

User-level TCBs and MAC

As discussed in Section 2.2.5, Mungi features a mandatory access control (MAC) model, which allows the enforcement of system-wide security policies. MAC requires the association of a domain label with each active entity (subject) in the system, and a type label with each object that can be acted upon. With the previous thread model, this resulted in both domain and type labels being kept for threads (since threads perform actions, but can also be acted upon).

User-level TCBs simplify the MAC design, however, since access control on threads is now based on capabilities to the object containing the thread's TCB. This object already has a label, which is validated as part of the regular MAC system for controlling object accesses, so it is no longer necessary to provide a separate type label for thread operations. This has the secondary benefit of simplifying the implementation of the thread-related system calls, which will no longer require explicit MAC validations.

Other issues with user-level TCBs

Other issues that were considered in relation to user-level TCBs involved the need for finer-grained protection, and the problems of binary incompatibility.

Conceivably, users may wish to allow others access to only read or write certain parts of TCBs. For example, since a future user-level scheduler may need to set threads' priority levels and CPU time limits, it would be beneficial to general security if one did not also need to give it permission to change or view other TCB values. There are two ways to address this problem: one is for Mungi to provide finer-grained protection in the form of more capability types, the other is to leave the kernel unchanged, but allow users to set up PDX services which access TCBs in a restricted manner. The latter can be supported without further modifications to the Mungi security model, and it is closer to the general Mungi principle of providing protection at page or object granularity, so it has been adopted.

The other problem user-level TCBs raise is that of potential incompatibilities in the future. This problem is not unique to TCBs (it affects all kernel-exported data structures), but it is particularly important since TCB accesses will be fairly common in user-level Mungi code. Any system needs room for expansion, and Mungi TCBs are no exception—in the future it can be expected that fields will be added, and the behaviour of TCBs changed. If user-level code makes assumptions about the contents of TCBs, this would be a nightmare to upgrade. The solution adopted to this problem is not to change the design of TCBs, but to avoid such a situation occurring—the thread model being designed is to form part of the Mungi kernel-level API, and it is not expected that user programs will use this interface directly. Rather, a convenience library with a function call interface should be provided. If TCBs were to change, this library code could absorb and hide the changes from user programs.

5.1.3. Thread death notification

Any thread model requires a method for notifying users about the death or deletion of threads. For example, a shell thread may spawn a new thread to execute a command, and needs to wait for it to exit (or die) before displaying the prompt and waiting for further user input. In many traditional systems, and in the current Mungi thread model, this is provided by some form of *wait* or *join* system call, which blocks until one or more threads exit, and then returns some information about this (usually the thread's exit status, which is an integer value indicating success or failure) to the caller.

Three alternatives for thread death notification were considered: a system call, a special semaphore, or a PDX callback function.

ThreadWait system call

This is basically the method used in the current Mungi thread model. A system call is provided which blocks until the specified thread exits, at which point it returns the thread's ID and exit status to the caller. It is also possible, under the old model, to wait for any of the children of the calling thread.

The first part (waiting for a specific thread) would be easy enough to do in the new model, but the second is much harder. Since there is no longer a thread hierarchy (the major problem identified in the current design), it is not possible to wait for all children of the caller. Instead, some mechanism would be required for specifying a set of threads to wait for.

Solutions considered include a variable length array of TCB pointers, or a call which waits for all threads with TCBs inside a given object or memory range. Both of these solutions were rejected as unsatisfactory, since they would require the kernel to keep lists of waiting threads, and either introduce a system call with an inelegant interface, or require applications to create all their threads inside the same object or memory range.

Thread death semaphore

Since thread death notification is essentially a form of synchronisation, another possibility would be to associate a semaphore (Mungi's synchronisation primitive, introduced in Section 2.2.8) with each TCB, which is signalled by the kernel when the associated thread exits.

One of the problems with using semaphores is that extensions to Mungi's semaphore model would be required. Currently, *SemWait* requires a read capability to the object containing the semaphore. In the case of TCBs, this would require users wishing to be notified of thread death to also be able to view the thread's entire TCB, which is arguably an undesirable impact on security and violates the principle of least privilege. Furthermore, a standard semaphore wakes only one waiter when it is signalled, but for thread death notification all waiters should be woken—such an option is available in the current API[1], but would need to be implemented.

These problems can be addressed, but two others are less trivially solved: collection of a thread's exit status, and waiting for multiple threads.

Other mechanisms for thread death notification deliver the thread's exit status along with the notification. Semaphores, however, do not deliver any data with the synchronisation event, so some other mechanism is required for accessing the exit status—the logical choice is to write it into the thread's TCB. However, this would again require read permissions to the TCB, as well as a complex user-level protocol (involving further semaphores) for the thread's owner to know when all waiters had collected the exit status, so the TCB could be reused.

Waiting on multiple threads is also a problem. Extending the semaphore model with a system call to wait on multiple semaphores (similar to the UNIX *select* call) was considered, but rejected as being unscalable and unsuited to a distributed Mungi system. It is possible to emulate such a service at user level, but with the significant overhead of one blocked thread per semaphore.

Thread death PDX callback

The final option considered was a callback mechanism. This would involve the kernel performing a PDX call on behalf of the deleted thread to one or more callback functions registered for the thread. The PDX call could pass as arguments the thread's TCB pointer and exit status, delivering the notification and related information at the same time.

One of the main differences between the PDX callback and the other methods is that is doesn't require a thread to be blocked waiting for notification. If a thread wishes to be

[1] The *wake_all* flag to the *SemCreate* system call causes all threads waiting for the semaphore to be unblocked when the semaphore is signalled.

blocked (for example, in a POSIX-compatibility library emulating the *wait* system call) this is still possible, by having the PDX callback signal a semaphore for which the thread waits, but threads don't block unnecessarily.

In many cases, the reason thread death notification is required is so that some cleanup actions can be performed when the thread exits (for example, releasing resources held by the thread). These actions can be performed directly by the PDX code, which only exists as an active thread when needed, thereby saving system resources. Furthermore, such a design offers benefits for distribution. Under a blocking model, the creator thread must wait for the death of its children threads in order to perform these cleanup functions. If the dead thread had been running on a remote node, this would always require communication with the node running the waiting thread, introducing often unnecessary delays. Under the callback model, the cleanup code can execute immediately on the same node as the dead thread, allowing for higher overall system utilisation.

Of course, there are some questions raised by a PDX callback model: how are callbacks registered, does a single thread death cause one or many callbacks, and in what environment does a callback PDX execute?

Two alternatives are available for registering callback entry points: they can be set by writing to the thread's user-level TCB, or via a dedicated system call. The former seems the natural choice, but since variable-length TCBs would raise many difficult problems with thread management, each TCB can only be designed to store a small, fixed number of callbacks. The system call approach is more heavyweight, and requires the kernel to maintain callback lists, but would allow an arbitrary number of callbacks per thread. Of course, if multiple callbacks were to be used, it would be necessary to specify in what order they are executed, leading to further system complexity.

Finally, we must consider in what environment a PDX callback executes. For security reasons, any callback PDX registered must be validated before it is invoked, which means that the deleted thread must hold a capability to the callback PDX entry point. This would, however, allow a thread to fake its own death by calling the PDX code itself—to address this problem of trust, callbacks can be executed with a well-known TCB pointer used only for kernel upcalls, which can be verified by the called PDX code. Such a special TCB is also required for other forms of kernel upcall, which do not have a logical calling thread (for example, MAC validations or device driver interrupts).

Conclusions

As we can see, in each of the three mechanisms considered for thread death notification, considerable problems are encountered when multiple notifications must be sent. In order to decide if multiple notifications are really necessary, consider the reasons for thread death notification:

- It is needed to perform cleanup operations, for resources held directly by the thread. This can be handled well with only one notification.

- It may be used to free resources not held directly by the thread, for example by some third party service with which the thread was interacting. This is more of a resource control issue than a thread management issue, but in general such resources are held on behalf of active protection domains, not individual threads—what is really required in these cases is notification of the expiry of an APD.

For these reasons it was decided that the Mungi kernel itself need only provide one death notification per thread, and that multiple notifications, if needed, can be provided by user-level code executing in the callback function. The PDX callback model is used, because it avoids blocking threads unnecessarily, and offers better performance for future distributed versions of Mungi.

5.1.4. Inter-domain *ThreadCreate*

Within the Mungi API, there are two methods for creating or changing protection domains: creating a thread in a different PD, or performing a PDX call. The PDX call is Mungi's basic secure procedure invocation mechanism, but is the inter-domain *ThreadCreate* really required? Both methods have their differences:

- A PDX call extends the caller's APD by only one Clist, whereas *ThreadCreate* allows specification of an arbitrary PD (but does not allow implicit extension of the creator's APD).
- PDX entry points must be registered with the Mungi kernel before use, and require extra capabilities, whereas inter-domain *ThreadCreate* allows the specification of a PD object at creation time. This PD object must obviously also be created beforehand, but can be changed arbitrarily after creation.
- The PDX call is synchronous (logically, the one thread transfers protection domains)—there is no execution in the original APD while the call occurs. *ThreadCreate*, on the other hand, is asynchronous—a new thread is created in a different APD, and executes concurrently with its creator.

Changing the API to allow *ThreadCreate* only within the caller's APD would be a significant simplification of the model, would allow a much simpler kernel implementation, and is arguably a good idea. Nevertheless, it was decided not to make such a radical change at this point, for the following reasons:

- Forcing the use of a PDX call to join a remote APD is less convenient, for the programmer, than simply creating a thread with a given PD object.
- The use of PD objects makes protection domains more explicit, arguably resulting in a cleaner security model.

- In many cases, if the user is starting an application in another APD, they don't need to execute in the current APD before the PDX call. Creating a local thread which then performs a PDX call would need space allocated for a local stack, furthermore, the completed PDX call would simply return to the local thread, which would directly exit. This is undesirable and unnecessary overhead.

Previous work [Cut98] has also considered this question and failed to reach a resolution—clearly this is an area which needs further investigation. Possibly what is needed is an asynchronous PDX call primitive (a system call which logically performs both *ThreadCreate* and *PdxCall*).

5.1.5. Resource control

Something which the current thread hierarchy attempts to address is the problem of resource control. Resource control relates to the problem of accounting for use of a system's resources, namely:

Virtual address space: This resource is managed through the existing model of bank accounts, described in Section 2.2.6.

Physical memory: An approach to managing the use of physical memory has been designed, which extends the concept of bank accounts with hierarchical resource groups [Che01, CH02]. This design has not been implemented in the current Mungi system, and since it relies on a hierarchical thread model, some work will be needed to adapt it to the design presented by this thesis, but it is an approach to solving the problem.

CPU time: This is the main resource with which we are concerned. It affects the allocation of processor time to threads running in the system—that is, the decisions the scheduler takes in selecting which threads will be executed from those that are runnable.

CPU time allocation

A number of methods were considered for controlling the use of CPU resources. In an earlier design, *ThreadCreate* required a thread template special object (special objects are discussed in Section 2.2.7, they have more restricted semantics for creation and access). This allowed restrictions on the number of threads that could be created, and the placing of resource limits on such threads. It was rejected, however, because limiting the number and type of threads created is not an effective way to control CPU utilisation.

What is needed is a model for allocation of CPU time that fits in with Mungi's other resource management mechanisms—some form of time resource group would seem the logical choice, but this thesis does not attempt to address the problem (which is clearly an area for future work).

Current applications do, however, require a means to control proportional CPU allocation. Device driver interrupt handlers, for example, need to take priority over other threads in the system to ensure that interrupts are handled as quickly as possible. To address these needs, and as a temporary measure until a proper CPU resource management model is designed, a mechanism and simple associated policy is provided for setting a thread's priority level. This priority level maps directly to L4's hard priorities, described in Section 4.3.1).

To provide some control over priorities, a policy is imposed whereby thread is not allowed to increase its priority, or to create threads with a higher priority, and all user-level threads are run below an arbitrary priority limit (kernel threads run above it). To provide timely servicing of device driver interrupts, they are created with a priority level slightly higher than the maximum user-level priority.

Threads created during PDX

The final issue of resource control is that of threads created inside a PDX call. The current API says that threads created by PDX code may be terminated at any time after the call returns. As discussed in Section 3.2.2, this restriction is unsatisfactory for many applications, and hence it has been dropped from the new model. The issue of PDX-created threads is really one of resource control, which should be managed by some form of CPU accounting and resource management mechanism, as discussed previously.

5.2. New thread model

This section introduces the specification of a new thread model, which results from the design decisions discussed in the previous section.

5.2.1. User-level TCB

The primary change made to the existing thread model is the introduction of user-level thread control blocks (TCBs). Each thread in the system has an associated TCB, which contains all user-visible properties of the thread and identifies it. The contents of the user-level TCB are shown in Figure 5.1, and described over leaf.

type *UTCB* $\equiv$ {
flags, state, start_time, cpu_time, cpu_limit,
priority, delete_callback, bank_account,
localnode_id, userflags, userinfo
}

Figure 5.1: Contents of the user-level TCB data structure.

Flags: Contains flags controlling the creation (and possibly operation) of the thread. Currently only one flag is specified: the *join* flag, which controls whether, when the thread is created in a protection domain different from the creator's, it can join an existing APD instantiated from the same protection domain object, or later be joined by other threads created in the same protection domain.

Thread state: This value is one of the following symbolic constants:

> **Unused:** Indicates that the TCB entry is free, and does not refer to an active thread.
>
> **Active:** The associated thread is running (or ready to run).
>
> **Sleeping:** The thread is sleeping, due to a *ThreadSleep* call.
>
> **Blocked:** The thread is blocked on a semaphore, due to a *SemWait* call.

The thread state value is set by the kernel when the thread changes state, writes by a user are ignored.

Start time: Time that the thread was started, user writes are ignored.

CPU time used: Total CPU time used by the thread since it was started, user writes are ignored.

CPU time limit: CPU time limit for the thread, this value is checked when accounting for the thread's CPU time to determine if it has been exceeded. This should be expected to change with an improved scheduling and resource management model.

Priority level: This reflects the thread's L4 priority level, writes by a user are ignored. This should also be expected to change with an improved resource management model.

Delete callback: Contains the address of a PDX entry point that is invoked when the thread is deleted. This forms the mechanism for thread-death notification, and is described along with *ThreadDelete* below.

Bank account: Pointer to thread's default bank account, used by certain system calls when required.

Local node ID: A Mungi implementation-specific field intended mainly for use by user-level library code,. In the current L4-based prototype, this stores the L4 thread ID of the Mungi thread. Section 6.3.2 discusses its use.

User data: Two word-sized fields for storing user-specific data, unused by the kernel.

Holding a read capability to a TCB object allows a user to access the status information stored there. Holding a write capability to a TCB object allows a user to set certain thread parameters by writing to the TCBs. Thread parameters that require action by the kernel to take effect (such as sleeping a thread, or setting its priority) are set by system calls and listed below. Parameters that are only used in response to certain events can be set directly in the

TCB by a user, although the current value of both types of parameter is reflected in the TCB. Reads from, and writes to the TCB are equivalent to calling *ThreadInfo* to access such thread information under the old model.

