

THE UNIVERSITY OF
NEW SOUTH WALES



SCHOOL OF COMPUTER SCIENCE
& ENGINEERING

**A Formal Model of the L4 μ -kernel API Using the
B Method**

Rafal Kolanski

rkol757@cse.unsw.edu.au

Bachelor of Engineering
(Software Engineering)

November 2004

Supervisor: Ken Robinson

Assessor: Kevin Elphinstone

Abstract

The increasing dependence of modern society on information systems increases the need for secure operating system kernels. Formal methods offer a way to achieve such high-level security, however they require a significant investment of time, and are not well suited for large kernels such as that of Linux. μ -kernels offer an alternative which is both elegant, and possibly small enough to make formal specification tractable. The L4 Pilot project aims to investigate various approaches towards the formalisation of the L4 μ -kernel and evaluating its feasibility.

In order to make formal verification of a system possible its behaviour, functionality and external interface must be clearly mapped out and understood. This thesis concerns the creation of a formal model of the L4 “Pistachio” API using the B Method.

Contents

1	Introduction	1
1.1	Aims	1
1.2	Operating System Kernels at a Glance	1
1.3	μ -kernels	2
1.4	L4 Overview	2
1.5	The B Method at a Glance	3
1.6	Motivation: APIs, Why We Need Them and Why Bother?	3
2	Introduction to L4 Internals	5
2.1	Address Spaces	5
2.2	Kernel Interface Page	5
2.3	Threads	6
2.3.1	Special Threads	6
2.3.2	Thread Identifiers	8
2.3.3	Thread States	9
2.3.4	Pagers	10
2.3.5	Schedulers	10
2.4	Scheduling	10
2.5	IPC	11
2.5.1	Memory Donation and Leasing Using IPC	11
2.6	System Calls at a Glance	12
3	Overview of the B Method	15

4	Related Work	19
4.1	Early Efforts	19
4.2	Modern Object-Level System Specification	19
4.3	Going Backwards	20
4.4	Top-only	20
4.5	Top-Down	21
4.6	Implications for This Thesis	21
5	Results	23
5.1	Design Methodology	23
5.2	Types and Constants	28
5.2.1	Machine KernelInformation	28
5.2.2	Machine AddressSpaceCtx	29
5.2.3	Machine ThreadStateCtx	30
5.2.4	Machine ThreadCtx	31
5.2.5	Machine ThreadIdCtx	32
5.2.6	Machine TimeoutCtx	32
5.2.7	Machine FpageCtx	33
5.2.8	Machine ErrorCtx	34
5.3	Address Spaces	34
5.3.1	Machine AddressSpace	34
5.4	Threads	36
5.4.1	Machine Thread	37
5.4.2	Machine Thread: Operations	44
5.5	IPC	51
5.5.1	Machine IpcCore	51
5.5.2	Machine IpcBase	51
5.5.3	Machine IpcBase: Operations	55
5.6	API	64
5.6.1	Machine WeakSyscall	64
5.6.2	Machine API	68

6 Discussion and Critique	69
7 Conclusion	73
7.1 Future Work	73
7.1.1 Complete Proof of Consistency	73
7.1.2 Verifying Correctness	73
7.1.3 The L4 Pilot Project	74
7.1.4 Further Research Using the B Method	74
A Final B Specification	77
A.1 API	77
A.2 WeakSyscall	84
A.3 IpcBase	90
A.4 IpcCore	103
A.5 Thread	104
A.6 AddressSpace	113
A.7 KernelInformation	115
A.8 ThreadCtx	116
A.9 ThreadIdCtx	117
A.10 ThreadStateCtx	118
A.11 TimeoutCtx	119
A.12 AddressSpaceCtx	120
A.13 FpageCtx	121
A.14 ErrorCtx	122
A.15 Bool_TYPE	123

List of Figures

5.1	Inclusion diagram for the B development.	26
5.2	A very simplified diagram of possible state transitions.	30
5.3	Possible state transitions in the model and operations which cause them. . . .	63
5.4	The receive phase only section of WeakIpc	65
5.5	The send phase with optional receive phase section of WeakIpc	66

Chapter 1

Introduction

1.1 Aims

To:

- produce a formal model of the L4 API based on L4 “Pistachio” using the B Method;
- gain a very thorough understanding of the L4 μ -kernel from a functional perspective;
- identify potential faults and shortcomings that may be useful to current implementers, and any future formal verification of L4;
- create a solid starting point for the continuation of the L4 Pilot project.

1.2 Operating System Kernels at a Glance

When referring to a “kernel” of an operating system we usually think of the core of the system, which the rest is entirely dependent on. This is an apt simplification.

The kernel [20] represents the part of the operating system that has complete access to everything inside a computer at the lowest level permitted by hardware. While it can grant this privilege onto other code fragments (such as device drivers), the *responsibility* for interaction with the hardware lies with the kernel. At the same time, the kernel must allow programs running within the operating system to utilise the hardware.

Needless to say, this responsibility is a very large cause for concern from a security standpoint. For example, today’s hard drives have no concept of users on the system and who has access to what. This implies that should any program gain full access to hardware for any reason, it can access all data on the hard drive. When a user level program achieves such access rights by utilising a fault in the kernel, it is known as a *local exploit*.

The process of design and development of operating systems is an eternal struggle between flexibility and control.

1.3 μ -kernels

Given the above security concerns, much work goes into making sure such breaches do not happen. However, most modern operating systems use monolithic kernels, where a large amount of functionality, device drivers, networking, etc. are included. This means there is a lot of code that runs with full privileges, written by many people over a long period of time. Indeed, in 1999, the Linux kernel had approximately 350 people working on it [15]! While it can be argued that the large amount of people looking and working on the kernel are more likely to discover possibilities of exploitation, the sheer amount of code they have to secure, while hundreds of *other* people work on it is terrifying. In fact, the current Linux kernel source code is approximately 30Mb [11]!

These are the very concerns that drove the computer science community into developing a concept called μ -kernels [14]. The basic aim of a μ -kernel is to decrease the amount of code running with full privileges to the bare minimum. The smaller (relative to monolithic kernels) amount of code makes securing it orders of magnitude easier. The price is that abstractions must be envisioned for hardware-interfacing programming, such as device drivers. This takes the form of increased complexity, decreased performance, or worse, both. Additionally, just what is the *bare minimum* remains a hotly debated topic [13].

1.4 L4 Overview

The L4 μ -kernel remains true to the initial vision of supplying only that which is essential, while incorporating many optimisations discovered on the long path of research in μ -kernels.

It essentially provides the following [12]:

- Address spaces
- Threads (privileged, kernel and user)
- Fast, Synchronous Inter-Process Communication (IPC)
- Basic scheduling
- Memory ownership management (granting, mapping)
- Core machine resource management (such as Floating Point Units)
- Core machine control (CPU voltages etc.)
- Symmetric Multi-Processor (SMP) support

Experiments running Linux on L4 have shown only a 4% performance penalty over normal Linux kernels [6], concluding that L4 is indeed fast enough for mainstream acceptance.

1.5 The B Method at a Glance

Various formal methods exist at present. Essentially, they allow one to express the behaviour of a program (or system) in abstract, mathematical terms, and therefore allow proof that the model is consistent (not self-contradicting). These models can then be implemented with confidence. Alternatively, some attempt to prove things using already existing code as input. However, no method, formal or otherwise, can *guarantee correctness*, i.e. that the model does *exactly* what the creators want it to do. For that, a human is needed (or a mind-reading computer).

The B Method, developed by Jean-Raymond Abrial [1], is a system of formal development from the initial high-level specification all the way to implementation via a process called *refinement*. It is one of the few formal methods to do so. The idea is to abstract away the implementation and concentrate on pure functional requirements at the top level. Then, with each refinement step, provide more information on how exactly the system fulfils those requirements, until enough information exists to implement the system. Each refinement step requires proof of consistency with the previous step.

Having only functionality at the top level allows one to easily evaluate what the system will be doing without getting bound up in implementation constraints (and thus make the verification of *correctness* much easier). At the same time, the refinement path towards implementation can be modified without *any* change to the abstract model.

Furthermore, by utilising the facilities contained within the B Toolkit [2] it is possible to verify the correctness of the top-level specification by *animating* it. In this mode, the user becomes the implementation of all non-deterministic or undefined operations/sets. For example: an operation which factors a composite number can be defined as returning any two numbers which, when multiplied yield the supplied number. The Animator will ask you for the two numbers, then check whether they are indeed factors, then the animation will proceed. Once the system behaves the way one wants it to, one can then safely proceed to refinement and implementation.

It is important to note the difference between using the B Method and systems such as Isabelle/HOL [17]. While the latter allows a completely free-form way of specification, the B Method provides a ready-made framework. This means that starting a development with B is much easier, but comes at the price of decreased flexibility.

1.6 Motivation: APIs, Why We Need Them and Why Bother?

The Application Programming Interface (API) is a crucial part of any system, through which programmers may control and interact with it. Specifically, it is an abstraction of the inner workings of the system (which allows people who may have no interest in how *exactly* the system works to use it). For an operating system kernel, it is yet more. Since the kernel runs in privileged mode and programs written by external programmers generally do not, it is also the only place through which unprivileged data is transferred into privileged mode. Most security checks on such data must therefore occur at the API level, so that internal system processes can be performed with confidence. For example: there will be an API call for a

thread to kill another, but will have check if the thread may indeed kill others (and that the thread does not kill itself), which will then call an internal operation which will perform the actual kill. The person writing the latter does not need to concern him/herself with security.

Given its importance, one would expect any operating systems verification project to at least give some thought to the API. As seen in *Related Work*, this isn't always the case. The API represents a model of the system as seen by a process being executed and, as such, what the system actually *does* (whereas the implementation is responsible for *how* it does it).

I believe it is significantly easier to implement a system given a formal model of its behaviour than vice versa. One would be hard pressed to prove anything about what the system is doing without first defining what it is the system should do. Nonetheless, people still try [10]. While this allows simple assertions to be made about the system, it does not make verifying *correctness* much easier than just reading the implementation.

It is also very difficult to create an abstract model of an API which already exists, since much of the time, its implementation has evolved more than it has been designed.

Another significant function of an API is a central point on which development of a system is based. Any model of the API can be referred to by people implementing or modifying the system. However, if the behaviour outlined by the API is self-contradictory, the system may well end up acting strangely or not at all. This is where a formal model of the API comes in.

The motivation for this thesis is to provide a relatively simple formal model for the L4 "Pistachio" μ -kernel. Presently, the API resides in a Reference Manual [12] and in the implementation. While the Reference Manual contains a documentation of all system calls, it is insufficient to create a model of the system, whereas the implementation is very complex, barely documented and highly optimised (with nearly every possible shortcut utilised); it cannot be used as a model.

A top-level model created in B should be quite small, and therefore easy to check for correctness, while the ability to animate it will make the task even easier. It can be very useful to those working on the L4 implementation and the future of the L4 Pilot project. Also, a new Security API is being worked on for L4. Comparing the new API with the current model may also provide useful information on how the new Security API might be implemented.

Chapter 2

Introduction to L4 Internals

This chapter describes some of the internal mechanisms in L4, as well as expanding upon the very brief overview of L4 provided in the introduction (page 1.4).

2.1 Address Spaces

In L4, address spaces do not have identifiers, other than their information structures being accessible by pointers inside the kernel memory area. When dealing with user mode applications, address spaces are identified by passing in the thread identifier of the thread whose address space is the one desired.

Each address space has a User Thread Control Block (UTCB) area to store user-mode details of threads contained inside it.

Since there is no way to specify an address space without a thread, the process of creating address spaces is merged into that of creating threads. This means creation of an empty address space is not possible.

Apart from address spaces reserved for special threads (see next section), there is an extra address space for internal kernel threads known as the *kernel address space*.

2.2 Kernel Interface Page

In L4, all information particular to the current instance of the kernel is stored in the Kernel Interface Page. The KIP must be mapped into each thread's address space and is read only.

Information provided includes how many threads are permitted in the system, how many interrupt threads there are, etc.

2.3 Threads

The number of threads is predetermined at system start up. The kernel reserves a certain number of internal structures for storing the kernel-level details of each thread. These structures are Kernel Thread Control Blocks (KTCBs) and hold information that no thread should not have direct access to (such as scheduling priority, thread identifier and thread state).

Data that a thread has on itself (which the kernel does not directly use to run) is provided in UTCBs stored in the address space's UTCB area. This includes extra call and return parameters to/from system calls and general information such as the thread's current identifier (which the thread can modify, but which will not affect the functioning of the kernel at all, only give the thread misleading information it is someone else). The size of the UTCB area limits the number of threads possible in each address space.

2.3.1 Special Threads

Sigma0

Sigma0 is the default system pager. Its job is to take up all possible non-kernel memory on start up, and give it out to threads as it is requested. Sigma0 is *not* a memory manager, and does not accept any form of request to take back (free) the memory that has once been given out. When other special threads load during the kernel loading procedure, memory is taken from sigma0. After this is done, a memory manager should take the remaining memory from sigma0 and manage it appropriately. Sigma0 gets its own address space, and is a *privileged thread*.

The Root Server

This is where an operating system start-up begins. After the root task (the kernel loader and setup subsystem) finishes loading the kernel, sigma0 and the root server have been created. The scheduler is then invoked and the root task loses control forever. This means that sigma0 and the root server will most likely be the only two threads running in the system (there can be more root servers if the root task is modified), and since sigma0 already has a predefined function, operating system start-up is the role of the root server. The root server also gets its own address space and is a *privileged thread*.

Sigma1

The sigma1 thread was originally designed and introduced into L4 as a method to manage task persistency (that is freezing tasks, saving them to disk, then being able to load and resume them again). The problem of task persistency has not been a key feature of L4 and has since been mostly deprecated. Many people using L4 do not know what sigma1 really does. Like sigma0, it is a privileged thread and also gets its own address space.

Privileged Threads

Having defined three privileged threads, knowledge of what it is they are privileged to do would be appropriate:

- Creation, deletion and modification of threads
- Set up and initialisation of address spaces
- Modification of machine settings (such as processor frequency) on architectures that support them.

A thread is defined to be privileged if it resides in the *sigma0*, *root server* or *sigma1* address space. This is how the initial root server can create more privileged root servers to perform various tasks (such as a dedicated thread manager).

The *kernel address space* also holds privileged threads, though this isn't listed in the reference manual. The exact comment in the source code [5] is “do not allow user to mess with kernel threads”. Considering them privileged seems to have no adverse effect, only the check is different.

Interrupt Threads

In ordinary operating systems, when an interrupt occurs it is directed to an interrupt handler which immediately invokes the appropriate function somewhere within the kernel. In a μ -kernel, the handler of the interrupt has to be implementable by the user, and the user can only work in user mode. This means that handlers will be in threads running outside of the kernel and some form of delivery must be devised.

The method L4 uses is to use the IPC abstraction for delivery of interrupt notifications. Once the interrupt is handled, the handler has to notify the kernel that to re-enable the interrupt (which again must happen through IPC). In order to do that, the notifications must be sent *from* somewhere that the handler can reply to, i.e. another thread.

This implies the existence of interrupts as threads, and indeed, this is the abstraction used. There are some key differences however. The interrupt is an actual pin on a chip, and so it cannot actually *run* as a normal thread would. This means an interrupt thread's thread states (see page 9) are more convoluted than usual. The IPC abstraction works because the kernel captures messages targeted at interrupt threads and performs its own internal actions as a result. It also sends an IPC on behalf of the interrupt thread when an interrupt occurs.

Interrupt threads are internal to the system and reside in the kernel address space. While they are defined to always exist, their UTCBs are created lazily, which means that the concept of creating an interrupt thread exists.

They do not have schedulers (since they do not run and hence have no scheduling parameters). Their pager is called a *handler* and it is the thread that associates a thread with an interrupt it must handle. To deactivate an interrupt, the handler is set to be the interrupt thread itself.

2.3.2 Thread Identifiers

Due to L4's number one priority being efficiency, the thread identifier system is very interesting indeed.

Firstly, the possible thread identifiers are divided into *local* and *global*. Global identifiers allow a thread to select another in the system, while local identifiers work only in the same address space.

Global identifiers

Global identifiers are further divided into the *thread number* and the *version*. Here is where the first optimisation occurs: the thread number is actually an index into the KTCB table. This means that looking up a thread is nearly instantaneous.

There are thread numbers reserved for various tasks inside the system:

- $[0, SystemBase)$: interrupt threads
- $[SystemBase, UserBase)$: kernel internal threads
- $[UserBase, maximum)$: user threads

Since thread numbers and KTCBs are the same unit of allocation, interesting properties arise when a thread ends up pointing to another thread that has been deleted (for example, when its scheduler is deleted). If the thread numbering system was left to itself, the first thread that gets the same KTCB as the old deleted thread would immediately be accepted as the old thread (meaning the new thread could still be the scheduler for the thread the old one scheduled).

This behaviour may or may not be desired by the person deleting the thread. Therefore, a user-controlled *version* field was introduced. To see if two identifiers are the same, the kernel checks the thread number and the version. This way, if the above behaviour was desired, the version number can remain the same. On the other hand if the new thread is to be different than the old one, the version field must be modified (by incrementing it by one, for example). This has the capacity to overflow eventually.

The interrupt threads always have a version of 1. The initial privileged threads also start with a version of 1.

Local identifiers

These can be identified by the fact their lowest six bits are 0. The rest of the bits are naturally a reference to the thread's UTCB location, again allowing near-instantaneous lookup of threads in the local address space.

The reason for their existence is further optimisation. Since switching between threads in different address spaces is a time-consuming operation, when switching to another local thread

the scheduling system may be bypassed. Since the fast IPC path between local threads (see page 11) uses this type of switching, it is another way to optimise IPC, the core component of L4.

Since the identifier must still be checked if its a local identifier, why not just check if a thread is in the same address space given a global identifier? Why are local identifiers even needed? The reason is quite likely to be the ability to restrict certain functionality in L4 to local (or global) threads only allowing it to be more efficient. Also, checking once is faster than checking twice. Finally, being able to specify *anylocalthread* for some operations is a useful addition.

Special identifiers

Three identifiers are reserved for special tasks:

- *nilthread* represents “no thread”. This has varied uses, from specifying a setting is to be left unmodified, through special meanings when invoking certain system calls, to performing send-only or receive-only IPC;
- *anythread* represents any thread in the system (for example, thread t_1 is willing to accept a message from any thread in the system);
- *anylocalthread* is like *anythread* but signifies local threads only

2.3.3 Thread States

Ignoring multiple-cpu states, threads may assume the following states in L4.

First is the state of non-existence, which means a thread’s KTCB is not being used. This is identified by an *aborted* state and the address space setting in the KTCB equal to NULL.

Next, the inactive state, when a thread is created but not yet active. This is identified by the *aborted* state and a non-null address space attribute in the KTCB. From here, the thread may be activated or deleted.

When a thread is activated or created active, it immediately starts waiting forever for an IPC from its pager (in order to allocate the necessary memory to start executing). The state is, as the name suggests *waiting_forever*.

When the IPC is received the thread may begin fully interacting with the system. Its state becomes *running*. In other operating systems this state is often referred to as *ready*. This does not necessarily mean the thread is running, only that it can. It is up to the scheduler to decide which threads are running.

The *halted* state is special. A *halted* thread must not be scheduled or execute in user mode. This includes switching into it without the scheduler being involved. Since the thread may be resumed, the previous states had to have been saved. Halting has no effect on IPC delivery.

For interrupt threads, halting means something completely different. An interrupt is *enabled* if the thread is *halted*. It is *disabled* if the thread is *not halted*.

The other states (*polling* and *waiting_timeout*) are discussed in IPC on page 11.

2.3.4 Pagers

When a running thread triggers a page fault (memory access to an area that it does have access to but whose data is currently somewhere else, e.g. on disk or unallocated) the kernel must resolve the issue. However, a μ -kernel does not have built-in memory management facilities and relies on the user to implement this to their liking. How then does the fault get resolved?

The answer is that every thread in L4 has a pager attribute containing a global thread identifier (which most of the time implies it also has a pager). When a page fault occurs, the kernel dispatches a page fault IPC on behalf of the thread to the thread with that identifier. If the thread exists, it gets the message and may resolve the memory issue. If it does not exist, the faulting thread is suspended or killed.

The reason a thread's pager may not exist is again efficiency. It is very slow to check every thread in the system to see if the thread we are deleting just happens to be its pager. It is better not to check and deal with the problem when a page fault occurs, or even better, to assume the user will prevent this situation from arising.

2.3.5 Schedulers

A very confusing thing about L4 is that schedulers and the SCHEDULE system call do not actually schedule anything.

Each thread in the system has a scheduler attribute. The same issues occur as with pagers, but the implications are different.

A scheduler is permitted invoke SCHEDULE on a thread whose scheduler attribute is its own thread identifier.

Scheduling attributes include the thread's priority, which *is* used for scheduling, but not directly.

2.4 Scheduling

L4 possesses an internal priority-based scheduler which bases its decisions on the parameters defined using the SCHEDULE system call.

The internal scheduler is *not a thread*. It is simply a function which picks the next thread to execute.

In order for a user to specify his/her own scheduler, the only way to do so is to set the priorities of all the threads in the system except the scheduler to 0. Then any thread switching or pre-empting condition will end up with the scheduler being chosen as the next thread to execute. The scheduler can then yield donating its time slice to another thread (using the

THREADSWITCH system call).

2.5 IPC

The Inter-Process Communication, or IPC is the core component of L4. Nearly all aspects of the system are abstracted by IPC when possible, this includes donation and leasing of memory to other processes.

IPC is *synchronous*. This means that for a successful transfer to occur, the sender must be sending or polling while the receiver is waiting (or running, if the sender is polling). What is more the receiver must be waiting *for* the sender for this to work. The special thread identifiers *anythread* and *anylocalthread* also declare who a thread is willing to receive from.

When a receiver is willing to receive but there are no candidates, it goes into a *waiting* state, which is either *waiting_forever* or *waiting_timeout* depending on whether the chosen waiting timeout is infinite.

When the sender tries sending an IPC but the receiver is not ready or currently willing to receive, it goes into a *polling* state and is placed in the receivers incoming queue. There is only one *polling* state, regardless of the timeout. Polling may include an additional *receive phase*, which means that should the send succeed, the kernel immediately places it into a *waiting* state with the receive phase parameters.

The *fast* IPC path in L4 occurs when the sender encounters a waiting receiver. The IPC occurs immediately and the sender's remaining time slice is donated to the receiver. In this manner, a thread can decrease the time it has to wait for a reply (since the scheduler would not necessarily choose the receiver as the next thread to execute). Sometimes, a thread can send an IPC, switch to the receiver and get a reply when the receiver switches back during the same time slice.

The actual data that gets transferred between threads is *virtual registers* (which may or may not be implemented in hardware) and *items*. The sender states how many of these registers it wishes to transfer. It can also send map items, grant items and string items.

Mapping and granting is covered in 2.5.1.

String items, as their name suggest, are used to transfer strings from one thread's buffer registers into another's. Since the size of strings is dynamic, various issues with page faults on either sides and inside the kernel itself arise. String items are not implemented on all architectures and their usefulness is often disputed.

2.5.1 Memory Donation and Leasing Using IPC

As part of IPC, the sender may send the receiver *map* and *grant items*. These items, whose exact structure is defined by an Fpage [12, section 4.1] determine leasing and granting of memory to the receiver and is performed during the IPC by the kernel.

Granting means that the sender completely gives away the memory to the receiver, and is no

longer able to access it. The receiver owns it and is responsible for its eventual deallocation.

Mapping allows the receiver to access a memory area (read and write) but not own it. This means the privilege can be later revoked.

2.6 System Calls at a Glance

KernelInterface

Tell a thread where the Kernel Interface Page is in the current address space.

ExchangeRegisters

May be used by any thread on another in the same address space. Allows modifying existing threads (pager, registers and handle) and also activating inactive ones.

ThreadControl

Only for privileged threads. Create, activate, modify or delete a thread based on parameters. Activation occurs when a valid pager is supplied. The scheduler is also set by this operation.

SystemClock

Obtain the current clock value (an internal counter value in μs).

ThreadSwitch

Stop execution of this thread and donate the remainder of the time slice to another one, or just yield completely.

Schedule

Change the scheduling parameters for a thread (see page 2.3.5). Must be the thread's scheduler.

Unmap

Revoke a mapping given to another thread as part of IPC (see page 2.5.1).