It often makes sense to apply the same restrictions to an entire group of threads (for example, threads operating within an APD, since they share the same protection domain and can modify each other's data). To take advantage of this property for performance reasons, multiple TCBs can be placed in a single object, with capabilities to this object applying to all TCBs held by it. The thread state value of each TCB in an object can be used to determine if the corresponding thread is in use.

5.2.2. System calls

The new system call interface for dealing with threads is shown in Table 5.1. The calls are detailed below:

System calls:		
ThreadCreate	(*ip*, *stack*, *param*, *tcb*, *pd*)	
	fails: *protection_violation*, *invalid_thread*, *table_overflow*, *invalid_info*, *invalid_Clist*	
ThreadDelete	(*tcb*, *status*)	
	fails: *protection_violation*, *invalid_thread*	
ThreadSleep	(*tcb*, *time*)	
	fails: *protection_violation*, *invalid_thread*	
ThreadResume	(*tcb*)	
	fails: *protection_violation*, *invalid_thread*	
ThreadSetPriority	(*tcb*, *priority*)	
	fails: *protection_violation*, *invalid_thread*	
ThreadMyID	()	→ (*tcb*)
	never fails	
Thread deletion callback function signature:		
Callback	(*tcb*, *status*)	

Table 5.1: System calls dealing with threads in the new model.

ThreadCreate: Creates a new thread, which starts execution at address *ip* with stack pointer *stack* and a parameter *param*. If *pd* is non-null, it must be a protection domain special object to which the caller holds an execute capability.

The caller must pass in the *tcb* parameter a pointer to a location (to which it must hold a write capability) for the new thread's TCB. The call will fail with an *invalid_thread* error if the selected *tcb* is invalid, or already references a running thread.

The *tcb* also specifies initial values of all of the fields that it contains, as well as flags controlling thread creation. The thread may be created with the *join* flag set, which controls whether it can be created in an existing APD, and behaves identically to the current model. No other flags are specified at this point.

ThreadDelete: Kills a thread, and writes the *status* value into a field in the TCB. Can be used on the caller, or a thread to which the caller has a destroy capability.

Upon deletion of a thread, the kernel invokes a call to the PDX entry point registered in the thread's TCB, with an argument containing the TCB address and exit status value. This call executes with a well-known kernel TCB pointer (for reasons discussed in Section 5.1.3), and allows for thread death notification.

Since a destroy capability is required to call *ThreadDelete*, it is possible for threads to use *ObjDelete* to delete the entire TCB object. This is specified as equivalent to deleting all the threads held by the object, and the thread death callback functions will be invoked, but with a status parameter of -1 (since none can be supplied by the user performing the deletion).

ThreadSleep: Stops a thread for a specified period of time, or until explicitly resumed. May be used on the caller, or on a thread to which the caller holds a write capability.

ThreadResume: Resumes a sleeping thread. May be used on a thread to which the caller holds a write capability.

ThreadSetPriority: Sets a thread's L4 priority level. This is a temporary measure for some applications which require a raised priority level, and should change with a proper scheduling system. Requires write capability to the thread's TCB.

Threads may only be set to a priority level equal to or lower than the caller's priority level.

ThreadMyID: Returns a pointer to the caller's TCB. Note that the caller need not have any capabilities to this TCB object, and hence may not have access to their TCB.

5.2.3. Access control

A summary of the capabilities required for the various operations is shown in Table 5.2.

The access control model used is generally based on existing Mungi conventions. When creating a thread with *ThreadCreate*, execute capability is required to the protection domain special object, as is normal for this call. A write capability is required to the TCB object.

ThreadDelete is analogous to deleting the TCB object, and hence a destroy capability is the natural requirement in that case. Regardless of capabilities held, a thread can always delete itself (logically this is what happens when the thread faults or exits).

System calls that modify a thread, such as *ThreadSleep*, *ThreadResume* and *ThreadSetPriority*, require a write capability, with the exception that a thread should be always able to put

Operation:	Requires:
ThreadCreate	execute capability to *pd*,
	write capability to *tcb*
ThreadDelete	destroy capability to *tcb*, or call on self
ThreadSleep	write capability to *tcb*, or call on self
ThreadResume	write capability to *tcb*
ThreadSetPriority	write capability to *tcb*
ThreadMyID	none

Table 5.2: Summary of capabilities required for thread operations under the new model.

itself to sleep. *ThreadMyID* has no restrictions, although there is no guarantee that the TCB pointer returned will be accessible to the caller.

Finally, as discussed in Section 5.1.2, all thread operations are also subject to Mungi's mandatory access control system. Each thread has a domain label, determined by the APD to which it belongs, and a type label, determined by the object which contains its user-level TCB. The MAC policy object validation handler will be invoked as usual for all thread operations, with the operations to be validated being the same as the regular capability permissions described above. This is possible without any extra complexity as a convenient side-effect of representing threads as normal memory objects, and means that MAC validations are required for system calls such as *ThreadDelete*, *ThreadSleep*, and *ThreadResume*—a feature lacking from the previous design.

6. Implementation

This chapter details an implementation of the thread model designed in the previous chapter. The design was implemented within the context of Mungi running on L4/Alpha, since this was the most stable and actively developed version of Mungi at the time. However, as will be discussed in Section 6.6, care was taken to ensure portability to other versions of L4.

When beginning work on the implementation, a decision had to be made regarding reuse of code from the existing Mungi version. Since the previous thread implementation was closely tied to the thread hierarchy, and was written in a unstructured manner, very little code was able to be reused. Furthermore, inside the kernel, the functions of thread management were split up from a single file into a number of modules relating to specific parts of the implementation. This modularisation will be discussed further in Section 6.5.2.

The implementation discussed in this chapter is available in the DiSy group's CVS repository, with the tag *Thesis_2002s2*. It could be checked out, for example, with the command:

```
cvs -d /home/disy/cvs_root co -P -r Thesis_2002s2 Mungi-1.2
```

6.1. Structure of the Mungi implementation

Mungi is implemented on top of the L4 microkernel, which, as discussed in Section 4.3.1, provides *thread* and *task* abstractions. The implementation uses one L4 thread for each Mungi thread, and one or more L4 tasks for each Mungi APD. This arrangement is illustrated in Figure 6.1, and described in more detail below.

The single address space (SAS), within which the Mungi kernel and user threads all execute, is provided by another lower-level task, the *RAM pager*, which is irrelevant to the thread implementation and will not be discussed further. The Mungi kernel task has full access to the SAS, and allows user tasks access to a restricted subset of the space, determined by the capability-based protection mechanism which it enforces (and which is described in Section 2.2).

The Mungi kernel consists of several threads:

- The system call handler thread, which receives L4 inter-process communication (IPC) messages from user threads, performs system calls on their behalf, and responds.

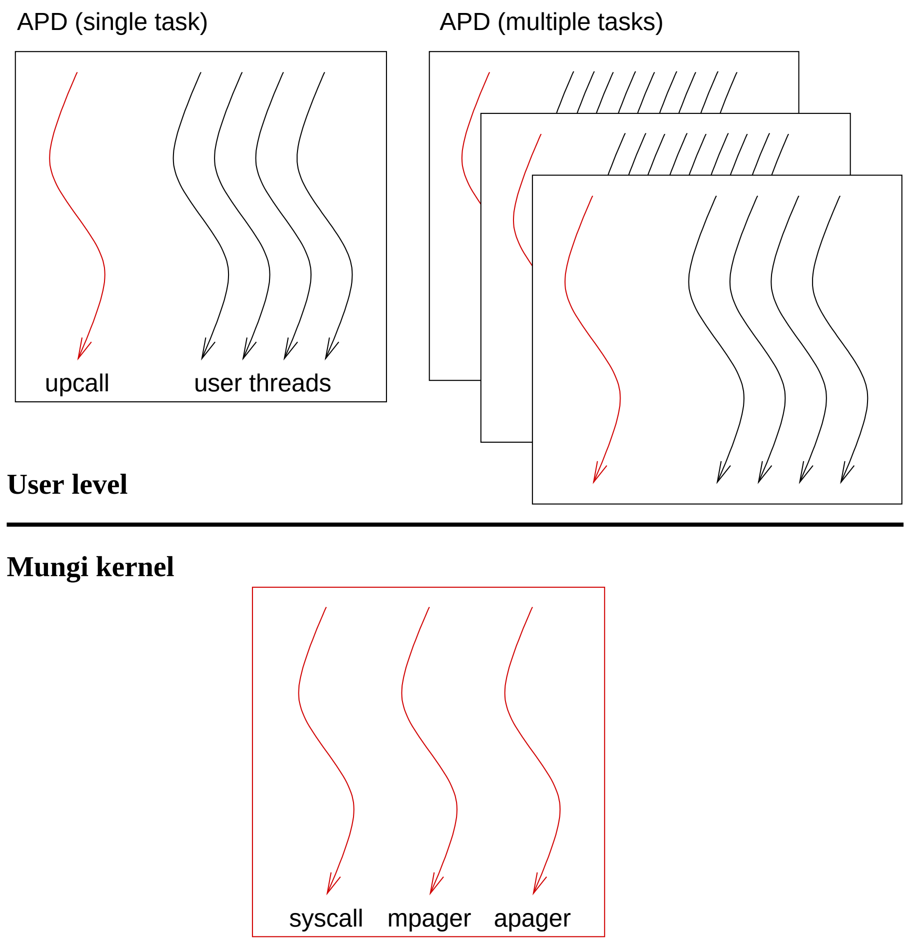

Figure 6.1: Structure of L4 tasks and threads used in Mungi.

- The pager threads *mpager*, and *apager*, which control paging of user tasks, and also access the kernel's data structures.
- A number of interrupt handler threads, not shown in the diagram, needed for device drivers.

At user level, each APD is represented by one or more L4 tasks. Several tasks may be needed for a single APD, since L4 supports a fixed, limited number of threads in each task[1]—this is a restriction specific to the version 2 API, which will be obsoleted by the next version of L4. Furthermore, L4 restricts the *lthread_ex_regs* system call, which is used to start or modify threads, to use only on threads within the same task as the caller—this requires Mungi to make use of an *upcall* thread within each task for controlling the other threads. As depicted, the upcall thread is the first thread (with a local thread ID of zero) in each task.

6.2. Walk through of a thread creation

To provide an overview of the implementation, and to introduce the various components involved, this section discusses the procedures that take place when creating a new thread in Mungi. For reference, the whole process is illustrated graphically by Figures 6.4 and 6.5.

6.2.1. Preparation

Before any thread creation can take place, the user must create and set up a stack and TCB. If the thread will be created in another APD, a PD object which specifies the thread's protection domain must also be provided.

The declaration of the user-level TCB and related types is shown in Figure 6.2. This structure is taken directly from the fields described in Section 5.2.1. In most cases, the user would simply supply a zero-filled TCB to the *ThreadCreate* call, however if they wanted to apply process limits, set flags affecting the thread's creation, or register a delete notification callback function for the thread, then they would need to set some of these fields as required.

The work required before *ThreadCreate* can be called is fairly significant, especially if the user wishes to make use of the full potential of the new thread model, by placing multiple TCBs and stacks in the same object, and using the thread delete callbacks for exit notification. A reasonably simple convenience library was written to perform these tasks, its interface is described in Appendix B.

6.2.2. System call stub at user-level

Having initialised the thread's TCB, the creator calls the *ThreadCreate* library function to perform the system call.

[1] For L4/Alpha there are 256 threads per task, with L4/MIPS the limit is 128.

As discussed in Section 6.1, Mungi user threads perform system calls by sending L4 IPC messages to the system call server thread in the Mungi kernel. This is implemented in a number of system call wrapper functions—these are fairly straightforward, they simply marshal the arguments into a message buffer, which is then sent to the Mungi kernel using an L4 IPC call. The system call stub in this example, *ThreadCreate*, is shown in Figure 6.3.

6.2.3. Inside the kernel

System call handler

As was depicted in Figure 6.1, the main thread in the Mungi kernel handles system calls from user threads. This code (implemented in kernel/src/syscalls.c) receives each message, determines the calling thread's ID (messages from invalid threads must be ignored), and then passes the message to the relevant system call handler function.

In this case the system call is a *ThreadCreate*, for which the handler function is *sys_thread_create*. This function performs necessary validations for the call, including validating the caller's right to write to the TCB, as well as its right to create threads in the PD, if one was specified.

If the thread is being created in another PD, an APD must be located. The code managing this process (in kernel/src/apd.c) is not strictly part of the thread implementation, however it performs the following tasks:

1. if the user has not specified the *join* flag (for convenience the implementation uses an inverse *no_join* flag), a new APD is instantiated from the given PD object;
2. otherwise, a cache of APDs which were created without the *no_join* flag set is consulted, and if one instantiated from the same PD is found, it is used;
3. if no APD could be found, a new APD is instantiated and added to the cache.

Finally, the *thread_create* function, which performs most of the work of the thread creation, is called. Its first task is to call *l4threads_get_thread* to allocate a thread in the APD.

l4threads_get_thread function

This function is part of the thread and task management code (located in kernel/src/l4threads.c), which will be discussed in further detail in Section 6.5.2. This code maintains a list of L4 tasks allocated to each APD, and tracks the status of threads within those tasks.

First, this function checks the tasks allocated to the APD. If the APD has been newly created, or if all threads in all tasks allocated to the APD are in use (recall that under L4 version 2, each task has a limited number of threads), a new task is created and assigned to the APD.

```
typedef enum threadstate {THREAD_UNUSED = 0, /* this TCB entry is free */
                          THREAD_ACTIVE,     /* thread is ready/running */
                          THREAD_SLEEPING, /* thread is sleeping */
                          THREAD_BLOCKED, /* thread is blocked on a sem */
                         } threadstate_t;

typedef struct utcb {
        int32_t         flags;          /* thread creation flags */
        threadstate_t   state;          /* thread state */
        time_t          start_time;     /* start time */
        time_t          cpu_time;       /* CPU time used */
        time_t          cpu_limit;      /* CPU time limit, 0 for unlimited */
        uint8_t         priority;       /* L4 priority level */
        pdx_t           delete_callback; /* thread delete callback PDX entry */
        void            *bank_account;  /* default bank account */
        l4_threadid_t   localnode_id    /* node-local L4 thread ID */
        uintptr_t       userflags;      /* user-specific data */
        void            *userinfo;      /* user-specific data */
} utcb_t;
```

Figure 6.2: User-level TCB declaration (include/sys/types.h).

```
int ThreadCreate(thread_t ip, void *stack, cap_t param, struct utcb *tcb, const apddesc_t *pd)
{
        syscall_t sysmsg;
        l4_msgdope_t result;

        mungi_lib_init();
        if (pd == NULL && ThreadCreateLocal(ip, stack, param, tcb) == 0)
                return 0;

        sysmsg.syscall.number = SYS_THREAD_CREATE;
        sysmsg.syscall.data.thread.create.ip = ip;
        sysmsg.syscall.data.thread.create.stack = stack;
        sysmsg.syscall.data.thread.create.param = param;
        sysmsg.syscall.data.thread.create.tcb = tcb;
        sysmsg.syscall.data.thread.create.pd = pd;

        safe_l4_ipc_call(mungi_tid, L4_IPC_SHORT_MSG, &sysmsg.msg,
                        L4_IPC_SHORT_MSG, &sysmsg.msg, L4_IPC_NEVER, &result);
        return sysmsg.sysret.retval;
}
```

Figure 6.3: *ThreadCreate* system call stub function, showing the marshaling of parameters into an IPC message to the Mungi kernel (lib/mungi/threads.c).