SpaceControl

Initialise an address space. Privileged threads only.

Ipc

Participate in Inter-Process Communication (see page 2.5)

ProcessorControl

Privileged threads only. Change CPU settings. Availability of settings varies with architecture.

MemoryControl

Used to set page attributes.

Chapter 3

Overview of the B Method

While a complete description of the B Method is outside of the scope of this thesis, I will attempt to give an overview for the benefit of the reader.

Development within the B Method is based on three levels:

Abstract Machines Units of isolation, in themselves layers. They represent the base definition of what the program components will do, i.e. the minimal functionality that satisfies the program requirements (as dictated by a client or programmer). They consist of:

- **CONSTRAINTS** — restrictions on parameters passed into a machine
- **DEFINITIONS** — purely syntactic translations. They use a feature of B called *jokers*. Any single-letter token counts as a joker and represents any set of tokens. For example:

$$\text{isFull}(x) == \text{bool}(x \in \text{full})$$

will define a function-like construct whose value is the boolean whether the parameter belongs to the set “full”. This feature is similar to `#define` in C and C++, but one definition cannot use another one within the same machine.

- **SETS** — abstract sets (such as `USERNAME` as an abstraction of all possible user names allowed in the system) and enumerations
- **CONSTANTS** — these may be anything that is not dependant on variables. Constants may be members of sets, sets, or functions (which are, in fact, also sets).
- **PROPERTIES** — define restrictions on sets and constants, which for the latter defines what they are.
- **VARIABLES** — a list of variables separated by commas. What these variables represent, their types, properties and relationships between them are contained in the invariant.
- **INVARIANT** — this is a logical assertion about the system which should always hold (a “safety constraint”), but, more importantly, defines what the machine really *is*. Within it, all variables types, properties, and relationships (including between those in this and

other machines) are defined. It is also the most difficult part of the specification to produce, since the invariant must be strong enough to encapsulate information about the system, and cannot contradict itself.

- **ASSERTIONS** — “facts” that derive from the invariant. They are mostly used to make proof easier.
- **INITIALISATION** — When the “machine” starts, not only do variables need to have defined values, their values must not break the invariant. This needs to be proven.
- **OPERATIONS** — Things that can be done with variables. A crude analogy is that of static class methods. At the abstract machine level, only *parallel* composition is allowed, i.e. all statements in the operation (including invoking other operations) occur at the same time; the operation itself is instantaneous. An operation may only invoke operations in other machines, and only when permitted by inter-machine relationships. The key to proving the invariant holds are *preconditions* in front of every operation. Invoking another operation requires that that operations’ preconditions are satisfied, and that any combination of variables satisfying the precondition will preserve the invariant.

In this manner a chain of “sane” states may be traced from the initialisation (which satisfies the invariant) via operations which cannot break it.

The relationship between machines is extremely restricted. A machine can either *include* (full read/write access to everything contained and included in it, plus invoking operations), *use* (full read access to everything in that machine) or *see* (read access only to sets and constants) other machines. When a machine includes another, it chooses (via PROMOTES) which operations of that machine will be visible when *this* one is included into yet another machine. If all operations are to be visible, INCLUDES is replaced by EXTENDS.

Please note that there is only *one* namespace in B and great care must be used in order to avoid collisions.

Refinements are functional redefinitions of machines at a more detailed level than what they are refining. For example, at the abstract machine level, a single-resource manager may simply say “pick any identifier which is not in the set of used identifiers”. A refinement might choose not to use a set at all, since it has no need for ever reclaiming identifiers (e.g. student numbers) and refine the used set down to just one number: the next free id. The refinement will then say “if the next free identifier is less than the number of permitted identifiers, pick that one and increment the next free identifier”. This can be refined again, and again, until things are simple enough to implement.

Refinements also have the capability of *sequential* composition, in which statements happen one after another, not at the same time as before.

In light of this, while refinements are still considered machines, some parts of them are viewed differently. The invariant is no longer an indicator of the valid state, but instead defines the relationship between the variables in the refinement and the refined machine. The operation definitions must exactly mirror the ones of the refined machine. However, preconditions may be different. This is because the refinement may have *more* functionality than what is being

refined. All that we are requested to prove is that all functionality of the operation in the refined machine is indeed implemented in this one, via the invariant relationship. No new operations may be added.

Note that refinements do not have access to other refinements. They are allowed to include other machines, but cannot be dependent on how those machines are refined.

Implementations are the culmination of refinement efforts in B. They use a completely different language, there are no preconditions, and there is no state. All state must derive from other machines. The implementation cannot base itself on information about the refinement or implementation of any other machine, allowing the refinement/implementation process to be independent for each abstract machine. This also means one machine can have many interchangeable implementations.

For a good reference to B syntax, I recommend Ken Robinson's "A Concise Summary of the B Mathematical Toolkit", available from:

<http://www.cse.unsw.edu.au/~cs2110/B-Summary/index.html>

Chapter 4

Related Work

4.1 Early Efforts

One of the first efforts at formal specification of a kernel was that of William R. Bevier [3]. His KIT Project consisted of verifying a simple multitasking system running on a simulated von Neumann architecture machine. In fact, two finite state machines are simulated: one “running” the abstract kernel model, the other is the actual kernel running on the computer. Using a Boyer-Moore Logic prover they then attempt to show the model satisfies the requirements, and then prove an *implements* relationship between the two state machines. Bevier finally descends into proof of machine code.

His progress was certainly very thorough, but one can clearly see that all though the modelling, everything is completely bound to the machine. Registers and bit fields make an early appearance, and it turns out the abstract model is not really abstract. This is compounded by arcane LISP-like syntax, plus the need to define absolutely everything, since there was nothing to base on. The author indicates problems they had with the theorem prover, the near-impossibility of proving the *implements* relationship, trouble with defining queues and various other aspects. One can quickly appreciate the progress in formal methods since then.

4.2 Modern Object-Level System Specification

While he does not specify a kernel of any kind, Yuan Yu [23] presents a way of machine code program proving on a model of a specific microprocessor. He also uses a Boyer-Moore logic prover, but the improvement made by tools in that area are clearly visible.

Yu succeeds in proving the functionality of various small programs and functions, but indicates just how complex the process is. Due to the large state one must work with, proof becomes difficult. Additionally, there is no way to prove the validity of the model of the processor, which consumes a large amount of the document.

While extremely useful, this type of specification is unlikely to be successfully applied to an entire kernel (or μ -kernel) when used on its own. However, it is very difficult to go from a

top-level specification all the way down to source code when it comes to operating systems, since they all require low-level interaction with the machine at some point. One can take things as low as possible, and then place all functions that need to be written in assembler or that cannot be generated into separate files, merely using them in the generated code as ‘magical’ statements that do what is needed. Later, object-level verification such as Yu’s can be used to prove these functions are not magical and indeed do what is expected of them.

4.3 Going Backwards

Another alternative is to attempt to work backwards and prove things about code that has already been written. An example of this is the VFiasco project [10]. Fiasco is an L4-based μ -kernel written in C++. Using a model of x86 hardware and a model of C++, the project will attempt to prove certain things about the μ -kernel, such as guaranteeing all page fault handlers terminate.

At the time of this writing, the VFiasco project seems to have stalled. No major updates have been posted since July 2003. Even if work resumes, they are unlikely to prove anything significant about Fiasco by analysing its source code (which is written in C++, a language notorious for the depth of its notation).

Apart from low-level object verification for specific application domains (such as that presented by Yu [23]), attempts of system proofs starting at the implementation are, in my opinion, doomed to fail due to lack of manpower and availability of computer intelligence. This thesis demonstrates the difficulty of discovering an abstract specification knowing the code base and a few people who know more or less what it is supposed to be doing. Code analysis is even less informed and would require an intelligent machine for it to figure out what the programmer was thinking, which might fail even then.

4.4 Top-only

When it comes to proving anything about operating systems, the majority of projects only approach the problem at the top level of creating an abstract model not directly linked to the implementation (that is, it models what the implementation does, but there is no simple way to derive one from the other automatically). This way they can be concerned with proving that the behaviour of a system is consistent, and leave the verification that the implementation does what the specification says it should do to humans.

For developing and research operating systems, such as EROS, proving that the theory is sound before embarking on the path to implementation (or continuing down it) is extremely valuable. The best implementation of a bad idea cannot possibly result in a good system. For EROS, the primary goal is *confinement* of tasks running within the system. Shapiro and Weber [18] set about proving their model for confinement is consistent. The general idea is to define all the semantics of the operating system, including a concept of *mutation* (if a process modifies resource x , x is *mutated* by the process; if another process reads x , then it too has been *mutated*), and showing that what is mutated by a process is a subset of resources defined

as modifiable by the security policy.

The above relates to a system with more anticipated complexity than the current version of L4 (which does not possess any advanced security architecture). For systems that are much simpler, the entire system may be modelled. An approach similar to the one undertaken by this thesis is that of J.M. Spivey’s work on specifying the kernel of a safety-critical X-ray diagnostic machine [19]. This model is interesting for many reasons. Firstly, the entire kernel is modelled, similar to my situation. Secondly, the author acknowledges that the true reason for performing the specification is to abstract the kernel from its implementation and to *document* it for future reimplementations (possibly on different machines). This indicates what this thesis will most likely be used as. Third, he uses the Z formal notation, which is the B Method’s predecessor. Finally, the model is a success due to finding a flaw in the system that could potentially have caused the X-ray machine to inflict damage. Again, this is an example of a model testing the ideas the system was built on.

The academic approach of inventing a kernel to verify and *then* proceeding with the implementation is utilised to good effect by S. Fowler and A. Wellings [8]. The kernel they devise is to be used as a basis for an “Ada95 runtime support system in a hard real-time environment”. After reviewing their work one can see the great deal of freedom that is available in the specification when there is no implementation to limit its progress. This is also true when the initial ideas that are eventually implemented in the kernel are still available for review and discussion. The approach is the exact opposite of attempting to verify an implementation after it has been completed, and in my opinion is the easiest way for a verified operating system kernel to be produced.

4.5 Top-Down

Since most formal methods do not allow gradual refinement (as seen in the B Method), very few operating system kernel specification attempts have been made in a top down fashion. Those that have been made mainly consist of two levels only: abstract specification and concrete implementation [9], or just abstract specification as mentioned in the previous section [7].

This “methodology” starts out with an abstract specification of the system functionality and proceeds from there. This method is often used to develop a functional abstract specification from the requirements (sometimes independently of any existing implementation) in order to discover flaws and weaknesses in the system [22].

4.6 Implications for This Thesis

Compared to the above, using the B Method presents interesting advantages as well as challenges. The ability to start with a relatively “pure” specification layer and then refine the parts that require it, allows (and requires) great simplification of L4 down to its core components. This should make the specification easier to understand, but its creation will require a very detailed understanding of L4. Due to this, the situation present at the beginning of the

thesis is clearly much more restricted than when creating a brand new kernel [8]. Progress will undoubtedly be slowed by gaps in understanding.

As discussed earlier, developing any implementation from a high-level formal model poses difficulties even for PhD students (since it often requires a formal model of the architecture the system is to run on, which in itself is material for another thesis). This places any code generation outside of the scope of this thesis. Indeed, even attempting refinement may fail due to time constraints. As stated by Shapiro and Weber [18]: “Operating systems are notoriously complex. Specifying the semantics of their operation is difficult, tedious, and error-prone.”

Chapter 5

Results

This chapter contains a detailed description (and justification) of the API specification that is the result of this thesis, as well as the methods and approach used to obtain it.

Basic knowledge of set theory and logic is assumed, however the B syntax and notation is explained on the way (see chapter 3 for an introduction).

It is intended as a guide to the complete B specification (provided in Appendix A). While the specification is listed in a top-down fashion to make finding things easier as well as to be able to see the broader picture, this section describes all machines and operations in a bottom-up fashion, describing how the complete model was obtained (this is also the order they were created in).

5.1 Design Methodology

To facilitate better understanding of the results, this section provides a description of the methods used to achieve it and the motivation behind them.

Basis

Firstly and most importantly, the model had to be based on a stable (that is, not moving) target. For this reason, the release version of pistachio available at the start of this thesis was chosen [5] as the basis for the model. For the duration of developing the model, further updates were ignored. This was especially helpful since the reference manual is always in the past of any fresh versions (especially those available from CVS).