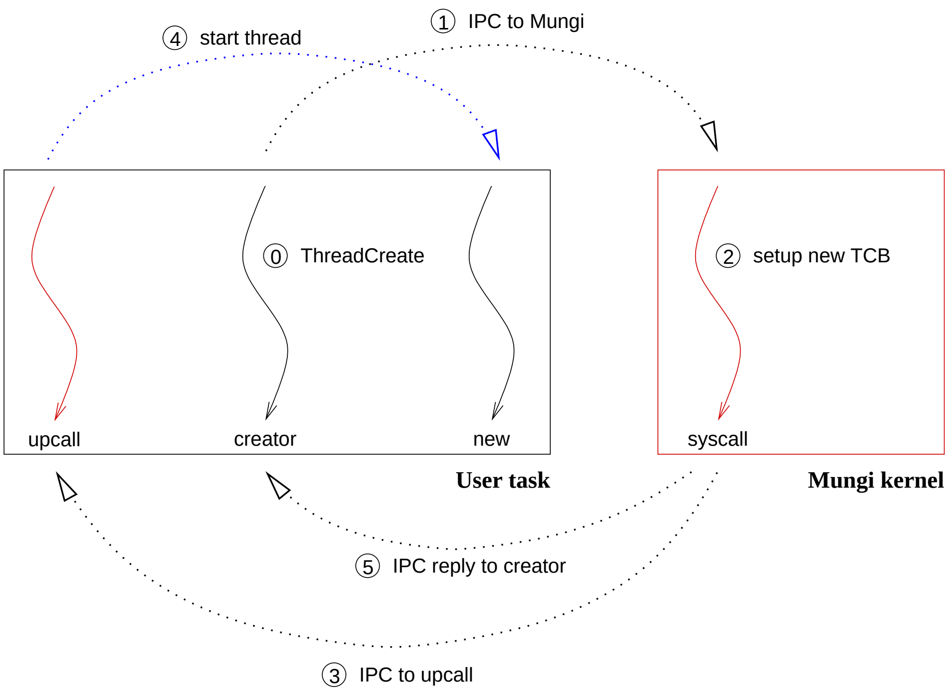

Figure 6.4: Thread creation using upcall to start the new thread.

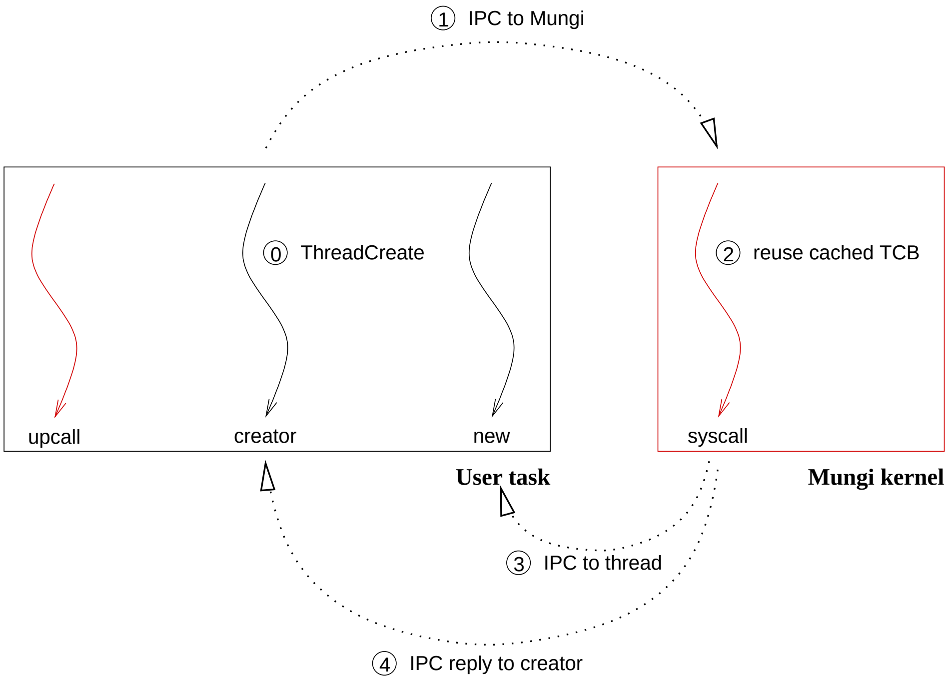

Figure 6.5: Thread creation using a ready, waiting thread within the same task.

This involves allocating a new task management structure, and starting the upcall thread (which handles kernel operations) in the task.

Each thread requires a certain amount of state information to be maintained in the kernel (primarily the thread's TCB). For performance reasons, it is not desirable to allocate and initialise this information on every thread creation, so the thread management code maintains a list of allocated but currently unused threads for each task. If possible, *l4threads_get_thread* provides a thread from this list, otherwise it performs the necessary allocations to create a new TCB.

thread_create function

Having determined which thread will be used for the request, the *thread_create* function performs the following tasks:

1. It initialises fields in the thread's kernel TCB.
2. The thread's priority level is set appropriately, using the *l4threads_setpriority* function. Because setting a thread's priority involves an L4 system call, and to keep thread creation fast, the kernel keeps track of the L4 priority level of allocated threads, and avoids making the call if the priority is unchanged. To prevent threads from bypassing Mungi and setting priorities themselves, they are all started with an MCP value of zero (recall that this forms L4's access control mechanism over the *thread_schedule* call).
3. Entries for the new thread are inserted into two kernel hash tables: a hash from L4 thread IDs to kernel TCB pointers, and a hash from user-level TCBs to kernel-level TCBs.
4. It initialises several fields in the thread's user-level TCB, and sets the thread's state to active.

The thread is now ready to be started. Starting a thread in L4 requires use of the *lthread_ex_regs* system call, which can only be used within the same task. As we have discussed, the upcall thread is used to perform this operation. Hence, to start a thread, the kernel must send an IPC message to the responsible upcall thread, which then starts the actual L4 thread.

Of course, sending an IPC and making an *lthread_ex_regs* call is expensive, and we would like to avoid it if possible. This can actually be done, partially due to the design of L4, which means that once a thread has been created, it can not be deleted. When each thread exits or is stopped through a *ThreadDelete* call, the Mungi kernel returns it to the pool of allocated threads, and makes the thread execute a small piece of code that waits for an IPC message from the kernel. This means that if the thread has already been created, the kernel need not invoke the upcall thread, but can send an IPC message directly to the created thread, containing the information necessary to start execution.

Note that there is a potential denial-of-service problem here, if the waiting thread has been controlled by a malicious thread in the same task, the kernel may block when trying to send the startup message. For this reason, the IPC message to the thread is sent with a small timeout value—if the message does not get sent within this time, the kernel can take alternative action.

So, the kernel determines whether its message should go to the upcall thread or the thread itself, and sends a message consisting of the following:

- the creation type to be performed (thread create or PDX call—we are only discussing thread creation here, but PDX calls are very similar and use the same mechanism);
- the L4 thread ID of the thread to be started (only used by upcall);
- the instruction pointer at which to start execution;
- the stack pointer of the new thread;
- the capability that has been given as an argument;
- the user-level TCB pointer of the new thread.

After sending this message, the *thread_create* function returns to the system call handler, which (assuming all went well) replies to the creator thread indicating the success of the call.

6.2.4. Upcall thread

As has been discussed, an upcall thread (the code for which can be found in kernel/lib/upcall/upcall.c) runs in each task, and executes calls on behalf of the Mungi kernel such as starting new threads. In the case of thread creation, the upcall thread in the destination task is sent a message to start the new thread, if required. Upon receiving this message it uses *lthread_ex_regs* to start the given thread.

Of course, as would be expected, it isn't quite so simple. As well as initial instruction and stack pointers, the new thread must be passed a capability as its argument. A capability occupies two 64-bit words, and due to the C calling convention on the Alpha, these words must be loaded into the first two argument registers of the target thread before jumping to the thread code to be executed. However, the *lthread_ex_regs* call only allows the instruction and stack pointers of the thread to be set directly, not any other registers.

To get around this problem, the arguments, instruction pointer, and user-level TCB pointer are placed on the thread's stack (an operation known as *stacking*), and *lthread_ex_regs* is used to make the thread execute a special piece of assembly code known as the trampoline, which is discussed in the next section.

6.2.5. Trampoline stub code

Creation of a new thread

The trampoline code (located in kernel/lib/upcall/asm/alpha/trampoline.S) is used to load arguments off the stack for the new thread—in fact, the code in this file does this and more, but we shall discuss the unstacking of parameters first. Figure 6.6 shows the placement of the thread's instruction pointer, parameter, and user-level TCB pointer (which is needed for the thread-local storage support, and will be discussed in Section 6.4) on the stack.

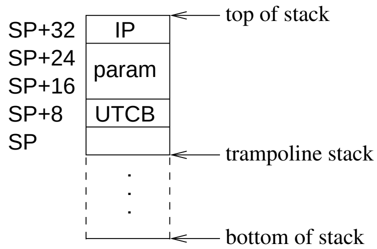

Figure 6.6: Placement of parameters on the stack given to the trampoline code.

The relevant trampoline function, *thread_trampoline* or *pdx_trampoline*, loads these parameters from the stack, restores the stack pointer, sets a return address for the new thread, and jumps to the correct instruction pointer.

Two different trampoline functions are needed for threads and PDX code, because although they both take a capability as an argument, threads return an integer (one word), whereas PDX calls return a capability (two words). Using the correct C calling convention on the Alpha architecture (which has only one return register, unlike the MIPS, which has two), if a function returns more than one word, its first argument must be a pointer to a memory location where the return value can be stored[2]—the PDX trampoline code takes care of setting up this extra parameter where required.

Thread return and restart

As was mentioned previously, the trampoline code sets a return address for the thread or PDX call. This return address is actually to another trampoline function, which takes care of sending an IPC message to the Mungi kernel notifying it of the thread's completion, and waiting to receive an IPC message which starts off a new thread or PDX (recall that upcall only starts each thread once, subsequent invocations are via direct IPC).

Constructing and receiving an IPC call in assembly is non-trivial, so ideally the return function would simply be a small stub which jumped into some C code that performed the actual

[2] In actual fact, the calling convention differs between compilers—versions of the GNU C compiler prior to 3.0 use two registers for the return value.

IPC and restart. However, to execute compiled C code requires a stack, and the only available stack in this case is that of the returning thread, which may no longer be valid after the thread has exited. Hence some assembly code is used, which constructs and sends an IPC message to Mungi, and decodes the response.

This code, which is found in the *thread_return* and *pdx_return* functions, loads only those registers needed for the IPC, and avoids the register stacking required by the regular IPC stubs, thereby saving cycles in the performance critical thread creation and PDX call paths. It also allows arguments to be copied directly into destination registers when starting a thread or PDX, without the need for stacking and unstacking them.

6.3. Optimised local thread creation

6.3.1. Motivation

As was discussed in Section 3.3.1, the performance of thread creation is crucial to some applications, especially device drivers [Les02]. In looking at what can be done to speed up thread creation, we find that the process presented in the previous section can be optimised, but still has some inherent overheads. As illustrated in Figure 6.5, the fastest thread creation involves:

1. an IPC from the creator to the kernel, for the system call;
2. setup operations inside the kernel;
3. an IPC from the kernel to the new thread;
4. a reply IPC from the kernel to the creator.

Of these tasks, only the kernel setup operations can be further optimised without modifications to L4, and, as will be seen in Chapter 7, these are already reasonably fast (although there is always room for improvement). To get real performance benefits, it seems the only possibility is to avoid the kernel invocation.

This is actually possible in some cases—if the thread is being created in the same task (and hence APD) as the creator, and will not cause page faults or make system calls, the creator could simply use the *lthread_ex_regs* primitive to start the thread without the kernel needing to know about its existence. If the new thread faulted or made a system call, however, the kernel would not know about it and would ignore the request (thereby permanently blocking the thread).

Obviously this in itself is unworkable, since it can not be known ahead of time whether a given thread will need to invoke the kernel—what is needed is a way to optimistically create local-only threads, yet still allow the kernel to discover and service requests from these threads. This would defer the in-kernel cost of thread creation until it is actually needed.

The idea of optimistically creating threads at user-level without kernel interaction, while still allowing the kernel to handle such threads correctly, is somewhat similar to the design used by L4's local IPC mechanism [LW01]. In that system, threads sharing a protection domain can communicate via an IPC (requiring a context switch) without kernel intervention, with the kernel still handling correctly threads that have switched in this manner.

6.3.2. Implementation

Shared data structure

In order for the kernel to be able to discover locally created threads when they page fault or make a system call, some level of cooperation is required with the creating thread (i.e. the code inside the *ThreadCreate* stub function). This takes the form of a shared table of pointers to user-level TCBs, allocated for each task, and indexed by L4's local thread ID.

The shared table is used for two purposes: to allow the kernel and user library to cooperate on allocation of threads within the task, and for the kernel to be able to locate the user-level TCB of a locally created thread when it faults. When the kernel creates a thread in the task, it sets the corresponding table entry to -1, notifying the user-level code that the thread is in use by the kernel (nothing is stopping malicious code from using this thread anyway, but since the thread is in the same task, and hence APD, this has no security implications). When the user library creates a local thread, it uses this table to locate a free thread, and sets the entry to the address of the new thread's user-level TCB.

Since this data structure is shared between the user and the kernel, concurrency issues must be taken into account. In particular, to avoid race conditions where both select the same table entry, updates to the table must be atomic. In such a situation, one would normally use a lock value, with threads wishing to read or update the table waiting for the lock to become free. Blocking is not desirable in this case, however, as it would allow a malicious or buggy user thread to obtain and not release the lock, leading to a denial-of-service attack against the Mungi kernel.

To solve this problem of atomic updates without the use of locks, since only one word must be updated at a time, it is possible to use lock-free synchronisation mechanisms [Gre99, Hoh02]. The implementation makes use of lock-free synchronisation through a feature available on most modern processors: instructions which allow the atomic read and update of a single word. On the Alpha, this is provided by the *load linked* and *store conditional* instructions. A table entry can be loaded from memory, compared with zero to see if it is free, and the updated version written—this write instruction only succeeds if the memory has not been modified since it was loaded, allowing atomic updates. The atomic update code for Alpha is shown in Figure 6.7.

```
/* atomically set the given address to the given value
 * only proceeds if the current value is NULL
 * if someone else gets in first, return false, else true */
static inline bool
uthread_update_table(struct utcb **addr, struct utcb *val)
{
        struct utcb *tmp;

        asm("  ldq_l   %1, 0(%2)      \n"
            "  bne     %1, 1f         \n"
            "  stq_c   %0, 0(%2)      \n"
            "  subq    %0, 1, %1      \n"
            "1:                       \n"
            : "+r" (val), "=&r" (tmp)
            : "r" (addr));

        return (tmp == NULL);
}
```

Figure 6.7: Alpha assembly code for atomic updates to shared user-level TCB table (kernel/include/mungi/l4threads.h).

Thread creation

Ideally, whether a thread is created locally or through the kernel should be transparent to the user, apart from the improved performance. To achieve this, the code for local thread creation is implemented inside the system call stubs—the user code calls *ThreadCreate* as before, and if possible, a local thread creation is used (the test for this possibility can be seen in the call to *ThreadCreateLocal* in Figure 6.3 on page 46).

Having determined that a local thread creation can be used, the library code performs several tasks required to start the thread. The first problem is locating the shared table of user-level TCB pointers—this table is created as a Mungi object by the kernel, but the library may not know its address. Of course, once the library knows its address it can be stored in a local static variable for future use, so the means of locating the table is not performance critical. The solution adopted by this implementation is to add another type of system call message—the library sends a query message to the Mungi kernel, and receives a response containing the address of the shared table. For improved performance, if multiple tasks are allocated to the APD, the addresses of as many of these other tables as will fit in the IPC message are also sent. The query message acts like, but isn't, a true system call, since it is not visible to user-level code.

Having located the table, the library must determine which local L4 thread ID can be used. This is simply a matter of locating a free entry in the table. However, to avoid contention between the kernel and user code (both of which create threads), the user code searches backwards through the table, whereas the kernel searches forwards, when locating a free entry. The current implementation uses a pure linear search to locate the free entry, which is acceptable since the table is reasonably small (being limited by the number of threads per task), however this would need to be changed for a Pistachio implementation—an obvious solution would be to store in the table the most recently freed or allocated index, and start searching from that location. If no table entries are free, all threads in the task are in use, and a local thread creation can not be used—in this case the library code performs the usual *ThreadCreate* system call, and the kernel creates the thread in another task,

Having located a free table entry, the library uses the atomic update code (shown in Figure 6.7) to set the entry to the address of the new thread's user-level TCB. If this fails, it must repeat the search to find a new free entry.

The library now knows the L4 thread ID that the new thread will have, it writes this into the *localnode_id* field of the TCB, and sets a flag in the TCB indicating that the thread has been locally created. Finally, it uses L4's *lthread_ex_regs* call to start the thread. This entire process is illustrated in Figure 6.8.

Thread discovery

Of course, creating the thread is only one part of the problem. The next part of the implementation involves the kernel responding to a system call or page fault from a user-created thread. This process is shown in Figure 6.9.

Both the system call and page fault handler functions call *thread_lookup_l4id* to locate the kernel TCB for a client thread. This function looks up the kernel's thread tables, but if the thread is not found it must now also consult the shared table of user-level TCB pointers for each task.

The task ID of a thread can be extracted from its L4 thread ID—the lookup code does this, locates the table for the relevant task, and indexes into it to retrieve the entry for the faulting thread. The kernel must now perform all the usual validations that would occur at thread creation time, including validating the APD's access to create the thread, as well as the validity of the user-level TCB. If any of these validations fail, the request is ignored, otherwise the kernel creates a kernel-level TCB for the thread, inserts it into the various thread tables, and clears the flag indicating local creation in the thread's user-level TCB, before proceeding with the requested operation.

This takes care of page faults or system call messages from the thread itself, but since we aim to make local thread creation transparent, we must also deal with system calls by other threads affecting locally-created threads, for example *ThreadDelete*. In this case the kernel knows the thread's user-level TCB pointer (since this is how users refer to threads), but not its L4 thread ID. This is where the *localnode_id* field of the user-level TCB comes in—it contains the thread's L4 ID, which can be used by the kernel to setup a TCB for the thread.

The code that deals with this can be found in the *thread_lookup_utcb* function, it behaves very similarly, performing the usual validations, creating a kernel TCB for the thread, and clearing the flag in the user-level TCB indicating a locally created thread.

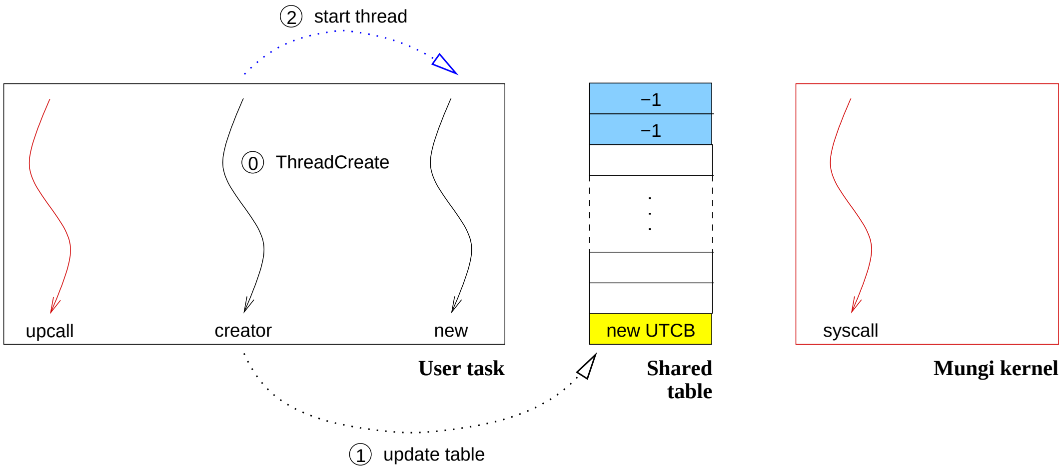

Figure 6.8: Operation of the fast local *ThreadCreate* optimisation.

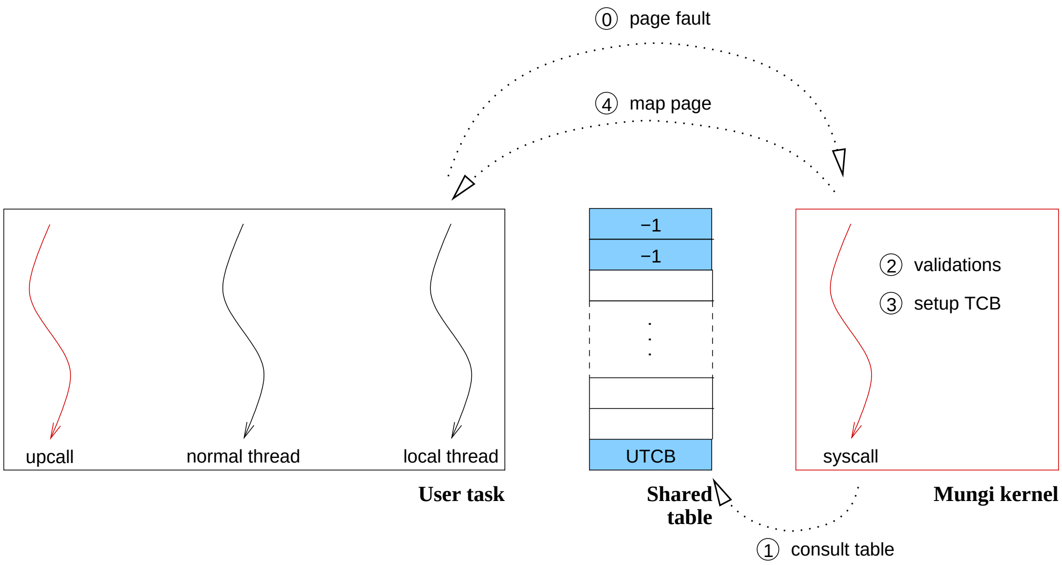

Figure 6.9: Mungi kernel behaviour when a locally created thread faults.

Thread death

When a user-created thread exits, its return address sends it to a cleanup function, which undertakes tasks necessary upon thread exit usually performed by the kernel, such as setting the state field in its TCB to indicate that it is unused.

Firstly, if the thread has a death notification callback function set in its user-level TCB, it must invoke the kernel so that this function is called, alternatively if it does not have a callback function, but the flag in its TCB indicating local thread creation has been cleared, it has been discovered by the kernel, and must also notify the kernel of its death. This notification is sent as a normal *ThreadDelete* system call message, as if the thread had deleted itself.

Relying on threads to notify the kernel of their death poses a problem with threads potentially not sending the notification. However, logically in such a case the thread is not dead, and can still be deleted by another thread (with appropriate capabilities) calling *ThreadDelete*. Furthermore, an analogous situation exists with kernel-created threads.

Finally, if the thread does not need to notify the kernel, it simply removes its entry in the local thread table, and puts itself to sleep.

Persistence and locally created threads

Locally created threads pose a big problem for persistence. As discussed in Section 2.2.9, the Mungi persistence implementation relies on taking a system-wide checkpoint of all threads. This works because the kernel knows about all threads in the system, however if threads are created locally without the kernel's knowledge, they can not be checkpointed, violating the goal of transparent local thread creation.

A solution to this problem would be to have the kernel scan the shared thread tables for new locally created threads on each checkpoint, but this would negate many of the advantages of local thread creation—the only threads to benefit would now be those that were sufficiently short lived to be created and removed between checkpoints. Another possibility would be to allow users control over whether their threads were checkpointed, but this would complicate the persistence model as well as the user programming interface.

Since the persistence model is still under development, and outside the scope of this thesis, a solution can not be provided at this point. Clearly, this is an area which requires further consideration.

6.4. Support for thread-local storage: a thread register

To make threads easier to use and program, it is convenient to allocate certain data on a per-thread basis—this is known as *thread-local storage*. Extensions to various programming languages (for example, unofficial GNU compiler extensions to C and C++, where a __*thread* keyword was added) allow individual static variables to be declared and allocated as local to the thread. For these extensions to work, however, a certain level of support is required from the compiler, linker, system libraries, and the operating system itself.

Implementing thread-local storage requires user code to be able to access a per-thread data structure known as the *dynamic thread vector* (DTV) [Dre02b]. One commonly used model for this involves storing a pointer to the thread's DTV inside a TCB which can be accessed by the thread—this is illustrated in Figure 6.10. This model can be naturally supported by the new thread model, by simply adding a DTV pointer to the declaration of the user-level TCB.

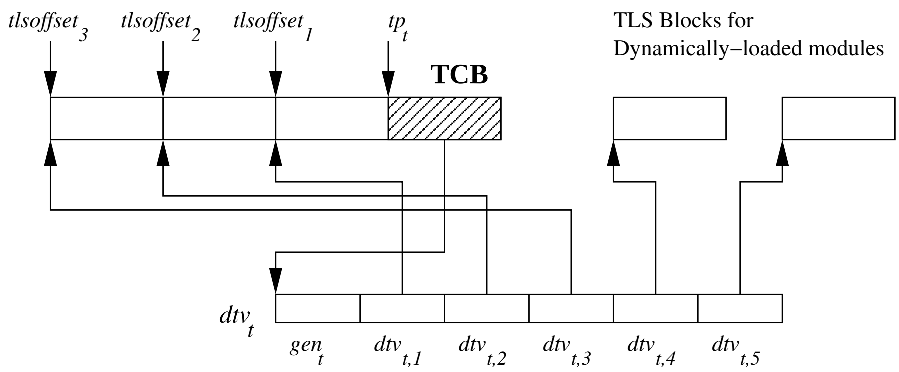

Figure 6.10: Method for accessing the thread-local storage vector from pointer in the TCB. Source: [Dre02b].

Placing a pointer in the TCB is not in itself enough, however. Threads must be able to access their thread-local storage quickly, which means locating their TCB, however under the Mungi implementation this would require a system call, unless restrictions were placed on the location and size of stacks and TCBs, allowing a thread to locate its TCB from the stack pointer. The approach taken by many architectures (including x86 and IA-64) to solve this problem is to use one of the CPU's registers to point to the TCB or thread data.

To support this, the Mungi build process was modified to avoid use of a particular register—register number 14 in this case, since it is untouched by L4/Alpha, although any general-purpose register would have been suitable. The various thread creation trampoline stubs (described in Section 6.2.5) were modified to set this register to point to the user-level TCB when starting a thread—this is why the user-level TCB address is placed on trampoline's stack, as shown in Figure 6.6.

Of course, setting a register to point to the thread's TCB does not provide thread-local storage. Compiler and linker modifications are also needed, but this is a first step, and is already useful to other code needing fast access to the TCB.

6.5. Kernel-level thread management

This section describes the details of the thread management code inside the Mungi kernel.

6.5.1. Kernel data structures

The kernel requires an efficient way of managing all threads in the system, which implies a number of data structures for tracking threads. In particular, the kernel must be able to:

1. locate a Mungi thread from its L4 thread ID, when the thread faults or performs a system call;
2. locate a Mungi thread from its user-level TCB pointer, when another thread performs a system call involving it (for example, *ThreadDelete*);
3. manage the allocation of L4 threads and tasks—in particular, locate a free thread in an appropriate task for *ThreadCreate* or *PdxCall* system calls;
4. keep track of which threads have their user-level TCBs in each Mungi object, so that when an object is deleted any threads inside it can be cleaned up.

To fulfil these requirements, the implementation uses the following data structures to track threads:

1. a hash table from L4 thread IDs to kernel TCBs;
2. a hash table from user-level TCBs to kernel TCBs;
3. an array of pointers to L4 task structures (described in Section 6.5.1), one for each L4 task—this is feasible since L4 has a small, fixed number of tasks, however it will require revision for future versions of L4;
4. a list of threads, associated with every object table entry for objects containing user-level TCBs.

Kernel TCB

The Mungi kernel-level thread control block declaration is shown in Figure 6.11. Each thread has a kernel TCB, which is allocated on the kernel's heap. TCBs contain the following fields:

utcb: pointer to the thread's user-level TCB;

object: pointer to the object table entry of the object containing the thread's user-level TCB;

```
typedef enum upcallid {UPCALL_NONE,         /* a normal/PDX thread */
                       UPCALL_MAC,          /* validation upcall for MAC */
                       UPCALL_THREADDEATH /* thread death notification upcall */
                      } upcallid_t;

struct mthread {
        struct utcb       *utcb;            /* user-TCB pointer */
        object_t          *object;          /* object containing utcb */

        l4_threadid_t     l4id;             /* L4 thread ID */
        struct l4_task    *task;            /* L4 task info */

        struct mthread    *pdx_caller;      /* calling thread (if PDX) */
        struct mthread    *pdx_callee;      /* PDX that we have called */
        upcallid_t        upcall;           /* calling source (if upcall) */

        struct checkpoint *checkpoint;      /* checkpointing data */
        bool              checkpointed;     /* thread has been saved */
        bool              user_created;     /* thread was user-started */
        void              *stack;           /* kernel-supplied stack */
        threadstate_t     state;            /* actual thread state */
        uint8_t           priority;         /* L4 thread priority */

/* Each thread is referenced by four data structures:
 *  - hash from user TCBs to kernel TCBs (threaddata.c)
 *  - hash from L4 thread IDs to kernel TCBs (threaddata.c)
 *  - list of threads in an l4_task structure (l4threads.c)
 *  - list of threads with user-TCBs in a given object (objects.c)
 */
        struct mthread    *utcb_next;       /* next in UTCB hash chain */
        struct mthread    **utcb_pprev;     /* previous ptr in hash chain */

        struct mthread    *l4id_next;       /* next in L4 ID hash chain */
        struct mthread    **l4id_pprev;     /* previous ptr in hash chain */

        struct mthread    *l4task_next;     /* next in task's thread list */
        struct mthread    **l4task_pprev;   /* previous ptr in list */

        struct mthread    *obj_next;        /* next in obj's thread list */
        struct mthread    **obj_pprev;      /* previous ptr in list */
};
```

Figure 6.11: Kernel TCB declaration (kernel/include/mungi/threads.h).

l4id: L4 thread ID of the thread;

task: pointer to the kernel structure managing the L4 task which contains this thread.

pdx_caller: if this is a PDX thread, pointer to the kernel TCB of the calling thread, otherwise NULL;

pdx_callee: if this thread is in a PDX call, pointer to the kernel TCB of the thread that is executing the PDX call, otherwise NULL if not in a PDX call;

upcall: field indicating if this thread is executing an upcall for the kernel, and, if so, what type of upcall—the currently used upcall types are:

> **NONE:** indicates that this is a normal user-created thread;
>
> **MAC:** validation upcall to the user-level MAC policy object;
>
> **THREADDEATH:** a thread death notification PDX upcall;

checkpoint: pointer to data used by the persistence system in checkpointing the thread;

checkpointed: boolean flag indicating if the thread has been checkpointed;

user_created: flag indicating that the thread was created directly by user code;

stack: thread's stack pointer, for kernel-allocated stacks (i.e. upcalls and PDX threads);

state: thread state, mirrored by the user-level TCB;

priority: thread priority, also mirrored by the user-level TCB.