Since this thesis was started with only a rudimentary knowledge of operating systems and absolutely no knowledge of L4, documentation was more important than being on the cutting edge.

This action resulted primarily in missing IPC redirectors, which are an extra functionality

added at the request of a certain company¹. It has generally been agreed to be a half-measure until a new security API is developed [16]. In my version of the documentation, they are mentioned, however in the implementation they are ignored. It has since come to my attention that they have indeed been implemented in the development version available via CVS.

Goals

The model was designed with the following goals in mind:

- A learning tool — animation of the model may allow for faster understanding of L4 internals by new inductees into the L4 verification project.
- Discovery — as L4 is continually adjusted and improved for efficiency, the initial intentions and ideas may have become overwhelmed by the deluge of optimisations. Trying to boil the system down to its essentials in the form of an abstract model may help to rediscover them.
- A starting point — while this undergraduate thesis presents a simplified model of the L4 API, the L4 Pilot project is proceeding with a “slice” of the system from specification all the way down to source code using Isabelle/HOL [17]. The initial slice consists of the virtual memory subsystem [21]. After the slice is complete, the API model should help with a decision on what the next slice will be.
- Documentation — during meetings with L4 personnel and conversations with former and current Advanced Operating Systems students, many of the questions I asked about L4 internals resulted in arguments of varying lengths. The conclusions reached are contained in the model and should help the next time such confusion arises.
- Experimentation — trying to formally model an existing system is a very difficult task that has already proven unsuccessful for some [10]. Just as there are a seemingly infinite amount of implementations possible for an abstract specification, so there is an infinite number of abstract thoughts that could have lead to the current implementation. Trying to find the minimal set of abstract components that describes the system is non-trivial. This thesis represents something that has not really been done before and so is bound to contain some inconsistency and misunderstanding, however someone trying to do this again will be able to build upon the knowledge.

Viewpoint

The API is the boundary between user space and kernel space. When building a model, the question is whose viewpoint do we model? From the perspective of a thread running in the system, operations would be the equivalent of system calls, they would return immediately and return actual values.

¹whose name cannot be disclosed at this time

From the kernel's perspective, however, the situation is quite different. Firstly, when a system call occurs, the kernel picks out the parameters from the thread's registers and memory, then passes them to an internal operation which implements the required functionality. Secondly, the operation does not have to return immediately. The kernel can freeze the thread, change its state, put it on a waiting queue and so forth. Finally, when the time comes to return a value, the system call does not return a value internally, but instead puts the return values back into the thread's registers and memory.

As far as creating a useful model, the second viewpoint is clearly superior. It allows for modelling of state transitions between threads and also does not force the modeller to specify everything in one go; variables in the system such as each thread's saved registers may be deferred to refinement without the need to change the top-level model later (which would be the case if all values had to be returned as operation return parameters).

This way of approaching the problem yields the non-standard situation of operations not returning any values. Whichever values would be returned are stored in internal state variables, modelled by non-determinism or deferred to refinement.

Completeness

Whatever is placed in a top-level specification in B is the *lower bound* on the functionality in the system. This means that leaving things out does not necessarily make the model wrong, as long as the behaviour is correct. An example of this is replacing difficult to specify functionality with non-determinism. For instance: stating that an operation either does nothing successfully or fails with an error is acceptable if that is what the system really does. During refinement, the exact details of what the operation does and how it determines failure can be defined¹.

This is the case with memory management in my model. Since memory management is part of IPC in L4, trying to add it into the specification at the top level makes things too complicated, especially since the top-level specification is restricted to parallel composition, but some form of iteration is required to implement mapping and granting (see 5.5.1, page 51).

Unfortunately, due to time constraints, I never reached the refinement stage, and so memory management remains absent from my model and its behaviour is simulated in my model by non-determinism.

Structure

When dealing with modular systems it is possible to structure a B development in such a way that functionality is distributed into various machines according to those models. A microkernel is unusual in this respect, since it must contain everything necessary to support an operating system *but no more*. This means that it is effectively a single module in which everything is intertwined. This does not create a very exciting structure for a development (as seen on figure 5.1).

¹This can only work if the state the operation is meant to work on has also been deferred to refinement.

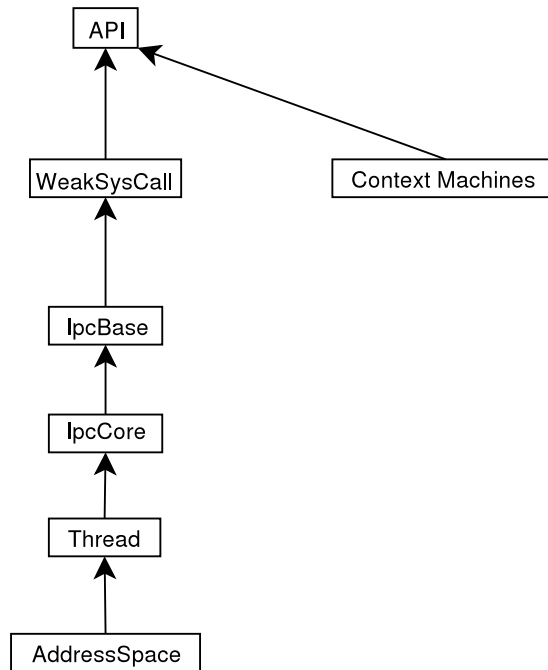


Figure 5.1: Inclusion diagram for the B development.

Since everything is reliant on everything else, the best separation I have been able to achieve is putting address spaces, threads and IPC in separate machines (though they still depend on each other as shown in figure 5.1).

All the types and constants are placed in context machines (suffixed “Ctx”). Apart from helpfully dividing the development into state machines and those providing abstract sets, the practice helps with refinement. They can also be used as a method of slowly building up the specification, adding only those context machines that are necessary at every stage.

The B Method does not impose any rigorous naming convention on its users, with the exception that all-capitals is reserved for names of sets and operations must begin with a capital letter. However, there is only one namespace, so names must be structured in such a way as to prevent collisions. This has led to including prefixes for all enumerations and a ‘d’ prefix for most definitions, as well as very long classifying names such as *thread_ipc_waiting_timeout*.

Possibly the biggest structural problem that B has presented is the inability to invoke two unrelated operations in parallel in a top-level specification. This means if you have two operations in the same machine: SetScheduler and SetError, clearly having nothing to do with each other, you cannot invoke them in parallel. This is very important because the top-level specification only allows parallel composition, and the B Toolkit only allows animation of the top-level specification.

This can be resolved by stripping the top-level specification as far as possible and heading straight for refinement where sequential composition can be used. However, one of my goals is successful animation of the specification, clearly mutually exclusive with this solution.

The other solution is to duplicate statements. In the above example, an extra operation

when both the scheduler and the error must be set which combines the statements from both in parallel will resolve the issue. This creates 'obvious' operations, but is the only way to reconcile the goals.

5.2 Types and Constants

The B Method uses first-order logic based on sets and set membership. Any type information is therefore also conveyed in terms of set membership. In order to define a system inside this methodology, we must first define abstract sets of *things* inside it. B supplies built-in sets such as NAT (all natural numbers, \mathbb{N}) and NAT1 ($\mathbb{N}_1 = \mathbb{N} - \{0\}$). Library machines provide other sets such as INTEGER and $\text{BOOL} = \{\text{FALSE}, \text{TRUE}\}$.

Sets may be defined *abstract* (a set with additional, optional, properties applied in the PROPERTIES clause) or *enumerated* (all members and the cardinality are defined immediately e.g. BOOL in the previous paragraph). Enumerated sets are used similarly to enumerations in programming languages such as C.

In addition to type information, all systems manage a finite set of resources. By defining abstract sets of *things* (such as thread numbers) and restricting their cardinality, we implicitly define an upper limit on the number of such things in the system. B later asks us to prove these limits are not exceeded.

5.2.1 Machine KernelInformation

In L4, a structure called a Kernel Information Page (KIP) contains all the constant values in the system (how many interrupts, first id of a user thread, etc.)

This machine serves a similar purpose, but we are concerned with limiting three main aspects of the kernel, the:

- number of threads in the system (`kMaxThreads`)
- number of address spaces in the system (`kMaxAddressSpaces`)
- number of threads in an address space (`kMaxThreadsPerSpace`)

These three constants are listed in the CONSTANTS part of the machine, and have the following properties:

PROPERTIES

$$\begin{aligned} kMaxThreads &\in \mathbb{N}_1 \wedge \\ 3 &\leq kMaxThreads \wedge \\ kMaxThreadsPerSpace &\in \mathbb{N}_1 \wedge \\ kMaxAddressSpaces &\in \mathbb{N}_1 \wedge \\ 3 &\leq kMaxAddressSpaces \end{aligned}$$

All three are defined to be non-zero. Each thread must have an address space; an address space can only be created by also creating a thread [12, section 2.4]. There are three address spaces initially in the system: the `sigma0` space, the root server space and the kernel space. The minimum number of address spaces is therefore 3, and the same goes for threads. The maximums must hence be at least 3 also.

5.2.2 Machine AddressSpaceCtx

In order to talk about address spaces within the model, further context is defined. This machine SEES all the constants and sets in KernelInformation (see 5.2.1). The SEES relationship allows only this.

The abstract set of all possible address spaces (think of them as pointers to address space structures) is defined and restricted to the maximum number of address spaces in the system:

SETS

ADDRESS_SPACE

PROPERTIES

$$\begin{aligned} \text{card} (\textit{ADDRESS_SPACE}) &= \textit{kMaxAddressSpaces} \wedge \\ \textit{kRootServerSpace} &\in \textit{ADDRESS_SPACE} \wedge \\ \textit{kSigma0Space} &\in \textit{ADDRESS_SPACE} \wedge \\ \textit{kKernelSpace} &\in \textit{ADDRESS_SPACE} \wedge \\ \textit{kRootServerSpace} &\neq \textit{kSigma0Space} \wedge \\ \textit{kSigma0Space} &\neq \textit{kKernelSpace} \wedge \\ \textit{kRootServerSpace} &\neq \textit{kKernelSpace} \end{aligned}$$

In the above, three new *distinct* constants are defined. Their function is to reserve three arbitrary members of ADDRESS_SPACE for the three core address spaces:

- *kSigma0Space* to hold *sigma0*
- *kRootServerSpace* to hold the root server
- *kKernelSpace* to hold the interrupt threads

Note that *sigma1*, which is also given its own address space in the kernel, is not listed here. That is because it is a persistence extension to L4 (and thus is outside the scope of this model), is unimplemented in the source code version my model is based on [5], and discussions with L4 personnel indicate that it is not a core feature of the kernel and should be disregarded at this stage.

These address spaces are *privileged*, and a DEFINITIONS clause is used to define a useful macro for testing whether an address space is one of these:

DEFINITIONS

$$\textit{dIsPrivilegedSpace} (s) \hat{=} s \in \{ \textit{kSigma0Space} , \textit{kRootServerSpace} , \textit{kKernelSpace} \}$$

A definition in B is a direct rewrite in the pre-processing stage. Note that the single letter ‘s’ is a *joker* and represents *any* B expression. All single-letter tokens are jokers in B.

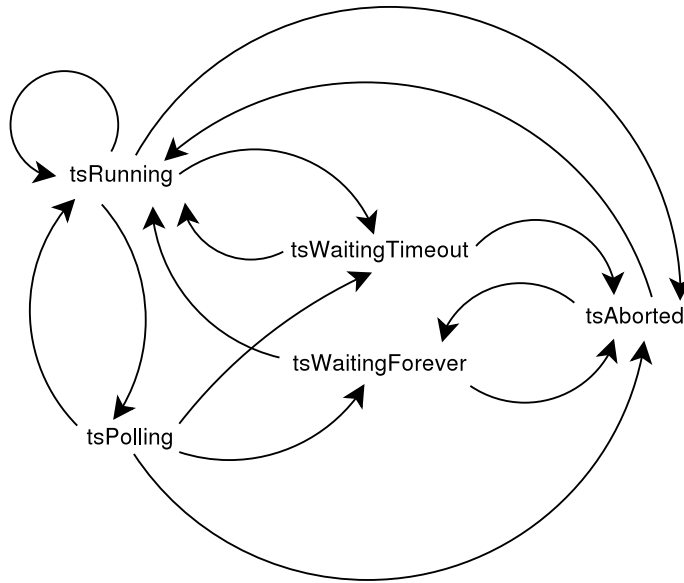


Figure 5.2: A very simplified diagram of possible state transitions.