Finally, there are a series of pointers that tie each TCB into four doubly-linked lists for the data structures described in the previous section:

1. a hash chain for the hash table from user-level TCB pointers to kernel TCBs;
2. a hash chain for the hash table from L4 thread identifiers to kernel TCBs;
3. a list of threads in each L4 task;
4. a list of threads with user-level TCBs in each object.

L4 task management

Figure 6.12 shows the *l4_task* data structure. These structures are also allocated on the kernel's heap, one for each active task. The APD structure contains a list of *l4_task* structures, for the tasks that make up the APD. Each active task is also pointed to from an array indexed by task ID, and linked into a list of all allocated tasks.

The contents of the *l4_task* structure are:

```
struct l4_task {
        l4_taskid_t       l4id;             /* L4 task ID */
        struct apd        *apd;             /* APD for this task */

        /* kernel-only table of lthreads, entries are either:
         *  - the TCB ptr, for a running thread
         *  - -1 (MTHREAD_INVALID), for an allocated but non-running thread
         *  - NULL, for an unallocated thread */
        struct mthread    *lthreads[L4_THREADS_PER_TASK];

        /* user-shared table of lthreads (separate object), entries are either:
         *  - the UTCB ptr, for a user-started thread
         *  - -1 (UTHREAD_INVALID), for a kernel-started/allocated thread
         *  - NULL, for an unallocated thread */
        struct utcb       **uthreads;

        struct mthread    *free;            /* free threads list */
        struct mthread    *used;            /* used threads list */

        struct l4_task    *apd_next;        /* next in APD task list */
        struct l4_task    *all_next;        /* next in global task list */
        struct l4_task    **all_pprev;      /* prev ptr in global task list */
};
```

Figure 6.12: L4 task structure declaration (kernel/include/mungi/l4threads.h).

l4id: L4 task ID of the task;

apd: APD of which the task is part;

lthreads: an array of pointers to kernel TCBs, indexed by L4 local thread ID—this is feasible, since there are a fixed number of threads per task, but will need revision to support L4 version 4;

uthreads: pointer to the shared array of user-level TCB pointers, used for local thread creation;

free: list of free (allocated but currently unused) threads in the task;

used: list of busy (allocated, and executing) threads in the task;

apd_next: next task in the APD's list of tasks;

all_next, all_pprev: pointers to a doubly linked list of all allocated tasks, used for checkpointing.

Hash tables

Finally, there are two hash tables—a hash from L4 thread IDs to kernel TCB pointers, and a hash from user-level TCB pointers to kernel TCB pointers. These are fairly straightforward structures (the hash functions are discussed in Section 6.5.2).

Note that it would be possible to avoid the use of the hash from L4 IDs to TCBs, by first looking up a thread's task ID in the table of L4 tasks, and then looking up the local thread ID in the task's table of threads, but this will not be as simple under future versions of L4, so the hash table has been used for greater portability.

6.5.2. Code overview

L4-specific thread and task management

For portability reasons, the code managing L4 tasks and the threads within those tasks (which is fairly specific to the L4 version 2 API) has been kept in a separate module (kernel/src/l4threads.c). This code is responsible for:

- allocating L4 threads on demand, when needed for upcalls, *ThreadCreate*, or PDX calls;
- keeping a pool of unused but ready threads for each task, which become available when threads are deleted or exit;
- releasing all the structures under its control, when an APD is cleaned up;
- providing code for traversing all threads in the system, for use when checkpointing;
- controlling the setting of L4 scheduler priorities for user threads.

Thread hash tables

As discussed in Section 6.5.1, there are two hash tables used for managing threads—the first from L4 thread IDs to kernel TCBs, and the second from user-level TCBs to kernel TCBs. The code responsible for maintaining these hash tables is also in a separate module (kernel/src/threaddata.c).

The hash functions used by both tables are shown in Figure 6.13. These have been chosen to avoid collisions by taking as much unique information as possible to form a 10-bit number (each hash table is currently $2^{10} = 1024$ entries in length). The L4 ID hash function performs an exclusive or of the task ID with the local thread ID (the other components of L4 thread IDs are redundant in Mungi), whereas the user-level TCB hash function utilises the fact that all valid user-level TCBs must be correctly aligned to drop insignificant bits from the address.

$$\begin{aligned}
\text{L4ID_HASH}(id) &= (id.task \otimes id.lthread)\ \&\ (\text{L4ID_HASHSIZE} - 1) \\
\text{UTCB_HASH}(p) &= (p/\text{sizeof}(\text{struct } utcb))\ \&\ (\text{UTCB_HASHSIZE} - 1)
\end{aligned}$$

Figure 6.13: Hash functions for L4 thread IDs and user-level TCB pointers.

Thread-related operations

The rest of the Mungi kernel interfaces with the thread code through the functions in this module (found in kernel/src/threads.c). The key functions provided by this module are:

thread_lookup_l4id: This function returns a pointer to the kernel TCB for the thread with the given L4 thread ID. This includes checking for user-created threads, as discussed in Section 6.3.2.

thread_create: This function creates a new thread, and is called to start the first user thread, or in response to a *ThreadCreate* system call.

thread_pdx_create, thread_pdx_upcall: Both of these functions start a PDX thread, and are used either by the system call handler in response to a PDX call (*thread_pdx_create*), or other parts of the kernel starting a PDX upcall function (*thread_pdx_upcall*).

thread_pdx_return: A function used when a PDX thread returns, which cleans up its state information, and returns to the original caller (in the case of a normal PDX call, but not in the case of an upcall).

thread_kill: This is called by the system call handler in response to a *ThreadDelete* call, and stops the thread before executing its death notification callback function, if one is registered.

thread_die: This function is called by other parts of the kernel, when a thread should be killed due to an unhandled exception (such as a protection fault)—it also executes any death notification callback function.

6.6. Portability of implementation

Due to time constraints involved, it was impossible to provide an implementation for all versions of Mungi—this implementation has targeted the L4/Alpha platform. Of course, very little of the code is architecture dependent, so care has been taken to ensure portability to other versions of L4.

6.6.1. L4/MIPS

The main Mungi platform apart from Alpha is the original L4/MIPS port. The L4/MIPS kernel has only minor differences to L4/Alpha (both are based on the L4 version 2 API), and most of these differences have been accounted for in the implementation—with the result that porting from Alpha to MIPS should be a straightforward task.

A different architecture will doubtless uncover bugs in the implementation, but the following should be all that is necessary to support L4/MIPS:

- write a MIPS equivalent for the trampoline assembly code discussed in Section 6.2.5;
- verify the MIPS version of the atomic update code for local thread creation (this has been written, but is untested);
- verify the MIPS versions of calls to L4, such as *lthread_ex_regs*, and *thread_schedule* (these calls have slightly different parameters to the L4/Alpha versions).

6.6.2. Pistachio

As discussed in Section 4.3.1 however, L4 version 4 differs significantly from version 2. Care has been taken in the implementation to restrict L4 version dependencies, but porting to Pistachio (the implementation of the version 4 API) will be more demanding.

Mungi is in the process of being ported to Pistachio, hence this thread implementation will also need to be ported. As a minimum, the following parts of the implementation will need revision:

- Much of the thread and task management code (in kernel/src/l4threads.c) is specific to version 2, in particular the allocation of threads within tasks—it will no longer be necessary to use multiple tasks per APD, but it will also not be feasible to use an array of threads within a task.
- Creating a thread requires use of the *ThreadControl* system call, which can only be performed by the privileged root task. Due to the layout of Mungi, this will require an IPC to the RAM pager task to perform thread allocation.
- The access control method for scheduling has changed, but this simply means starting threads with the Mungi kernel as their scheduler, rather than an MCP of zero.

7. Results

7.1. Testing environment

All benchmarks were performed on the same test machine: *rothaus*, a DEC AlphaStation 433au desktop workstation. This machine has a 433MHz Alpha EV5.6 (21164A) CPU [Sam97], performing in-order execution, with split first-level instruction and data caches of 8kB each, and a unified second-level cache of 96kB. All tests were run directly from RAM, 192MB of which was installed.

The Alpha architecture [Com98] provides a 32-bit cycle counter, which increments once per clock tick—on a 433MHz machine this overflows after approximately nine seconds, allowing enough time for the short style of benchmarks used. The cycle counter is retrieved into a user register with the *rpcc* instruction, allowing for low-overhead timing.

In order to collect standard deviations as well as averages, the results of individual measurements were taken and stored into an array by the timing code. This measurement involves a small amount of overhead (typically in the order of five clock cycles), which was also measured on each run and subtracted from the results.

7.2. System call times

The execution time of all relevant system calls was measured for three different systems, using the techniques described in the previous section. The results obtained are summarised in Table 7.1, shown graphically in Figures 7.1 and 7.2, and discussed in further detail in the following sections.

7.2.1. Recent Mungi implementations

Obviously, the new implementation of Mungi must be benchmarked and compared to the previous version, 1.2, which is publicly available [Mun02a]. Measurements were made of the time to:

- create a thread in the same APD as the caller (with the use of the kernel, and also locally in the new implementation);

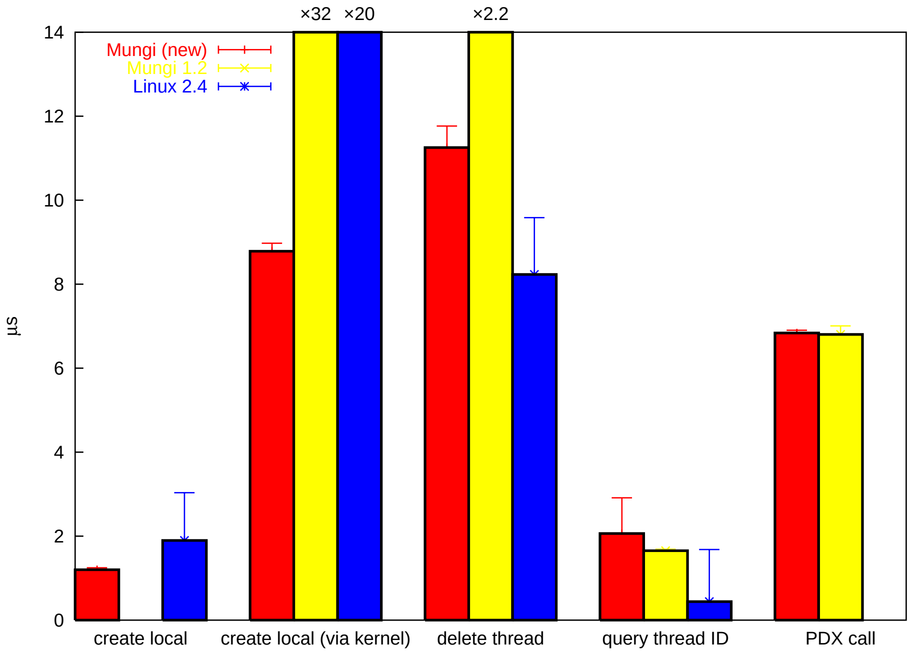

Figure 7.1: Graph of micro-benchmark results, general operations.

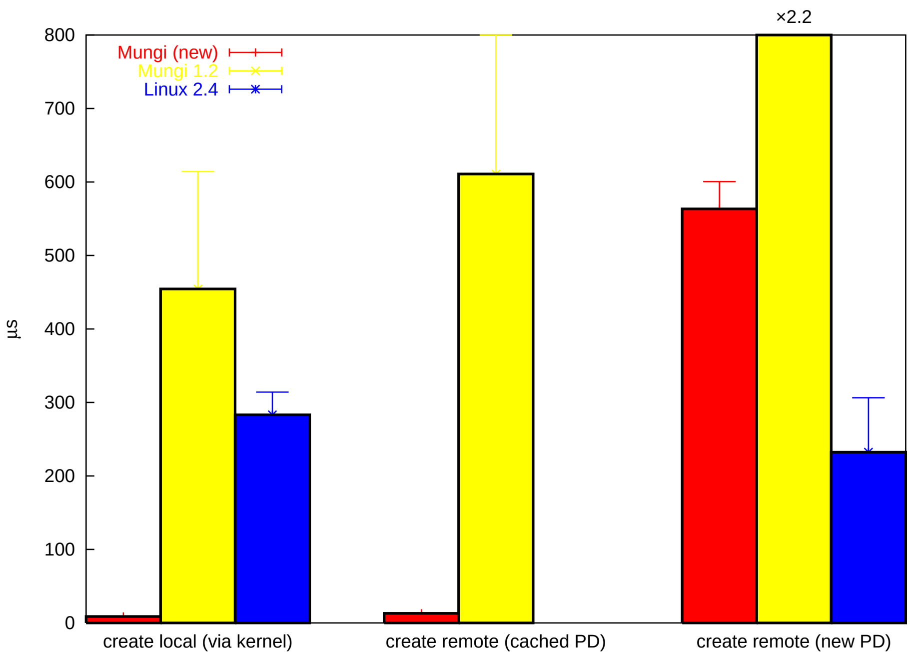

Figure 7.2: Graph of micro-benchmark results, kernel-assisted thread creation.

	Mungi (new)		*Mungi 1.2*		*Linux 2.4.18*	
Operation	$\overline{x}$	σ_{n-1}	$\overline{x}$	σ_{n-1}	$\overline{x}$	σ_{n-1}
create thread (local)	1.20	0.05			2	1
create thread (via kernel)	8.8	0.2	500	200	280	30
create remote thread (cached)	13.0	0.4	600	200		
create remote thread (new PD)	790	40	1800	400	230	70
delete thread	11.3	0.5	31	3	8	1
query thread ID	2.1	0.8	1.65	0.03	0.4	1
PDX call (cached)	6.84	0.07	6.8	0.2		

Table 7.1: Summary of microbenchmark results, with values given in μs.