If you are looking for a definition or DEFINITIONS clause in the marked-up version of the specification and cannot find it, please note that the mark-up system in the B Toolkit places them at the *end* of machines, not at the beginning as one might expect.

5.2.3 Machine ThreadStateCtx

This machine defines an enumerated set (THREAD_STATE) of simplified states that threads can experience in L4:

- tsAborted — the thread exists, but has not been initialised
- tsRunning — the thread has been initialised and if scheduled, can run
- tsPolling — thread is waiting on an IPC send to another thread
- tsWaitingTimeout — thread is waiting for incoming IPCs from one or more threads, with a finite time-out
- tsWaitingForever — as above, but the time-out is infinite

Figure 5.2 presents the possible transitions between these states. A more complete diagram can be found on page 63.

These states differ from the ones in the L4 implementation in following ways:

- Multiprocessing-related states (e.g. xcpu_waiting_deltcb) are missing (my model is too abstract to model multiple-CPU interaction);

- Locked states are missing (e.g. `locked_running`) as they are also primarily multiple-CPU related;
- The halted state is missing. After discussing this with other individuals involved with L4, this state is better modelled by a flag. As defined in [12, section 2.3], halting a thread prevents it from executing in user mode, but ongoing IPC is *not affected*. One understanding of this is that it simply prevents the thread from being scheduled. Furthermore, the `ExchangeRegisters` system call needs to resume halted threads, creating the need for another (saved) thread state. This can still be refined in if necessary, but makes for a simpler model;
- The aborted state has a slightly different meaning that in the L4 implementation [5, `thread.cc`]. In L4, all kernel thread control blocks are pre allocated and their initial state is aborted. When a thread gets created inactive, the state *remains* aborted. The actual existence of a thread is defined as the thread having been assigned to an address space. Deleting a thread involves deleting this assignment. In my model, the non-existence of a thread is marked by its absence from the set of existing threads, so the threads do not have any actual state. Once the thread is created inactive, the two viewpoints merge.

5.2.4 Machine ThreadCtx

ThreadCtx defines the context of threads excluding thread identifiers (see 5.2.5 ThreadIdCtx).

Two more definitions are added:

DEFINITIONS

$$\begin{aligned} \text{canSend} (t) &\hat{=} \text{thread_state} (t) \in \{ \text{tsRunning} , \text{tsPolling} \} ; \\ \text{canReceive} (t) &\hat{=} \text{thread_state} (t) \in \{ \text{tsWaitingTimeout} , \text{tsWaitingForever} \} \end{aligned}$$

These demonstrate definitions in B are purely syntactic, since `thread_state` is only defined in Thread (5.4.1) which this machine does not see.

To send an IPC, a thread must either be running (it invokes the IPC) or polling (the kernel invokes the IPC on behalf of the thread). To receive one, it must be waiting.

The machine defines a set TCB restricted to `kMaxThreads` (see 5.2.1), which represents all possible threads creatable the system. I have chosen this name due to its similarity to the pre allocated Kernel Thread Control Blocks in the system. The two constants `kSigma0` and `kRootServer` reserve two distinct members of TCB for `sigma0` and the root server respectively.

Additionally, the constant `kIntThreads` reserve a subset of TCB for interrupt threads as follows:

$$\begin{aligned} \text{kIntThreads} &\subset \text{TCB} \wedge \\ \text{kIntThreads} &\neq \{ \} \wedge \\ \text{card} (\text{kIntThreads}) &\leq \text{kMaxThreadsPerSpace} \wedge \\ \text{kSigma0} &\notin \text{kIntThreads} \wedge \\ \text{kRootServer} &\notin \text{kIntThreads} \end{aligned}$$

kIntThreads is a proper subset of TCB, of which kRootServer and kSigma0 are not members. Since interrupt threads go in the kernel address space, there must not be more than kMaxThreadsPerSpace of them. There must be at least one interrupt thread in the system.

The set EXREGS_FLAGS defines the various options that can be passed into the ExchangeRegisters system call [12, section 2.3]: ex_h represents h , ex_R represents R and so on for all the flags: *hpufisSRH*. These will be covered in detail as the design of ExchangeRegisters is incrementally revealed.

5.2.5 Machine ThreadIdCtx

Now that the thread context is defined, thread identifiers, the user's view of threads need to be addressed.

The set THREAD_GNO represents all possible global thread identifiers. The constants *kAnyGNo* and *kNilGNo* represent *anythread* and *nilthread* respectively. There must be enough thread numbers for all threads plus two for the aforementioned constants, making the set cardinality $kMaxThreads + 2$.

Local thread identifiers are a performance enhancement. In L4, the local thread id of a thread is literally an offset into the array of user TCBs in the current address space. This makes working with threads in the same address space much faster, and allows a further optimisation to IPC called Local IPC (LIPC). Since my goal was to create as simple and high-level model of the system as I could, this and other optimisations were removed.

Thread versions, on the other hand, are another performance enhancement, this time to do with thread renaming. An example of how this is useful:

1. Thread A is created and activated, with thread B as its scheduler;
2. B is deleted, but A 's scheduler is not updated, since finding all threads B was scheduling would be slow;
3. Thread C is created, and gets the same kernel TCB as B had (which in L4 also means the same thread no. in the global thread ID). This process is very efficient and does not actually allocate anything new. If the version field is not incremented, then C can function as A 's scheduler, which is undesirable. Thread versions allow this fast renaming to work, but *the kernel does not do this automatically*.

The above shows why at the abstract specification level, versions and thread numbers can easily be made into one set, which I have chosen to call GLOBAL_TNO. Versions can be added during further refinement.

5.2.6 Machine TimeoutCtx

While in the L4 implementation time-outs have actual values, in an abstract specification the exactness of these values do not have much of an effect. The enumerated set TIME-

OUT therefore contains only three values: *eZeroTimeout*, *eFiniteTimeout*, *eInfiniteTimeout*. Refinement to exact values is possible.

The effect of this is essentially IPC:

- *eZeroTimeout* requests an action be taken (or it will fail) immediately
- *eFiniteTimeout* means that the thread will wait or poll for some time until timed out by the kernel, or cancelled by another thread
- *eInfiniteTimeout* indicates that unless the operation is cancelled, the wait will go on indefinitely

Three definitions are added: *isFinite*, *isInfinite*, *isNoTimeout* testing for *eFiniteTimeout*, *eInfiniteTimeout* and *eZeroTimeout*. This is to prevent relating to those members of TIMEOUT directly, in case the values will be changed from abstract to exact at some point in the future; such a change would require only modification of the definitions.

5.2.7 Machine FpageCtx

This machine is only used at the top level to pass in Fpage parameters. Since my model does not contain memory management, the implications of this machine are not far-reaching.

Nonetheless, the machine defines read, write and execute permissions in an enumerated set PERMS, and then uses definitions to describe Fpages and their aspects:

$$\begin{aligned}
dFpage (b , s , p) &\hat{=} b , s , p ; \\
dFpagePerms &\hat{=} \text{prj2} (\mathbb{N} \times \mathbb{N} , \mathbb{P} (PERMS)) ; \\
dFpageBase (f) &\hat{=} \text{prj1} (\mathbb{N} , \mathbb{N}) (\text{prj1} (\mathbb{N} \times \mathbb{N} , \mathbb{P} (PERMS)) (f)) ; \\
dFpageSize (f) &\hat{=} \text{prj2} (\mathbb{N} , \mathbb{N}) (\text{prj1} (\mathbb{N} \times \mathbb{N} , \mathbb{P} (PERMS)) (f)) ; \\
dIsFpage (f) &\hat{=} \text{bool} (f \in \mathbb{N} \times \mathbb{N} \times \mathbb{P} (PERMS)) ; \\
FPAGE &\hat{=} \mathbb{N} \times \mathbb{N} \times \mathbb{P} (PERMS)
\end{aligned}$$

FPAGE is just a Cartesian product of two natural numbers and a set of permissions (a subset of PERMS). *dIsFpage* checks for this, and *dFpage* constructs an Fpage out of a base, size and set of permissions.

In order to retrieve individual aspects of Fpages, projections are used. A projection is a function that, given two parameter types, will take a member of the Cartesian product of those types (a pair), and return the left or right member of the pair.

Thus *dFpagePerms* is a projection (*prj2*) returning the second element of the pair whose first element is of type $\mathbb{N} \times \mathbb{N}$ and second is a subset of PERMS.

To retrieve the base, first the (base, size) pair is retrieved using *prj1*, then *prj1* is used again to retrieve the base. If the second step is *prj2*, the size is retrieved.

5.2.8 Machine ErrorCtx

The enumerated set `ERROR` lists all the possible errors returned to the thread in the API model. These will be explained in detail later.

Also, the definition *dIpcFailures* lists the failures during IPC that are beyond the deterministic control of the abstract model. If IPC fails non-deterministically, one of these is the error.

5.3 Address Spaces

Once the context is set up, the first important aspect of L4, on which all other aspects are based, is address spaces. Since my model does not go into the details of memory management, a simple model of which spaces are used by the system and which of those have been initialised suffices.

5.3.1 Machine AddressSpace

The machine SEES `KernelInformation` (see 5.2.1) and `AddressSpaceCtx` (see 5.2.2) which imports their abstract sets and constants. It needs to `SEE` both since SEES is *not transitive*.

Next, two variables are introduced:

- *spaces* represents the address spaces that have been created
- *initialised_spaces* represents those address spaces that are created and initialised

Invariant

Their relationship is defined in the `INVARIANT` clause (see 3):

INVARIANT

$$\begin{aligned} \textit{spaces} &\subseteq \textit{ADDRESS_SPACE} \wedge \\ \textit{initialised_spaces} &\subseteq \textit{spaces} \end{aligned}$$

There cannot be more address spaces created in the system than the system can hold, nor can more be initialised than have been created. Being initialised automatically implies being created.

Think of it as a total pool of pointers to address space structures being exhausted as they are created, and *initialised_spaces* being an attribute of the structure.

Initialisation

Every variable in `B` must be initialised in a manner that establishes the invariant. Since this is an abstract model of the functioning of L4, one would expect the variables to be initialised

to what the root task sets them to be before giving over control to L4.

In *AddressSpace* (see 5.3.1) three address spaces were reserved: `kSigma0Space`, `kRootServerSpace`, `kKernelSpace`. These are the spaces created and initialised by the root task on start up. We will do the same:

INITIALISATION $spaces := \{ kSigma0Space , kRootServerSpace , kKernelSpace \} \parallel$
 $initialised_spaces := \{ kSigma0Space , kRootServerSpace , kKernelSpace \}$

Note the parallel composition (\parallel) operator, which is the only way to compose operations at the top specification (machine) level in B. This contrasts with the *sequential* composition of ordinary programming languages.

Operations

Next we define the operations that modify the state (variables) of this machine in the OPERATIONS (see 3) clause.

Since the operations are designed in such a way that satisfying their preconditions guarantees success, they do not return any values (this is not always the case; more on this later).

The three operations we will have to perform are: creating an address space, initialising it and deleting it. Given my current understanding of L4 and the reference manual [12, section 4.3: SPACECONTROL], once an address space is initialised it cannot be *uninitialised*. SpaceControl forces the `UtcbaArea` and `KernelInterfacePageArea` to be valid for the call to succeed. Having these two areas set to valid values constitutes initialisation. To invalidate them would require passing invalid parameters to SpaceControl, which will fail. Ergo, no uninitialisation.

Since these are the first operations described in this document, let us look at them in detail.

CreateAddressSpace ($space$) $\hat{=}$
PRE $space \in ADDRESS_SPACE - spaces$ **THEN**
 $spaces := spaces \cup \{ space \}$
END

To guarantee the success of this operation, the address space identifier passed in must be one of those not yet created. This becomes the precondition (see 3). In L4's case, address space identifiers are just pointers to address space structures and in any case, not visible at the user level.

Once the precondition is satisfied, the new identifier is added to the set of created address spaces.

Note that for operations, $\hat{=}$ is not purely syntactic as is the case for definitions.

InitialiseAddressSpace ($space$) $\hat{=}$

```

PRE   space ∈ spaces   THEN
        initialised_spaces := initialised_spaces ∪ { space }
END

```

If the space identifier passed in is one of those already created, the operation succeeds and the identifier is added to the set of initialised address spaces.

```

DeleteAddressSpace ( space ) ≐
PRE   space ∈ spaces ∧ ¬ ( dIsPrivilegedSpace ( space ) )   THEN
        spaces := spaces - { space } ||
        initialised_spaces := initialised_spaces - { space }
END

```

Here we see a more interesting use of a precondition. To satisfy the invariant, it suffices that any member of ADDRESS_SPACE be passed in. For the operation to make sense, however, extra meaning is added. We would like B to (via proof obligations) force us to prove any invocation of the operation is on an existing address space (otherwise the invocation is superfluous).

Additionally, L4 does not allow deletion of privileged threads on the basis of them belonging to privileged address spaces [5, SYS_THREAD_CONTROL in thread.cc]. Clearly, requiring that *space* not be a privileged space as precondition to DeleteAddressSpace forces the same behaviour as in the implementation, since in defining DeleteThread we will be asked to prove we are not deleting a privileged address space.

5.4 Threads

Threads are an extremely important component in L4; they are the ones communicating, using system resources, and invoking the kernel's system calls.

Nearly all relationships in the system involve threads, making distribution of the thread model throughout more than one machine a difficult task. Eventually, the thread functionality was divided into three machines:

- *Thread* contains all aspects of threads not directly related to IPC (such as state, pagers, schedulers, etc.)
- *IpcCore* contains the place holder for an operation copying one thread's virtual registers onto another
- *IpcBase* contains purely IPC-related aspects of threads, such as which thread is waiting on another.

Since everything in the kernel is extremely intertwined, and B imposes a layered approach, some of this distribution ended up more complex than would be expected from a greatly simplified specification.

5.4.1 Machine Thread

Since the reader is now familiar with the main sections in a B machine, each variable involved in a machine will from this point on be described one-by-one, followed by a description of its link to previously introduced variables as described in the invariant. New sections will still be explained as they arise.

The Thread machine models all non-IPC related functionality of threads (including that which enables IPC functionality of course).

A very interesting abstraction that this machine provides is the lack of a single executing thread or any mention of individual processors. There are two ways of looking at this abstraction, again relating to kernel vs. thread points of view (see 5.1 on page 24):

1. All threads with a *running* state are executing simultaneously on a magical machine with one processor per thread;
2. From each thread's perspective, the notion of being suspended while another thread is running is not immediately noticeable, so each thread can believe it is executing all the time (indeed, making threads believe this successfully is one of the roles of an operating system).

Section: INCLUDES

The first thing to note is that the machine INCLUDES AddressSpace. That means everything in the AddressSpace machine except for what it SEES is now visible at any stage in the Thread machine. Since all machines must eventually be included into one, that is also the reason for the rather verbose naming conventions.

Write access must be done exclusively using AddressSpace's operations. The initialisations for the two machines are composed in parallel (in the case of the B Toolkit; other tools may compose them sequentially, which is more agreeable with a programmer's viewpoint).

Section: PROMOTES

This brings us to the "PROMOTES InitialiseAddressSpace" clause. This means that when Thread is included into another machine, the operation InitialiseAddressSpace will be available in addition to all the ones defined in thread. INCLUDES is not transitive for operations. B will ask us to prove that the invariant of the Thread machine cannot be violated by any of the promoted operations.

Variable: threads

$$threads \subseteq TCB$$

Similarly to how *spaces* represents created address spaces (see 5.3.1), *threads* represents the set of threads created in the system. It is bound by the size of TCB, the set of all possible threads (see 5.2.4).

Variable: thread_gno

$$\begin{aligned} & thread_gno \in threads \mapsto GLOBAL_TNO \wedge \\ & kAnyGNo \notin \text{ran} (thread_gno) \wedge \\ & kNilGNo \notin \text{ran} (thread_gno) \end{aligned}$$

The thread number is an attribute of the thread. In abstract terms, that means it is a function from the set of threads in the system (*threads*) to the set of all possible thread numbers (*GLOBAL_TNO*). The function is injective (one-to-one), since no two threads' numbers may be identical. What is more, the function is total. All threads in the system must have an identifier.

Note: in B, we say that a function is a member of the set of all functions meeting given constraints, hence the membership operator (\in).

The two thread numbers reserved for *nilthread* and *anythread*, *kNilGNo* and *kAnyGNo* (see 5.2.5) may not be in the range of this function.

Variable: halted_threads

$$halted_threads \subseteq threads$$

The set of all threads which are halted. With the exception of interrupt threads, this means a thread will not enter user mode. It may be argued that interrupt threads cannot enter user mode either due to being an abstraction of an interrupt routine, but being halted nonetheless has a different meaning for them (it means the interrupt in question is enabled; more on this when *thread_state* is covered).

Variable: active_threads

$$\begin{aligned} & active_threads \subseteq threads \wedge \\ & kSigma0 \in active_threads \wedge \\ & kRootServer \in active_threads \wedge \\ & kIntThreads \subseteq active_threads \end{aligned}$$

As the name suggests, *active_threads* represents the set of all threads which have gone through the activation procedure. For the privileged threads (*sigma0*, the root server and interrupt threads), this happens on kernel start up. That is why *sigma0*, the root server, and all interrupt threads are always in this set.

Variable: thread_space

$$\begin{aligned} & thread_space \in threads \mapsto spaces \wedge \\ & thread_space (kSigma0) = kSigma0Space \wedge \\ & thread_space (kRootServer) = kRootServerSpace \wedge \end{aligned}$$

$$\begin{aligned}
\text{thread_space } [\text{active_threads}] &\subseteq \text{initialised_spaces} \wedge \\
\text{thread_space } [\text{kIntThreads}] &= \{ \text{kKernelSpace} \} \wedge \\
\text{thread_space}^{-1} [\{ \text{kKernelSpace} \}] &= \text{kIntThreads}
\end{aligned}$$

This function maps all threads to address spaces in the system. Since no address space may exist without a thread in it (a space may only be created by `THREADCONTROL`, which also creates a thread inside it [12, section 2.4]), the function is surjective (denoted \rightarrow).

The address spaces of `sigma0` and the root server are `kSigma0Space` and `kRootServerSpace` respectively.

The last three statements use the notation for *relational image*, which is defined as:

$$r [S] = \{ y \mid \exists x \cdot x \in S \wedge x \mapsto y \in r \}$$

In the case of a function, this means all y such that $f(x) = y$.

Note that the $x \mapsto y$ means the pair (x,y) , or “ x maps to y ”.

The first statement declares that for a thread to be active it must reside in an initialised address space.

The second means *all the interrupt threads reside in kKernelSpace*, and the third that *all threads in kKernelSpace are interrupt threads* (where $^{-1}$ denotes the relational inverse).

In earlier versions of the specification, I had similar rules for the `sigma0` and root server address spaces. After much discussion, this turned out to be incorrect. Strictly speaking, L4 will allow a thread to create another in the same address space using `EXCHANGEREGISTERS` [12, section 2.3]. Whether this is a good idea or not has been left to the implementers of privileged threads to decide.

However, interrupt threads are only an abstraction of the underlying hardware, and cannot actually run or have an implementation inserted from user mode. It is only because of this that the above constraints hold.

Variable: `thread_scheduler`

$$\begin{aligned}
\text{thread_scheduler} &\in \text{threads} - \text{kIntThreads} \rightarrow \text{TCB} \wedge \\
\text{thread_scheduler} (\text{kSigma0}) &= \text{kRootServer} \wedge \\
\text{thread_scheduler} (\text{kRootServer}) &= \text{kRootServer}
\end{aligned}$$

In L4, the scheduler is defined as part of the kernel thread control block. This means that all threads will have some field to set. However, for interrupt threads, schedulers are meaningless because they never actually run. So `thread_scheduler` maps all threads in the system *except* interrupt threads to some TCB that might or might not be in the system.

There are two issues with this:

- Why would you permit a scheduler which does not exist?

- Why not use a partial function for `thread_scheduler`?

The answer to the first question is invariably: efficiency and flexibility. Why put the extra check in? Schedulers are not that important in L4. Their name is quite deceptive, since they are not involved in scheduling anything. A thread’s scheduler is allowed to set that thread’s scheduling parameters, such as priority. If a thread has an invalid scheduler, these parameters simply remain the same and harm comes to the kernel. If the user depends on schedulers being valid and mistakenly sets one as invalid or deletes a thread’s scheduler, the technical term used is “shooting oneself in the foot”. In other words, if the user uses schedulers, he or she should worry about managing them.

Partial functions (as opposed to the total functions introduced so far) do not have to contain a mapping for every member in their declared domain. That is, if a function is declared $X \mapsto Y$, then $\text{dom}(X) \subseteq Y$. I used a total function for two reasons: firstly, it models the situation that occurs in L4 better (i.e. threads running with invalid schedulers), and secondly because it is then simpler to check a thread’s scheduler (no need to check if the thread is in the domain first).

The root server is initialised as the scheduler for `sigma0` and itself. I believe this situation should be maintained at all times, since otherwise the privileged threads could lose control of the system. It is possible this constraint is unnecessary, however no information has been presented to support this.

Variable: `thread_pager`

$$\begin{aligned} & \text{thread_pager} \in \text{threads} \mapsto \text{TCB} \wedge \\ & k\text{Sigma0} \notin \text{dom} (\text{thread_pager}) \wedge \\ & \forall kk . (kk \in k\text{IntThreads} \wedge kk \notin \text{halted_threads} \Rightarrow \text{thread_pager} (kk) = kk) \wedge \\ & \forall kk . (kk \in k\text{IntThreads} \wedge kk \in \text{halted_threads} \Rightarrow \text{thread_pager} (kk) \neq kk) \end{aligned}$$

In L4, the process of page faults (see 2.3.4) is resolved via IPC. This means a faulting thread needs a target to ‘send’ to (however it is the kernel which really performs the action on behalf of the thread). This target is known as a *pager*.

The function is partial for two reasons:

- `sigma0`, being the initial system pager holding all the memory, does not have another pager to fall back on
- until the thread is activated, the pager field in its TCB is meaningless (in fact, setting a valid pager during a `THREADCONTROL` operation constitutes activation)

Since `EXCHANGEREGISTERS` [12, section 2.3] may set the pager to a thread which does not exist, the range of the function cannot be enforced. Furthermore, a thread’s pager may be deleted, and checking for this is an time-expensive operation (similar to schedulers, see 5.4.1), making it possible for a thread to have an invalid pager at some point in time.

Even though interrupts are abstracted as threads in the system, the abstraction is not complete. This is because interrupt threads:

- never run, but also never reach a *running* state (page 41)
- when activated, they are *halted* and their pager (referred to as the *handler*) is set to the thread that will handle the interrupt (interrupt handling is also implemented though IPC)
- when inactive, their pager is set to themselves, and they are *not halted*

One of my initial assumptions upon examining the L4 source [5] and documentation was that setting a (non-interrupt) thread to be its own pager should be prevented by the kernel, since it is easily detectable, and its results are uncertain. After much discussion with L4 personnel, a position was reached, which is worth noting:

- There is no good reason for setting a thread as its own pager, but there is also no reason to prevent someone from doing it;
- In the current implementation, an IPC will be sent to the thread's pager without checking for who that pager is. Since the time out for that IPC is infinite, a send to itself will cause the thread to be suspended indefinitely. The usefulness of the behaviour is unknown, but the behaviour itself is well-defined.

Variable: `thread_state`

$$\begin{aligned} & thread_state \in threads \rightarrow THREAD_STATE \wedge \\ & active_threads \cap thread_state^{-1} [\{ tsAborted \}] \subseteq kIntThreads \wedge \\ & tsRunning \notin thread_state [kIntThreads] \wedge \end{aligned}$$

All threads in the system must be in one of the known states.

The *aborted* state and a thread being active are mutually exclusive, with the exception of interrupt threads, which do not achieve a *running* state under any circumstances. Since they participate in IPC, they can assume waiting and polling states, but once IPC is resolved they return to *aborted*. A probable reason for this is efficiency: since the scheduler only looks for *running* threads to execute, it will automatically overlook interrupt threads, at the price of making the interrupts-as-threads abstraction less complete.

See figure 5.2 on page 30 for a diagram of possible state transitions.