- create a thread in a different APD to the caller (with both cached and uncached APDs);
- delete a thread;
- query a thread's ID—note that under the new implementation, using the thread register discussed in Section 6.4, this could be done within a single cycle, but for a fair comparison a system call invoking the kernel has been used;
- perform a PDX call—while not strictly part of the thread model, these times are important, since the PDX path shares a lot of code with the thread creation path, and because PDX performance is crucial in Mungi, it is important to check that it has not degraded.

The results of these measurements are shown in Table 7.1. Of particular interest was the significant difference between kernel-assisted thread creation in the caller's APD versus in a cached APD—in terms of the implementation, these are almost identical, yet they differ in time by approximately 1800 clock cycles. The extra work required for the remote creation is mainly:

- validating access to the specified PD object;
- performing two extra MAC validations;
- locating the cached APD in a hash table.

Of these tasks, the extra PD validation should be another lookup in the validation cache, the MAC system was disabled during the test runs, and would be expected to cost little more than the function call overhead, leaving only the hash table lookup. Further investigations, which will be discussed in Section 7.3, revealed that VCache implementation was to blame.

Operation	*Time*
null system call	4.6
create thread	83
create thread (cached)	48
delete thread	48
create task	600
delete task	310

Table 7.2: Relevant microbenchmark results (in μs) from previous Mungi implementation. Source: [HEV+98].

7.2.2. Mungi 1.0

Since results have been published for a previous implementation of Mungi [HEV+98], they should also be compared. This implementation of Mungi, known as version 1.0, had a somewhat different API [HVER97] from the current version—in particular, instead of APDs instantiated from explicit PD objects, it used a hierarchy of protection domains known as *tasks*. Creating a thread in a different APD is roughly equivalent to creating a new task.

Table 7.2 shows the relevant results, which were obtained on a 100MHz R4600-based SGI Indy workstation with 64MB of RAM. Unfortunately, the Mungi 1.0 implementation was specific to L4/MIPS, so the tests could not be repeated on the Alpha hardware. Furthermore, the MIPS system is significantly slower than the Alpha, so a direct comparison of results cannot be made. Based on published results [SPE00] for the SPECint95 benchmark [SPE95], the Alpha system should be around seven times faster than the Indy.

Based on this ratio, we can see that the times for thread creation and deletion under the new implementation are similar to those for Mungi 1.0, to the extent that they are are meaningfully comparable.

7.2.3. Linux

Comparisons between Mungi versions are useful, but benchmarks for a widely used system would also be instructive. As an example of a real system, comparative benchmarks of Linux were taken on the same hardware. Linux is a widely used, freely-available re-implementation of UNIX, with ports to many platforms. The latest available version of the kernel at the time, Linux 2.4.18, was used for benchmarking—the results of these benchmarks are also shown in Table 7.1.

Of course, Linux and Mungi differ greatly in their programming interface, so direct comparisons could not be drawn. Threaded processes are akin to threads in the same Mungi APD, so the Mungi times for creating a local thread are compared to times for various thread creation operations under Linux.

There are many different options for threading under Linux, with various tradeoffs. The most commonly used is the Linux implementation of Pthreads, which provides full kernel-level threads. The time to perform a *pthread_create* operation was measured at 277μs, and is reported as the kernel-assisted thread creation time.

A number of user-level thread packages are also available for Linux. Three of these libraries were investigated:

- Patched MIT Pthreads [Dyb98] is a fully user-level implementation of the Pthreads standard. Unfortunately, this package doesn't support Linux on Alpha hardware, and attempts to build it under the OSF/1 operating system were unsuccessful.
- GNU Portable Threads [Eng01] is another user-level thread library aiming for completeness and portability. Version 1.4.1 of this library was benchmarked, with the *pth_spawn* operation averaging 80.4μs.
- The State Threads Library [SGI02] is a user-level library targeted at supporting highly scalable network servers. It supports a limited thread API with explicit preemption points for context switches. The *st_thread_create* function was benchmarked at 1.90μs, using version 1.3 of the library.

Since threads and processes are more distinct concepts in the UNIX model, there is no way to directly create a thread in a different protection domain, and there is no implicit caching of protection domains as in Mungi. The closest analogue to Mungi's thread creation in another APD is the Linux *fork* system call, which creates another process, and was timed at 233μs. Interestingly, under Linux a *fork* operation was faster than *pthread_create*—this can be explained, however, by the fact they internally both use Linux's *clone* primitive to perform the operation.

Thread deletion was timed using the *pthread_cancel* function—this operation doesn't actually delete the thread, merely send a cancellation request to it, but it was the closest available equivalent to Mungi's *ThreadDelete* call.

For querying a thread's ID, the *getpid* system call was used. The result (of 0.439μs) has a particularly high standard deviation due to several outlying measurements over 20μs.

7.3. System call profiling

In order to provide a breakdown of the Mungi system calls measured, timing profiles of the important operations were taken. This involved updating the clock code (which is part of Mungi—the majority of the code can be found in lib/mlib/clock.c) to time points of execution within a large system call path, storing these in a memory object, and taking averages of various components of the overall cost. There is obviously significant overhead in these measurements, but they do provide good information about the relative costs involved in a system call. Example output from the clock code (for a local thread creation) is shown in Figure 7.3.

```
** ThreadCreateLocal Timings
        ( 999 472 380 374 462 378 387 381 )
        total setup time              -    436 (8 readings)
                ( 358 123 125 123 128 127 135 128 )
                l4_myself                     -    112 (8 readings)
                ( 385 192 107 107 108 107 108 107 )
                get uthread table             -    109 (8 readings)
                ( 111 72 71 67 149 67 67 67 )
                get free uthread              -     40 (8 readings)
                ( 145 85 77 77 77 77 77 79 )
                tcb/stack setup               -     43 (8 readings)
        ( 795 369 342 343 352 512 346 349 )
        ex-regs                       -    383 (8 readings)
** Bottom line
        ( 1978 945 806 801 955 968 811 808 )
        total time                    -    966 (8 readings)
```

Figure 7.3: Example output from the profiling clock code, showing a local thread creation.

This example shows that the *lthread_ex_regs* call takes 40% of the time involved in a local thread creation, and that of the remaining 60%, the most significant sections are the 12% which is used in L4's *id_nearest* call (which is required to determine the creator's L4 ID), and 11% spent locating the shared table of user-level TCB pointers.

The most interesting results of this profiling are shown in Table 7.3: a breakdown of the in-kernel costs incurred during local and inter-domain thread creations. The highest proportion of the time comes from the security validations, which take 45% of the cost for a local thread creation, and rise to 53% for an inter-domain thread.

Operation	validations	locate APD	check TCB	allocate thread	initialise thread	hashtable insertions	initialise TCB
create thread (local)	45%	N/A	15%	12%	9%	16%	4%
create thread (remote)	53%	10%	9%	8%	5%	9%	5%

Table 7.3: Breakdown of in-kernel costs incurred during thread creation.

Timings of the VCache implementation showed that each cache lookup was averaging 717 cycles (1.66μs), even without the added cost of the security code which calls it—obviously this is responsible for a significant proportion of the time. Since the VCache used in this test stored only 16 entries, there is presumably something broken in the current implementation, which, if fixed, would result in an improved time for thread creation (and many other system calls).

7.4. Local thread creation

7.4.1. Mungi

To test the effectiveness of the local thread creation optimisation discussed in Section 6.3, a benchmark was run which creates threads in the same protection domain, and measures the cost for each creation. The threads created sleep without exiting, so this benchmark should show the increasing cost of thread creation with other threads already present.

The results of this test for two repeated executions are shown graphically in Figure 7.4, and conveniently illustrate three different ways that threads can be created. For both testing runs, the creation time rises gradually until slightly over 250 threads have been created. At this point, both executions show an increase in creation time, however the initial run (shown in red) increases significantly more than the repeated run (shown in blue).

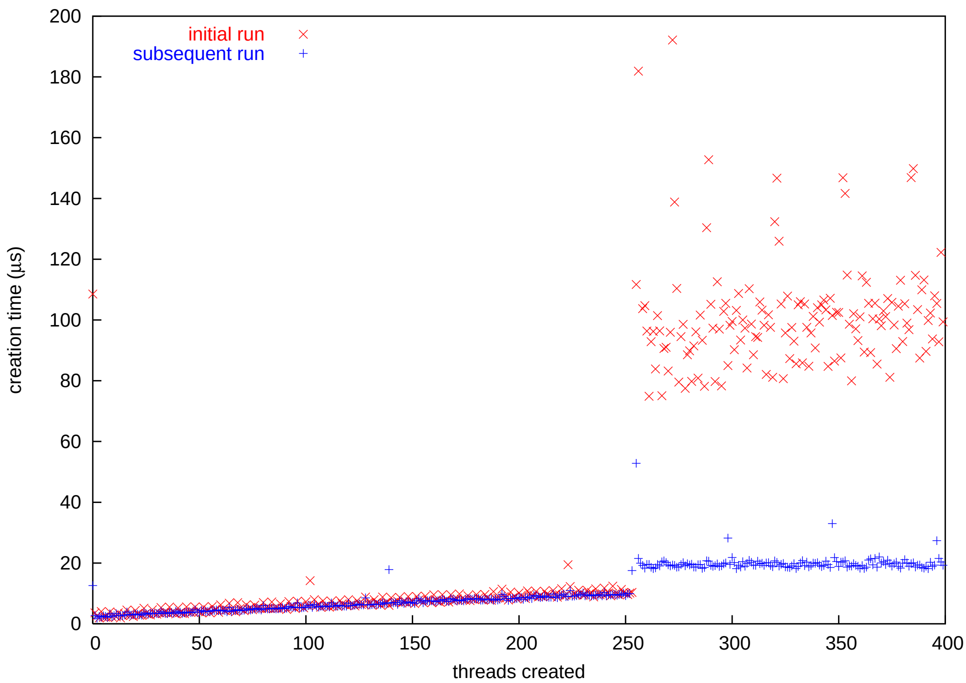

Figure 7.4: Plot of thread creation time against number of local threads created, under the new Mungi implementation.

A probable explanation for the effects seen in this graph is as follows: due to the upcall thread, and the thread executing the test code, there are 254 unused threads in the creator's task (recall that L4/Alpha has 256 threads per task). Up until this point, local thread creation is used, resulting in the small but linear increase in time. For all creations after the 254th thread, the kernel must be invoked—recall, however, that this is only an issue under current versions of L4. Under Pistachio, the number of threads per task is unlimited, so this performance limit on the number of locally created threads should not apply—instead, the user task will need to request more threads be allocated to it by the privileged server, which can be batched to avoid significant impact.

If more than 254 threads are created, then the kernel must be invoked. The first time this test is run, each new thread in the other task must be created by an IPC to the upcall thread for that task, however on the subsequent test execution, the threads have been started and the kernel simply sends them an IPC to begin execution, resulting in the significantly faster thread creation time seen.

7.4.2. Linux

For comparison, the same test was performed on Linux using both kernel- and user-level threads. Figure 7.5 shows the results for the kernel-level *pthread_create* call, and Figure 7.6 shows the results using the user-level State Threads Library.

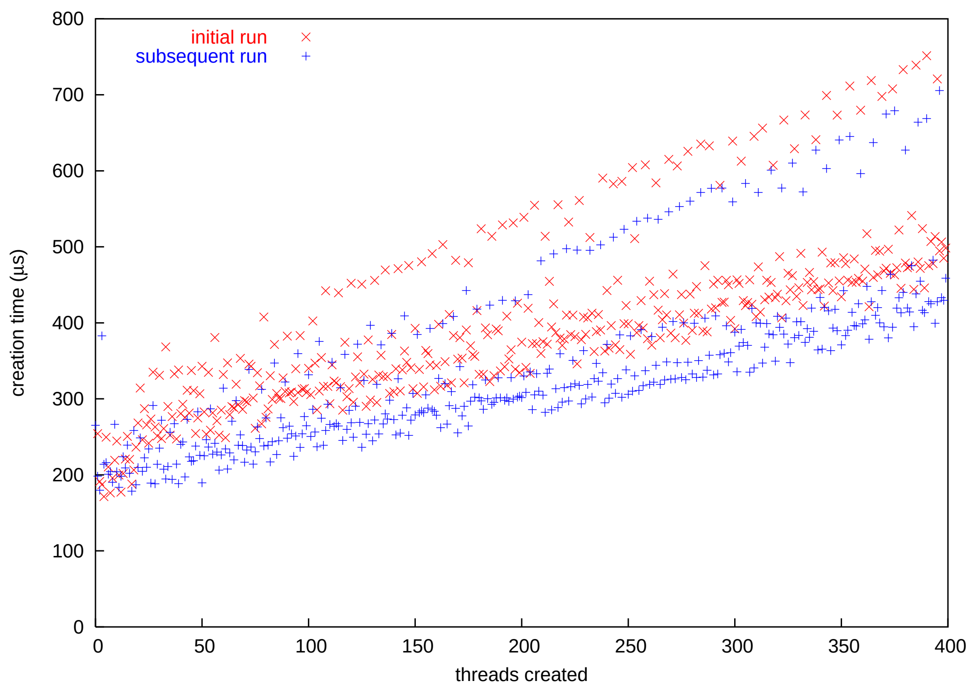

Figure 7.5: Plot of thread creation time against number of threads created, using *pthread_create* under Linux 2.4.

As can be seen, the results for Linux don't show the same interesting modality as for Mungi, since the same creation mechanism is used without interruption. The kernel threads show a roughly linear increase in creation time, presumably due to the thread management data

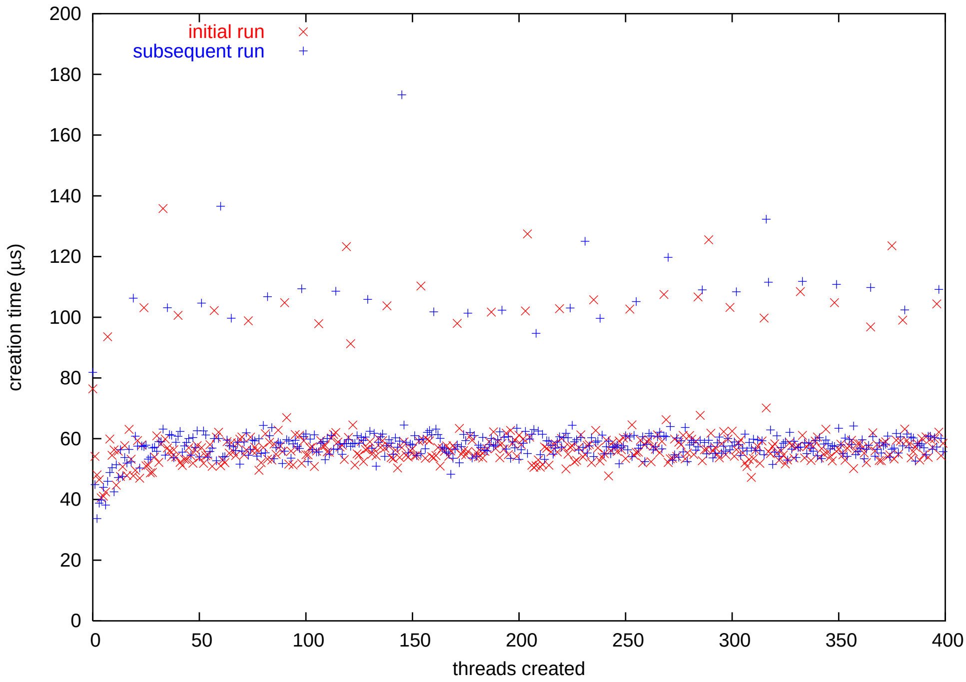

Figure 7.6: Plot of thread creation time against number of threads created, using *st_thread_create* under Linux 2.4.

structures used inside the Linux kernel. The cost of the user-level library's thread creation rises rapidly over the first few threads, but avoids the scalability problems of the kernel, showing a basically constant creation cost (up to 10,000 threads were tested, and showed no significant increase).