Variable: `threads_in_space`

$$\begin{aligned} & threads_in_space \in spaces \rightarrow 0 .. kMaxThreadsPerSpace \wedge \\ & \forall ss . (ss \in spaces \Rightarrow card (thread_space \triangleright \{ ss \}) = threads_in_space (ss)) \end{aligned}$$

An abstract variable maintaining a count of the number of threads in each address space. The first statement restricts the range of this function to be less than the maximum allowed in the system (`kMaxThreadsPerSpace`).

The second statement is to make sure that `threads_in_space` maintains the correct value at all times. Essentially, it states that `threads_in_space(ss)` must be equal to the cardinality of the set of all threads in that address space.

For a relation r and set S , the domain restriction operator (\triangleright) is defined as follows:

$$r \triangleright S = \{x \mapsto y \mid x \mapsto y \in r \wedge y \in S\}$$

Note that strictly speaking the number of threads in ss is

$$\text{card}(\text{dom}(\text{thread_space} \triangleright \{ss\}))$$

but for functions, the cardinality of their domain is equal to the cardinality of all the mappings.

Assertions

In B, proof can be made easier by re-stating some aspects of the invariant in different ways. These statements may be put into an ASSERTIONS clause, and once proven may be assumed to be true.

For threads, given that:

$$\text{thread_scheduler} \in \text{threads} - \text{kIntThreads} \rightarrow \text{TCB} \wedge$$

we can conclude that:

$$\text{thread_scheduler} [\text{kIntThreads}] = \{\}$$

In other words, if the domain of `thread_scheduler` does not include `kIntThreads`, then no member of `kIntThreads` will ever yield a result in a relational image.

Initialisation

Firstly, introduce the threads that will be known to the kernel once start-up is completed. These are `sigma0`, the root server, and interrupt threads.

$$\text{threads} := \{ \text{kSigma0}, \text{kRootServer} \} \cup \text{kIntThreads}$$

The threads will all be activated:

$$\text{active_threads} := \{ \text{kSigma0}, \text{kRootServer} \} \cup \text{kIntThreads}$$

Note: L4 actually allows a process of activating interrupt threads. What this means is that while all kernel threads exist in the system to start with (i.e. they have kernel TCBs), they do not have user mode TCBs (UTCBS). Possession of a UTCB is what defines a thread as *active*. I can only assume that this was done to conserve resources by lazy allocation, which means that from a top-level specification's perspective, they are all created active, since my model does not include UTCBs.

They will be in the address spaces `kSigma0Space`, `kRootServerSpace` and `kKernelSpace` respectively:

$$\begin{aligned} \text{thread_space} &:= \{ kSigma0 \mapsto kSigma0Space , \\ &kRootServer \mapsto kRootServerSpace \} \cup kIntThreads \times \{ kKernelSpace \} \end{aligned}$$

Note: the Cartesian product of $kIntThreads$ and the singleton set $\{kKernelSpace\}$ is a function mapping all the interrupt threads to that space.

Next, these threads will have thread numbers. We do not know what they are, but we do know they must satisfy the invariant, i.e. be unique and not include *anythread* and *nilthread*:

$$\begin{aligned} \text{thread_gno} &:\in \{ kSigma0 , kRootServer \} \cup kIntThreads \mapsto \\ &GLOBAL_TNO - \{ kNilGNo , kAnyGNo \} \end{aligned}$$

The $:\in$ operator means *assign value to any of*, meaning that if we take the set of all possible functions whose domains are $\{ kSigma0 , kRootServer \} \cup kIntThreads$ and whose ranges are $GLOBAL_TNO$ (but do not include $kAnyGNo$ and $kNilGNo$), any of those functions can be assigned to the variable on the left hand side of the operator. This is known as non-deterministic assignment.

If we examine the invariant (page 38) we see that any such function is satisfactory.

An operating systems programmer might object to such a vague assignment, and indeed, it is far from the implementation. During refinement, it would be replaced by an arbitrary function of the implementers choosing. For an abstract model, it offers simplicity.

In my model, interrupt threads start out disabled:

$$\text{halted_threads} := \{ \}$$

The decision is arbitrary. It is possible to include setting, say, the root server as a handler for some interrupts during kernel start-up, but if the root server wants to handle an interrupt it can set itself as its handler just as easily. In the interest of flexibility, I have chosen the latter.

Since interrupt threads start out disabled, they are their own pagers. Additionally, sigma0 is the root server's pager:

$$\text{thread_pager} := \{ kRootServer \mapsto kSigma0 \} \cup \text{id} (kIntThreads)$$

Note: id is the *identity relation*, defined as:

$$\text{id}(S) = \{ x \mapsto x \mid x \in S \}$$

The root server starts up as the scheduler for sigma0 and itself [5, thread.cc]:

$$\text{thread_scheduler} := \{ kSigma0 \mapsto kRootServer , kRootServer \mapsto kRootServer \}$$

The root server and sigma0 start with a *running* state, while interrupt threads start out as *aborted*:

$$\begin{aligned} \text{thread_state} &:= \{ kSigma0 \mapsto tsRunning , kRootServer \mapsto tsRunning \} \\ &\cup kIntThreads \times \{ tsAborted \} \end{aligned}$$

Finally, we set the thread counters in the respective address spaces:

$$\begin{aligned} \text{threads_in_space} &:= \{ kSigma0Space \mapsto 1 , kRootServerSpace \mapsto 1 , \\ &kKernelSpace \mapsto \text{card} (kIntThreads) \} \end{aligned}$$

5.4.2 Machine Thread: Operations

CreateThread

Creates an inactive thread, given a free TCB, thread number, space and scheduler:

$$\mathbf{CreateThread} (tcb , global_tno , space , scheduler)$$

Let us look at what is needed to guarantee the operation succeeds, which becomes the precondition (see 3).

Firstly, the supplied *tcb* must be a member of TCB (see 5.2.4), but not already assigned to a thread in the system (a member of threads):

$$tcb \in TCB - threads$$

Next, the thread number *global_tno* must not be one used for a thread in the system, nor one of the reserved identifiers (*nilthread* or *anythread*):

$$\begin{aligned} global_tno &\in GLOBAL_TNO \wedge \\ global_tno &\notin \mathbf{ran} (thread_gno) \wedge \\ global_tno &\neq kNilGNo \wedge \\ global_tno &\neq kAnyGNo \end{aligned}$$

Since an inactive thread is being created, the only restriction placed on *scheduler* is that it be a member of TCB.

The address space the thread is to be created in need not exist, but it cannot be the kernel space (which is reserved for interrupts):

$$\begin{aligned} space &\in ADDRESS_SPACE \wedge \\ space &\neq kKernelSpace \end{aligned}$$

The reference manual [12] states that if the no address space is passed in to `THREADCONTROL` for a creation operation, a new address space is created. The `CreateThread` operation takes this into account, by creating a new address space if the one passed in is not known to the system. If the address space is known to the system, then the total number of threads in it must be less than the allowable maximum, or adding a new thread will exceed it:

$$(space \in spaces \Rightarrow threads_in_space (space) < kMaxThreadsPerSpace)$$

Given these properties, `ThreadControl` is guaranteed to succeed in this model. Let us look at what this operation does. Note that while I might imply a sequential process by going over the statements one by one, the mode of composition in a top-level specification is *parallel* (see 3).

If the address space supplied is one of those in the system, increment the thread counter (page 41). If it is not, create a new, uninitialised address space (page 35) and set the thread counter to one:

IF $space \notin spaces$ **THEN**

```

    CreateAddressSpace ( space ) ||
    threads_in_space ( space ) := 1
ELSE
    threads_in_space ( space ) := threads_in_space ( space ) + 1
END

```

Simultaneously, add *tcb* to the set of threads in the system (page 37), set its thread number, address space, scheduler, and state (*aborted*, since the thread will be inactive):

```

    threads := threads ∪ { tcb } ||
    thread_gno ( tcb ) := global_tno ||
    thread_space ( tcb ) := space ||
    thread_scheduler ( tcb ) := scheduler

```

Note: `||` is the parallel composition operator

In the description of following operations, trivial preconditions to do with type safety (such as $tcb \in \text{TCB}$) will be omitted unless not obvious.

ActivateThread

In order for a thread to be able to do anything in the system, it must first be activated. This can be done as part of creation (page 46), or as an ActivateThread operation on an inactive thread:

```

ActivateThread ( tcb , space , pager , scheduler )

```

tcb must be an existing but inactive thread:

$$tcb \in \text{threads} \wedge tcb \notin \text{active_threads}$$

The pager must be an existing thread and the scheduler must exist and be running when the thread starts executing [12, section 2.4]:

$$\begin{aligned}
 & pager \in \text{threads} \wedge \\
 & scheduler \in \text{active_threads}
 \end{aligned}$$

Since thread activation is a part of THREADCONTROL, the possibility of the thread being migrated while being activated exists. As for CreateThread above, we must make sure that the thread fits into the new space:

$$\begin{aligned}
 & space \in \text{initialised_spaces} \wedge \\
 & (space \neq \text{thread_space} (tcb) \Rightarrow \\
 & \quad \text{threads_in_space} (space) < kMaxThreadsPerSpace)
 \end{aligned}$$

The operation itself updates the pager and the scheduler, adds *tcb* to active threads, sets its state to *tsWaitingForever*, and migrates the thread if necessary.

In L4, a thread will begin waiting for an IPC from its pager straight after activation. This is why its state begins as waiting forever. The IPC component will be initialised in ActivateThread2 (see 5.5.3).

The actual migration is performed as follows:

```

IF   space  $\neq$  thread_space ( tcb ) THEN
    thread_space ( tcb ) := space ||
    threads_in_space := threads_in_space  $\Leftarrow$  { space  $\mapsto$  threads_in_space ( space ) + 1 ,
        thread_space ( tcb )  $\mapsto$  threads_in_space ( thread_space ( tcb ) ) - 1 }
END

```

If the designated space is not the same as the space the thread is currently in, it is assigned to the new space.

The thread counters for the two address spaces (current and target) are updated using *right overriding* (denoted \Leftarrow). The exact definition is:

$$r_1 \Leftarrow r_2 = r_2 \cup (\text{dom}(r_1) \Leftarrow r_2)$$

The definition uses another of B's operators, *domain subtraction* (\Leftarrow), defined as:

$$S \Leftarrow r = \{ x \mapsto y \mid x \mapsto y \in r \wedge x \notin S \}$$

In other words:

$$\begin{aligned} \text{threads_in_space} &:= \text{threads_in_space} \Leftarrow \{ \text{space} \mapsto \text{threads_in_space} (\text{space}) + 1 , \\ &\quad \text{thread_space} (\text{ tcb }) \mapsto \text{threads_in_space} (\text{thread_space} (\text{ tcb })) - 1 \} \end{aligned}$$

has the same effect as:

$$\begin{aligned} \text{threads_in_space} (\text{space}) &:= \text{threads_in_space} (\text{space}) + 1 ; \\ \text{threads_in_space} (\text{thread_space} (\text{ tcb })) &:= \text{threads_in_space} (\text{thread_space} (\text{ tcb })) - 1 \end{aligned}$$

Note that this cannot actually be written at the top-level specification since it uses sequential composition. Parallel composition does not allow the same variable to appear twice on the left hand side of an expression.

CreateActiveThread

CreateActiveThread is a merger of CreateThread and ActivateThread, the only difference being that migrating the thread is not possible as it does not exist yet.

DeleteThread

The preconditions to successful thread deletion are that the thread exist and that it is not in one of the privileged spaces (kSigma0Space, kRootServerSpace, kKernelSpace).

The thread is then removed from the set of known, active, and halted threads.

Domain subtraction (see page 46) is used to remove the thread from the domains of all thread-based functions in the machine:

$$\begin{aligned} \text{thread_space} &:= \{ \text{ tcb } \} \Leftarrow \text{thread_space} \quad || \\ \text{thread_state} &:= \{ \text{ tcb } \} \Leftarrow \text{thread_state} \quad || \\ \text{thread_pager} &:= \{ \text{ tcb } \} \Leftarrow \text{thread_pager} \quad || \\ \text{thread_scheduler} &:= \{ \text{ tcb } \} \Leftarrow \text{thread_scheduler} \quad || \\ \text{thread_gno} &:= \{ \text{ tcb } \} \Leftarrow \text{thread_gno} \end{aligned}$$

Furthermore, if the thread is the only one left in the address space, the address space is deleted, otherwise the thread counter is decremented:

```

IF   { tcb } = thread_space-1 [ { thread_space ( tcb ) } ] THEN
    DeleteAddressSpace ( thread_space ( tcb ) ) ||
    threads_in_space := { thread_space ( tcb ) }  $\Leftarrow$  threads_in_space
ELSE
    threads_in_space ( thread_space ( tcb ) ) := threads_in_space ( thread_space ( tcb ) ) - 1
END

```

SetScheduler

For THREADCONTROL, modifying the thread's scheduler is one of the possible tasks.