8. Discussion

8.1. Performance analysis

As discussed in Section 3.3.1, good performance was one of the main goals of the implementation. The results presented in the previous chapter would suggest that this goal has been achieved.

8.1.1. Comparison with previous Mungi implementations

Mungi 1.2

The microbenchmarks taken show that for all thread-related system calls other than querying a thread's ID, the new implementation is significantly faster (often by an order of magnitude) than the previous Mungi implementation.

To be fair, this difference is mostly due not to the extremely good performance of the new implementation, but the unoptimised and unbenchmarked state of Mungi 1.2. This version was a complete rewrite of the kernel from version 1.0, for which the main goals were cleanliness and portability over high performance—in general, it has not been significantly benchmarked or optimised.

Of course, the notable exceptions to the trend are the PDX call path, which has been significantly optimised in both kernels, and is of equivalent time, and querying the thread's ID. The slightly higher time for a thread ID query is unexpected, but not totally surprising due to the nature of the data structures. The previous implementation maintains a table of tasks indexed by task ID, and for each task a table of threads indexed by local thread ID. Hence, locating a thread requires two array lookups. Under the new implementation, this is implemented as a hash table from L4 thread IDs to kernel TCBs (for ease of portability to Pistachio), with potentially higher cost due to hash function collisions—obviously, this code could be instrumented to determine the cause of the extra cost, if the time difference was deemed important.

Mungi 1.0

As shown in Section 7.2.2, the results of the new Mungi implementation are comparable with the numbers published for Mungi 1.0, allowing for the inevitable error in comparing different architectures. This is promising, since Mungi 1.0 was significantly optimised to produce good microbenchmark times, and was closely tied to the L4/MIPS microkernel it ran on. The only system call that is significantly cheaper is task creation, which has no direct equivalent in the new Mungi API.

8.1.2. Comparison with Linux

Due to the significantly different nature of the Linux system to Mungi, it is harder to draw conclusions from the results obtained.

Normal thread creation times are significantly faster in Mungi than in Linux. Ongoing work aims to improve the Linux thread implementation [Dre02a, DM02], however due to its experimental nature and incomplete status this was unable to be benchmarked. It would be interesting to see how improved threading under Linux compared with Mungi.

What is both surprising, and extremely promising for Mungi, is that fully user-level thread implementations on Linux are slower than Mungi's fast local thread create, which provides all the benefits of a kernel-scheduled thread (in this case, the L4 microkernel). Even the State Threads Library, which offers restricted thread semantics, was several hundred cycles slower than Mungi's local thread creation.

These results point to a powerful argument for the microkernel-based design of Mungi, which offers primitives (in this case, thread creation) that are significantly cheaper than the alternatives available in Linux, yet still fully functional.

8.1.3. Summary

In all important cases, the new Mungi implementation has been shown to outperform both Mungi 1.2 and Linux, and be comparable to the highly optimised Mungi 1.0 implementation. Further performance gains could be achieved by kernel improvements, in particular to the security and VCache code, which are bottlenecks in the current implementation. Nevertheless, the new thread model design has proven implementable with good performance, and without adversely impacting on the kernel's other functions.

8.2. Possible L4 improvements

L4 aims to be a minimal yet flexible microkernel, providing the abstractions required to implement arbitrary systems at higher levers. This section discusses several problems with the L4-provided abstractions that had to be accounted for in the implementation, and suggests alternatives.

8.2.1. Limited number of threads per task

As discussed in Section 6.1, the limited number of threads per task is a significant restriction on system design. In the case of Mungi, it requires significant extra code for allocating multiple tasks to APDs, and limits the usefulness of local thread creation, as demonstrated in Section 7.4. Furthermore, it results in repeated page faults from the different tasks allocated to an APD, even though all threads within an APD share the same protection domain (by definition).

A solution to this restriction is to allow a mechanism for arbitrarily allocating threads to tasks. Such a mechanism has been introduced by version 4 of the L4 API, although it has other problems which will be discussed in Section 8.2.3.

8.2.2. Access control on the *lthread_ex_regs* call

The *lthread_ex_regs* system call may only be used on threads within the same task as the caller. This forces Mungi to make use of the upcall thread, discussed in Section 6.1, to perform *lthread_ex_regs* operations on behalf of the kernel.

The use of upcall introduces significant complexity into the Mungi design, particularly since care must be taken to protect the kernel from malicious upcall threads that have been taken over by user code in the same task. There is also a time overhead involved in the cost of communication with the upcall threads.

What would be more suitable for Mungi, and presumably other systems as well, is if the creator of a task was able to use *lthread_ex_regs* on threads within that task. This would allow Mungi to directly control user threads, without the need for upcall, while still maintaining security (since the creator of a task can delete all the threads inside it simply by using the *task_new* system call).

In fact, this proposal is used and relied upon by the current implementation of persistence in Mungi [NBH02], and can be implemented in L4/Alpha (and presumably other L4 kernels) without significant difficulty.

8.2.3. Use of the privileged task in L4 version 4

In addressing the problem of a fixed number of threads per task, discussed in Section 8.2.1, version 4 of the L4 API introduces a *ThreadControl* system call, which allows a privileged thread to create and delete other threads (effectively controlling the allocation of threads to tasks). A privileged thread is defined as one within the root task, the initial user task started by L4.

This design addresses the problem of allocation of threads to tasks, but in a way which is very unsuited to Mungi. In the current implementation, the root server is the RAM pager task (discussed briefly in Section 6.1), which manages the SAS, and knows nothing about the threads and tasks used by Mungi. A port of Mungi to Pistachio requires the RAM pager to perform this thread allocation, using IPC messages from the Mungi kernel to request allocations. Not only does this introduce extra overhead, it also complicates the types of messages being handled by the RAM pager, leading to greater potential for deadlocks.

The concept of a privileged thread or task is not problematic, however the fact that only one task can be privileged is. What is needed is a mechanism for transferring the privilege to another task, or possibly sharing the privilege—this could take the form of a special type of IPC message to delegate the privilege, which implies cooperation on the behalf of the recipient, however the exact mechanism is not very important.

9. Conclusion

In Chapter 3, the following were proposed as design and implementation goals for a thread model:

- high performance;
- simplicity and flexibility;
- security.

The performance of the system has been tested, and these results are presented and discussed in the preceding chapters. Of course, the performance of a thread model is not indicative of the complete system, and further benchmarks testing more aspects of the implementation will need to be run and examined, but the results so far are extremely promising. We can conclude from these results that the thread model designed by this thesis can be implemented with good performance, and furthermore that the use of a microkernel such as L4 allows, rather than hinders, the provision of high performance thread primitives.

The design and implementation of the model have tried to maintain simplicity and flexibility, both in the selection of user-level TCBs for representing and identifying threads, and in the modular nature of the kernel implementation. Of course, the flexibility of any system can realistically only be judged through its use, and very little of this has happened to date. The Mungi device driver model [Les02], which was implemented in parallel with the thread model, uses the new model successfully, lending some weight to this argument. An implementation of the Pthreads standard, and the threaded applications that this would bring to Mungi, would allow better judgement of the model's flexibility.

Security has been provided through Mungi's natural protection mechanisms, namely password capabilities and mandatory access control, and was a key motivating factor in the decision to use user-level TCBs to represent threads. Using the same security mechanisms resulted in many benefits, in particular, the same caching mechanism could be used to make the provision of security lightweight[1], MAC support within the thread model was provided naturally and without additional work, and the resulting access control model for threads is extremely flexible.

[1] Unfortunately, due to the problems with the VCache discussed in Section 7.3, the security validations are not as lightweight as they could be.

To summarise, a new Mungi thread model has been successfully designed and implemented. As a result of the key design decisions of using user-level TCB objects to identify threads, and regular password capabilities to provide protection, a simple, flexible, and secure model has been provided. Furthermore, the implementation shows performance which is comparable to, or better than, all other systems to which it has been compared. This thread model should provide a strong foundation for future Mungi work, some suggestions for which are provided in the following section.

9.1. Future work

As with any thesis, many new questions have been raised and left unanswered. The biggest outstanding problems, resource management and distribution, are analysed further in the subsequent sections. Additionally, the following smaller issues were raised and could benefit from further work:

- As was discussed in Section 6.6, the implementation should be ported to other platforms on which Mungi runs. In particular, L4/MIPS and Pistachio on IA-64 (and other platforms as they become supported).
- A simple user-level convenience library has been provided for programming with the new thread model, its API is covered in Appendix B. This library is insufficient, and should be replaced by an improved programming environment that covers more than just thread creation and deletion. An implementation of the Pthreads standard [POS95] will also be required in the future when porting threaded POSIX applications.
- As discussed in Section 6.3.2, further consideration is required of how locally created threads can be made persistent without negating the performance benefits they provide.
- An implementation of thread-local storage (as discussed in Section 6.4), while involving significant improvements to the compiler and linker, would make programming threaded applications easier.
- As was discussed in Section 5.1.4, further investigation is required of the need for both inter-domain *ThreadCreate* and PDX calls. Another possible extension to the model would be the addition of an asynchronous PDX call primitive.

9.1.1. Resource management and scheduling

The main problem left unsolved by the thread model introduced in this thesis is that of resource management, in particular management of CPU time allocated to threads, and hence control over scheduling decisions. The problem of resource management has already been discussed, particularly in Sections 3.1.3, 3.2.2 and 5.1.5. Some form of resource management system based on bank accounts seems the most logical solution, with one distinct possibility being a design based on lottery scheduling [WW94]. Lottery scheduling is a probabilistic scheduling algorithm based on the transfer of currency known as *lottery tickets*. If the lottery tickets could be tied into the Mungi bank account model somehow, this would offer a particularly clean model for management of CPU time.

An alternative possibility would be to allow the scheduler to be implemented in user space as a separate component from the kernel. Whatever system is chosen, it must be based on the L4 scheduler, which was described in Section 4.3.1. The model (which is also described in Section 4.3.1) for implementing user-level schedulers is also feasible within Mungi, but involves a significant overhead (an extra context switch for each scheduler invocation).

Consideration is also required of Mungi on symmetric multiprocessor (SMP) systems. L4/Alpha has support for SMP machines, and support is being added to Pistachio—porting Mungi to such a system should be reasonably straightforward, however the question of load balancing between the multiple CPUs now becomes an issue. L4 itself provides only the means to migrate a thread onto a different CPU (the *l4_migrate* system call under L4/Alpha, or a parameter to the *schedule* call under Pistachio), so something within Mungi will be needed to perform this task.

9.1.2. Distribution

Mungi is designed to be easily extended to support distribution (one of the key features of a SASOS), and any distributed Mungi system will require support within the thread model.

In particular, this model will need to be extended to allow migration of threads onto different nodes. A mechanism is required to allow migration policies to be implemented at user level (potentially along with the schedulers as outlined above). At the simplest level, this could just be the addition of a *ThreadMigrate* system call, or thread migration could be handled at user-level without the intervention of the kernel. Whatever model is chosen depends on the design of distributed Mungi, and hence cannot yet be determined.

A. System Call Prototypes

This appendix presents the complete thread-related system call prototypes for the C language. These follow the established Mungi convention whereby every system call returns either zero on success, or an error code on failure. The exception to this is *ThreadMyID*, which can never fail, so for convenience it directly returns the TCB pointer to the user.

A.1. include/syscalls.h

```
/***************************************************************************
 *
 *      $Id: syscalls.h,v 1.4 2002/10/14 04:37:17 andrewb Exp $
 *      Copyright (C) 2002 Operating Systems Research Group, UNSW, Australia.
 *
 *      This file is part of the Mungi operating system distribution.
 *
 *      This program is free software; you can redistribute it and/or
 *      modify it under the terms of the GNU General Public License
 *      version 2 as published by the Free Software Foundation.
 *      A copy of this license is included in the top level directory of
 *      the Mungi distribution.
 *
 *      This program is distributed in the hope that it will be useful,
 *      but WITHOUT ANY WARRANTY; without even the implied warranty of
 *      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 *      GNU General Public License for more details.
 *
 ***************************************************************************/

/* Mungi system calls */

#ifndef __MUNGI_SYSCALLS_H
#define __MUNGI_SYSCALLS_H

#include <sys/types.h>
#include <exception.h>


/**********\
```

```
 * Threads *
\**********/

int
ThreadCreate ( thread_t ip, void *stack, cap_t param, struct utcb *tcb,
               const apddesc_t *pd );
/*
 * Start a new thread starting execution at address "ip" with stack "stack"
 * and parameter "param".
 * "tcb" specifies the location of the new thread's TCB, and contains initial
 * thread attributes.
 * "pd" specifies optional APD information. If pd is NULL the new thread
 * shares the caller's APD (ie. changes done by the new thread affect the
 * calling thread's APD).
 *
 * errors: ST_THR, ST_INFO, ST_PROT, ST_OVFL.
 */

int
ThreadDelete ( struct utcb *tcb, int status );
/*
 * Terminates thread with an exit value of "status".
 *
 * errors: ST_PROT, ST_THR.
 */

int
ThreadSleep ( struct utcb *tcb, time_t time );
/*
 * Make the thread sleep for the amount specified in time in
 * nanoseconds. A value of zero specifies a thread yield. A value of
 * SLEEP_INFINITY suspends the thread until a ThreadResume() system call is
 * performed on the thread.
 *
 * errors: ST_PROT, ST_THR.
 */

int
ThreadResume ( struct utcb *tcb );
/*
 * Resumes sleeping thread
 *
 * errors: ST_PROT, ST_THR.
 */

int
ThreadSetPriority ( struct utcb *tcb, uint8_t priority );
/*
 * Sets the L4 priority level of thread.
```

```
 *
 * errors: ST_PROT, ST_THR.
 */

struct utcb *
ThreadMyID ( void );
/*
 * Returns TCB pointer of calling thread
 *
 * Always successful.
 */


#endif /* __MUNGI_SYSCALLS_H */
```

B. Convenience Library Interface

The changed thread model introduced by this thesis makes creating a thread and waiting for it to exit somewhat more complicated for the programmer than the previous design. To address this, and avoid code duplication, a small library of utility functions was written. This library provides the bare essentials—simple *thread_create_local* and *thread_create_remote* functions. It is hoped that future work will investigate more appropriate programming environments for SASOS systems, and incorporate this thread model. In the meantime, the (far from perfect) interface to this library is documented here.

B.1. Thread library

Mungi already contains a library of functions to make using the system call primitives easier for users, and this thread code adds to it. The functions described below form the interface, and can be found in lib/mlib/threads.c.

B.1.1. create_single_stack

SYNOPSIS

```
#include <mlib/threads.h>

void *thread_create_single_stack(size_t size, clist_t *child_cl,
                                 cap_t *retcap);
```

DESCRIPTION

Allocate a new stand-alone stack object of the given *size*. If *child_cl* is non-NULL, a capability to the new stack object is added to the Clist. If *retcap* is non-NULL, a capability to the new stack object is returned through it.

This function may always be called, even if the library has not yet been initialised via a call to *thread_init*.

RETURN VALUE

Returns pointer to the top word on the new stack object, or NULL on error.

B.1.2. init

SYNOPSIS

```
#include <mlib/threads.h>

void thread_init(size_t stack_size,
                 unsigned int stacks_per_object);
```

DESCRIPTION

This function must be called to initialise the library, before any other library functions are used. The parameters control the size and number of stack objects that will be created by the library—*stack_size* is the size of each stack allocated, *stacks_per_object* the number of stacks per Mungi object created. Hence, the library will allocate stack objects of size $stack\_size \times stacks\_per\_object$.

B.1.3. get_stack

SYNOPSIS

```
#include <mlib/threads.h>

void *thread_get_stack(clist_t *child_cl, void **stack_base,
                       struct ptr_list **retelt);
```

DESCRIPTION

Allocates a stack from the pool available, and returns a pointer to the top of it. If *child_cl* is non-NULL, and a new Mungi object is created, adds a read/write capability for the object to the Clist. If *stack_base* is non-NULL, returns through it the base address of the stack. If *retelt* is non-NULL, returns through it a pointer to the chunk allocator list element, which can be reused.

RETURN VALUE

Returns pointer to the top word on the stack, or NULL on error.

B.1.4. get_tcb

SYNOPSIS

```
#include <mlib/threads.h>

struct utcb *thread_get_tcb(void);
```

DESCRIPTION

Allocates a correctly aligned user-level TCB from the pool available, initialises its fields, and sets the thread delete callback function to the address used by the library.

RETURN VALUE

Returns pointer to the user-level TCB structure, or NULL on error.

B.1.5. create_local

SYNOPSIS

```
#include <mlib/threads.h>

struct utcb *thread_create_local(thread_t ip, cap_t param,
                                 bool detached);
```

DESCRIPTION

Creates a new thread in the same APD as the caller. The thread starts execution at address *ip*, and with a parameter of *param*.

The *detached* field controls whether the thread can be waited for using the *thread_wait* function. If true, the thread's exit notification will not be available from *thread_wait*.

RETURN VALUE

Returns pointer to the user-level TCB structure for the new thread, or NULL on error.

B.1.6. create_remote

SYNOPSIS

```
#include <mlib/threads.h>

struct utcb *thread_create_remote(thread_t ip, cap_t param,
                                  bool detached,
                                  const apddesc_t *pd);
```

DESCRIPTION

Creates a new thread in a different APD from the caller, specified by *pd*. The thread starts execution at address *ip*, and with a parameter of *param*.

The *detached* field controls whether the thread can be waited for using the *thread_wait* function. If true, the thread's exit notification will not be available from *thread_wait*.

RETURN VALUE

Returns pointer to the user-level TCB structure for the new thread, or NULL on error.

B.1.7. get_callback

SYNOPSIS

```
#include <mlib/threads.h>

pdx_t thread_get_callback(cap_t *retcap);
```

DESCRIPTION

Returns the address of the callback PDX function used by the library. If *retcap* is non-NULL, returns through it the capability to call this function. The returned capability must be in the APD of any threads using this service.

B.1.8. wait

SYNOPSIS

```
#include <mlib/threads.h>

struct utcb *thread_wait(int *status);
```

DESCRIPTION

Blocks until a thread created using the library exits. If *status* is non-NULL, returns the thread's exit status through it.

RETURN VALUE

Returns the user-level TCB pointer of the exiting thread.

B.1.9. cleanup

SYNOPSIS

```
#include <mlib/threads.h>

void thread_cleanup(void);
```

DESCRIPTION

Frees all objects allocated by the library. Should only be called once all managed threads have exited.

B.2. Chunk allocator

To simplify implementation of the thread library described in the previous section, which must allocate and maintain free stack and user-level TCB objects, a simple memory allocator was written. This so-called chunk allocator manages contiguous memory regions of uniformly-sized chunks, more than one of which are contained within a single Mungi object.

B.2.1. init

SYNOPSIS

```
#include <mlib/chunk_alloc.h>

struct chunk_alloc *chunk_init(size_t chunk_size,
                               unsigned int chunks_per_object);
```

DESCRIPTION

Initialises a new chunk allocator, which will manage chunks of size *chunk_size*, and create one Mungi object for every *chunks_per_object* chunks.

RETURN VALUE

Returns a pointer to the new chunk allocator structure, or NULL on error.

B.2.2. get

SYNOPSIS

```
#include <mlib/chunk_alloc.h>

struct ptr_list *chunk_get(struct chunk_alloc *chunks,
                           cap_t *retcap, cap_t *newcap);
```

DESCRIPTION

Allocates a new chunk from the allocator *chunks*. If *retcap* is non-NULL, returns a capability to the object containing the chunk through it. If *newcap* is non-NULL, and a new object must be allocated to fulfil the request, returns a capability to that object through it.

RETURN VALUE

Returns a pointer to a *ptr_list* structure. This structure has a field named *data*, which is a pointer to the beginning of the chunk. The structure pointer should be freed, or subsequently returned to the allocator using the *chunk_release* function.

B.2.3. release

SYNOPSIS

```
#include <mlib/chunk_alloc.h>

void chunk_release(struct chunk_alloc *chunks,
                   struct ptr_list *chunk);
```

DESCRIPTION

Returns a chunk previously allocated with *chunk_get* to the pool of chunks.

B.2.4. release_base

SYNOPSIS

```
#include <mlib/chunk_alloc.h>

void chunk_release_base(struct chunk_alloc *chunks,
                        void *chunk);
```

DESCRIPTION

Returns a chunk whose base address is *chunk* to the pool of chunks. This can be used instead of *chunk_release*, when the returned pointer is not available, but may be more costly since it requires *malloc* to be used.

B.2.5. free_all

SYNOPSIS

```
#include <mlib/chunk_alloc.h>

void chunk_free_all(struct chunk_alloc *chunks);
```

DESCRIPTION

Frees all memory used by the chunk allocator and delete any created objects. Should only be called once all chunks are no longer in use.

Bibliography

[ABLL92] T. Anderson, B. Bershad, E. Lazoswka, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. http://www.cs.berkeley.edu/~brewer/cs262/Scheduler.pdf.

[AH98] Alan Au and Gernot Heiser. *L4 User Manual*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, January 1998. UNSW-CSE-TR-9801. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[Bla94] Richard John Black. *Explicit Network Scheduling*. PhD thesis, Churchill College, University of Cambridge, December 1994. http://www.cl.cam.ac.uk/ftp/papers/reports/TR361-rjb17-explicit-network-scheduling.ps.gz.

[Car96] Mark Carnes. Sombrero system interface, December 1996. ftp://ftp.eas.asu.edu/pub/cse/sasos/syscall.pdf.

[CH02] Kingsley Cheung and Gernot Heiser. A resource management framework for priority-based physical-memory allocation. In J Morris and F Lai, editors, *Proceedings of the 7th Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, Monash University, Melbourne, Australia, January 2002.

[Cha95] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, University of Washington, 1995. http://www.cs.duke.edu/~chase/research/thesis.ps.

[Che01] Kingsley Cheung. The Mungi resource management model and physical memory control mechanisms. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, May 2001.

[Com98] Compaq Computer Corp. *Alpha Architecture Handbook Version 4*, 1998. http://www.support.compaq.com/alpha-tools/documentation/current/alpha-archt/alpha-architecture.pdf.

[Cus93] Helen Custer. *Inside Windows-NT*. Microsoft, 1993.

[Cut98] Daniel Cutting. Protection domains and threads in Mungi. BSc(Hons) thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1998.

[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.

[DM02] Ulrich Drepper and Ingo Molnar. The new native POSIX thread library for Linux, October 2002. http://people.redhat.com/drepper/nptl-design.pdf.

[DoD86] US Department of Defence. *Trusted Computer System Evaluation Criteria*, 1986. DoD 5200.28-STD.

[Dre02a] Ulrich Drepper. Design of the new GNU thread library, April 2002. http://people.redhat.com/drepper/glibcthreads.html.

[Dre02b] Ulrich Drepper. ELF handling for thread-local storage, October 2002. http://people.redhat.com/drepper/tls.pdf.

[Dyb98] Scott Dybiec. *Patched MIT Pthreads Documentation*, December 1998. http://www.humanfactor.com/pthreads/mit-pthreads.html.

[Edw00] Antony Edwards. A component architecture for system extensibility. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2000. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[EHL99] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual: MIPS R4x00, Version 1.11, Kernel Version 79*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, May 1999. Available from http://www.cse.unsw.edu.au/~disy/L4/.

[EKB+92] Joseph R. Eykholt, Steve R. Kleiman, Steve Barton, Roger Faulkner, Anil Shivalingiah, Mark Smith, Dan Stein, Jim Voll, Mary Weeks, and Dock Williams. Beyond multiprocessing—multithreading the SunOS kernel. In *Proceedings of the 1992 Summer USENIX Technical Conference*, pages 11–18, San Antontio, TX, USA, 1992.

[Eng01] Ralf S. Engelschall. *GNU Pth—The GNU Portable Threads Manual*, February 2001. http://www.gnu.org/software/pth/.

[FCL93] Michael J. Feeley, Jeffrey S. Chase, and Edward D. Lazowska. User-level threads and interprocess communication. Technical report 93-02-03, Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195, USA, 1993. ftp://ftp.cs.washington.edu/tr/1993/02/UW-CSE-93-02-03.PS.Z.

[Gal00] Emmanuel Galanos. High performance serving on Mungi. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2000.

[Gre99] Michael B. Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, July 1999. http://www.cis.upenn.edu/~mbgreen/thesis/thesis.ps.gz.

[Har85] Norman Hardy. KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985. http://www.cis.upenn.edu/~KeyKOS/OSRpaper.ps.gz.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[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 (ACSC)*, 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.

[Hoh02] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, Dresden University of Technology, July 2002. http://os.inf.tu-dresden.de/~hohmuth/prj/phd.pdf.

[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/.

[IBM01a] IBM K42 Team. *K42 Overview*, October 2001. http://www.research.ibm.com/K42/white-papers/Overview.ps.gz.

[IBM01b] IBM K42 Team. *Scheduling in K42*, October 2001. http://www.research.ibm.com/K42/white-papers/Scheduling.ps.gz.

[L4K01] L4Ka Team. *L4* eXperimental *Kernel Reference Manual*. University of Karlsruhe, version 4-x.2 edition, October 2001. http://l4ka.org/projects/version4/l4-x2.pdf.

[Lam95] Fondy F.Y. Lam. Resource accounting in Mungi. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, 1995.

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

[Les02] Ben Leslie. Mungi device drivers. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2002.

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

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[LW01] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001.

[MSS+93] Kevin Murray, Ashley Saulsbury, Tom Stiemerling, Tim Wilkinson, Paul Kelly, and Peter Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proceedings of the 2nd USENIX Workshop on Microkernels and other Kernel Architectures*, pages 31–43, September 1993.

[Mun02a] Mungi source code, version 1.2. Available from http://www.mungi.org/src/, October 2002.

[Mun02b] School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia. *The Mungi Kernel API, Version 1.2*, August 2002. http://mungi.org/src/.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, 1996.

[NBH02] Brett Nash, Ewa Z. Bem, and Gernot Heiser. Orthogonal persistence in the Mungi operating system. Submitted to the 26th Australasian Computer Science Conference (ACSC), September 2002.

[Nem00] University of Cambridge Computer Laboratory. *The Nemesis System Documentation*, 2nd edition, January 2000. http://nemesis.sourceforge.net/manual.ps.gz.

[POS90] *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1990. IEEE Std 1003.1-1990, ISO/IEC 9945-1:1990.

[POS95] *Portable Operating System Interface (POSIX)—Part 1c: Threads Extension*, 1995. IEEE Std 1003.1c-1995, ISO/IEC 9945-1:1996.

[PWH01] Daniel Potts, Simon Winwood, and Gernot Heiser. *L4 Reference Manual: Alpha 21x64*. University of NSW, Sydney 2052, Australia, March 2001. UNSW-CSE-TR-0104. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[RF97] Dickon Reed and Robin Fairbairns. *Nemesis Kernel Overview*, May 1997. http://www.cl.cam.ac.uk/Research/SRG/netos/old-projects/pegasus/publications/overview.ps.gz.

[Sam97] Samsung Electronics. *21164 Alpha Microprocessor Hardware Reference Manual*, 1997.

[Sch02] Gavin Schokman. Mapping the K42 API to L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 2002.

[SGI02] Silicon Graphics, Inc. *State Threads Library Documentation*, 2002. http://state-threads.sourceforge.net/docs/.

[Sha98] Jonathan S. Shapiro. *EROS Object Reference Manual*, 1998. http://www.eros-os.org/devel/ObRef/Cover.html.

[Sha99] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999. http://www.eros-os.org/papers/shap-thesis.ps.

[SMF96] Alan C. Skousen, Donald S. Miller, and Ronald G. Feigen. The Sombrero operating system: An operating system for a distributed single very large address space — general introduction. Technical Report TR-96-005, Computer Science and Engineering Department, Arizona State University, Tempe, AZ 85287-5406, April 1996.

[Sol97] Frank G. Soltis. *Inside the AS/400, Featuring the AS/400e series*. 29th Street Press, Loveland, CO, USA, 1997.

[SPE95] Standard Performance Evaluation Corporation, Manassas, VA, USA. *SPECint95 Benchmark*, August 1995.

[SPE00] SPECint95 benchmark results, September 2000. http://www.spec.org/osg/cpu95/results/cint95.html.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on OS Principles (SOSP)*, pages 170–185, Charleston, SC, USA, December 1999. http://www.eros-os.org/papers/sosp99-eros-preprint.ps.

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

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

[Voc98] Jerry Vochteloo. *Design, Implementation and Performance of Protection in the Mungi Single-Address-Space Operating System*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, July 1998. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[Win00] Simon Winwood. Flexible scheduling mechanisms in L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2000. Available from http://www.cse.unsw.edu.au/~disy/papers/.

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

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, USA, November 1994.

[YL98] K. K. Yue and D. J. Lilja. Dynamic processor allocation with the Solaris operating system. In *Proceedings of the International Parallel Processing Symposium*, pages 392–397, Orlando, Florida, April 1998.