This operation is trivial, assuming only that the thread and the scheduler exist, and updating the thread's scheduler.

Migrate

This operation performs the exact same task as the migration in ActivateThread (page 45).

As the explanation progresses it becomes obvious that there are entire blocks of statements being repeated throughout. Indeed, the next operation (MigrateAndSetScheduler) makes this quite clear.

During the development of the specification, a certain feature of the B Method in its current state caused a great deal of problems and excess complexity: two operations from the same level cannot be invoked in parallel *even if their statements are not related!* This means that Migrate and SetScheduler cannot be called in parallel and need a separate operation which combines the two.

One solution to this is to defer any specific actions to refined machines as opposed to a top-level specification, then use sequential composition in the refinements. The problem with this is that the resulting top-level machines cannot be animated (see 1.5) in any meaningful manner, making verification of *correctness* much more difficult and detracting from using the specification as a learning tool.

MigrateAndSetScheduler

As mentioned in the previous section, this is an exact merger of Migrate and SetScheduler. The preconditions are combined using ' \wedge ', and the statements of the two operations are composed in parallel.

SetState

As the name suggests, this sets the state *state* for the thread *tcb*. The preconditions are quite strong, to prevent the *aborted* state from being involved:

- $tcb \notin kIntThreads$ (since interrupt threads have their own meanings for states);
- $state \neq tsAborted$; transitioning to an *aborted* state would mean that the thread was somehow being deactivated. L4 does not provide any functionality to do this;
- $tcb \in active_threads$; we do not want to artificially activate the thread, that's what `ActivateThread` was intended for

ActivateInterrupt and DeactivateInterrupt

In L4, activating an interrupt thread is done by making it *halted* and setting its pager to a value other than itself. To deactivate it, the pager is set to itself and halting is reset.

In retrospect, the concept of *activation* for interrupt threads in my model is slightly mis-named, since they are always part of *active_threads*. Think of them as enabling and disabling interrupts.

Both operations assume that the thread passed in (*tcb*) is an interrupt thread. `DeactivateInterrupt` takes no further parameters (not necessary). `ActivateInterrupt` requires that a thread designated to be the interrupt's handler is supplied, which is set to be the interrupt thread's pager (see page 40 for details).

UnWait

Active threads participate in IPC. This means they may assume states involving waiting for an event (incoming IPC, delivery of outgoing IPC, or a time out). When this event occurs, they return to whatever state is appropriate for them. For normal threads, this is *running*, but for interrupt threads, this is *aborted* (page 40).

The operation assumes that the thread (*tcb*) is an existing one.

It is carried out as follows:

```
SELECT   tcb ∈ active_threads ∧ tcb ∉ kIntThreads   THEN
    thread_state ( tcb ) := tsRunning
WHEN     tcb ∈ active_threads ∧ tcb ∈ kIntThreads   THEN
    thread_state ( tcb ) := tsAborted
ELSE
    thread_state ( tcb ) := tsAborted
END
```

The `SELECT` statement in B is similar in structure to a *if-elseif-else* construct in programming languages, except that the testing order is *non-deterministic*. Only one of the cases will be

acted out: either *any* case where the condition matches, or the *else* case when none of them do.

When the cases are mutually exclusive and the order is not important, such as is the case here, they may be used as a simpler to read version of nested *if-then-else-end* blocks. They will become more important when evaluating error conditions in API-level machines.

WakeUpAndWait

When a running thread attempts to send an IPC to another thread, one of three things happen:

- the other thread is not waiting: the running thread polls — use `SetThreadState` (page 48)
- the other thread is waiting, no receive phase is included: the IPC occurs, the remote thread is woken — use `UnWait` (page 48)
- as above, but a non-trivial receive phase is included: the IPC occurs, the remote thread is woken, but the current thread starts waiting — use `WakeUpAndWait`

The operation takes three parameters: *running_thread* (the one sending the IPC), *waiting_thread*, and *wait_state* (specifies what kind of waiting the sending thread is to perform).

For this operation to succeed:

$$\begin{aligned} & \text{running_tcb} \in \text{active_threads} \wedge \\ & \text{waiting_tcb} \in \text{active_threads} \wedge \\ & \text{isWaiting} (\text{wait_state}) \wedge \\ & \text{isRunning} (\text{thread_state} (\text{running_tcb})) \wedge \\ & \text{isWaiting} (\text{thread_state} (\text{waiting_tcb})) \end{aligned}$$

Both threads must be active, the first must be running (`isRunning` tests for equality with `tsRunning`), the second must be waiting (`tsWaitingForever` or `tsWaitingTimeout`), and the `wait_state` must really be a waiting state (also `tsWaitingForever` or `tsWaitingTimeout`).

The operation overrides (see page 46) `thread_state` with two mappings:

- `running_tcb` \mapsto `wait_state`
- for normal waiting threads: `waiting_tcb` \mapsto `tsRunning`
for interrupt threads: `waiting_tcb` \mapsto `tsAborted` (see page 48)

ThreadExchangeRegisters

This operation comprises the thread-only functionality contained in the success path of `EXCHANGE_REGISTERS` [12, section 2.3]:

ThreadExchangeRegisters (*tcb* , *control* , *pager* , *unwait*)

The parameters are as follows:

tcb — the thread to act on

control — a subset of EXREGS_FLAGS, representing the set of actions the operation is to take (see 5.2.4)

pager — the pager to set the thread's pager to, if indicated by control

unwait — a Boolean value indicating whether the target thread should be woken (exactly like the UnWait operation); this is to correctly set the thread state if the IPC-level IpcBaseExchangeRegisters (see page 62) cancels the IPC waiting or polling.

Since the model contains no specification of TCRs at the thread level, the operation does not take an equivalent of the UserDefinedHandle parameter. Since there is also no specification of user-level registers saved in the kernel, IP, SP and FLAGS are not passed in.

Preconditions are as follows:

$$tcb \in threads \wedge control \subseteq EXREGS_FLAGS \wedge pager \in TCB \wedge \\ tcb \notin kIntThreads \wedge unwait \in BOOL$$

The operation can only work on threads known to the system. Since we are not dealing with the thread that invoked the EXCHANGEREGISTERS system call at this stage, we want to make sure that an interrupt thread is never the target. This is because interrupt threads are abstractions only and cannot invoke system calls, and EXCHANGEREGISTERS requires that both threads be in the same address space.

The pager can be any TCB; no checking is performed as outlined in [5]. The *unwait* parameter is a member of BOOL, which is either TRUE or FALSE.

The operation is constructed using three IF statements in parallel, corresponding to:

- setting the pager (if *ex_p* \in control)
- halting/resuming the thread (if *ex_h* \in control) — add the thread to *halted_threads* if (*ex_H* \in control), remove it otherwise
- resetting any waiting (if *unwait* == TRUE), only slightly different from UnWait in that we know the thread is not an interrupt thread, sets the state to *running* if the thread is active and *aborted* when it is not

DualWakeUp

Threads are not the only cause of IPC happening. When an IPC cannot be resolved immediately, the situation may arise that two threads, one polling on the second and the second waiting on the first, might be inside the system. It is then up to the scheduler to cause the IPC to happen. When it does, both threads need to be woken up and resume running. This is also true if the IPC fails.

Two threads are passed as arguments: *polling_thread* (which must be *polling*) and *waiting_thread* (which must be *waiting_timeout* or *waiting_forever*).

Since the same variable cannot occur twice on the left hand side of an assignment during parallel composition and the threads might be interrupt threads, a select covering the four possible cases is used, overriding *thread_state* with *running* for non-interrupt threads and *aborted* for interrupt threads.

5.5 IPC

5.5.1 Machine IpcCore

This machine EXTENDS the Thread machine, meaning it INCLUDES (see 3) it and also PROMOTES all operations.

A new definition is added for whether a thread can invoke the IPC system call (for interrupt threads, this means whether the kernel can perform the IPC on behalf of the thread):

$$\begin{aligned} \text{canIPC} (t) &\hat{=} \\ &t \in \text{active_threads} \wedge \\ &(t \in \text{kIntThreads} \Rightarrow t \in \text{halted_threads}) \wedge \\ &(t \notin \text{kIntThreads} \Rightarrow \text{thread_state} (t) = \text{tsRunning} \wedge t \notin \text{halted_threads}) \end{aligned}$$

The thread must be active, running and not halted (except for interrupt threads, which must be halted to be enabled).

The IpcCore machine does not seem very useful, seeing as it has no state and contains only a single operation (PerformIpc), which does nothing (denoted *skip* in B). This is because the IpcCore operation represents the transfer of information contained in Message Registers (MRs) from the sending to the receiving thread, but MRs do not exist in the current version of my specification. Refinement can add these and extend the IpcCore machine to do more (since *anything* refines *skip*, any functionality that preserves the invariant is permitted).

The decision to leave out MRs is due to them being a too large step to include in a top-level specification. When an IPC is performed, MRs do not merely get transferred, but can also perform grants and maps (see 2.5.1). Such functionality requires some concept of iteration over the MRs, which cannot be done in any readable manner (if at all) in a specification restricted to parallel composition. In refinement, sequential composition is permitted and an elegant manner with which to do this may be devised. Unfortunately, due to time constraints refinement was not an option.

Non-deterministic granting and mapping inside this operation could handle the possibility, but would not give it any meaning.

5.5.2 Machine IpcBase

This machine is the basis for all state transitions during IPC. It INCLUDES IpcCore and so builds on all machines described so far.

It does not promote any operations, however, even though there are operations which have nothing at all to do with IPC (such as `InitialiseAddressSpace`). This is due to the IPC operations being non-deterministic (i.e. there are situations which might cause it to fail which are not contained in this specification), which means that the first possibility of failure is in this machine (previous operations always succeeded given the preconditions). The error condition is stored in the Error Thread Control Register, which means some form of this TCR had to be specified in this machine.

Unfortunately, the inability to perform two operations from the same machine in parallel made every promoted operation forever unable to clear the Error TCR. This means local versions which only add that functionality shadow all operations which might normally be promoted.

The machine itself uses information on which thread is waiting/polling for which other thread to check when to allow the IPC to occur, and handles invoking the proper state-transition operations from Thread in these cases. It also invokes them when IPC fails.

A description of the variables and invariant follows.

Variable: `thread_ipc_waiting_for`

In L4, threads which are in a *waiting* state must be waiting for a specific thread number, or *anythread*. They cannot wait for *nilthread*. If a thread wants to make sure the waiting operation times out, it should wait for itself [12, section 5.6]:

$$\begin{aligned} & \text{thread_ipc_waiting_for} \in \text{active_threads} \leftrightarrow \text{GLOBAL_TNO} \wedge \\ & k\text{NilGNo} \notin \text{ran} (\text{thread_ipc_waiting_for}) \end{aligned}$$

Only active threads may participate in ipc, but they may wait for any thread number. The reference manual states that if the partner does not exist, the IPC operation will fail. However, the thread might exist when IPC is invoked, but be deleted before IPC completes, which is why `thread_ipc_waiting_for` cannot have *active_threads*, or even *threads* as its permitted range.

Variable: `thread_ipc_waiting_timeout`

The function has an identical range to `thread_ipc_waiting_for`, but specifies information on the timeout for waiting functions:

$$\begin{aligned} & \text{thread_ipc_waiting_timeout} \in \text{active_threads} \leftrightarrow \text{TIMEOUT} \wedge \\ & e\text{ZeroTimeout} \notin \text{ran} (\text{thread_ipc_waiting_timeout}) \wedge \\ & \text{dom} (\text{thread_ipc_waiting_timeout}) = \text{dom} (\text{thread_ipc_waiting_for}) \wedge \\ & \text{dom} (\text{thread_ipc_waiting_timeout}) = \text{thread_state}^{-1} [\{ \text{tsWaitingTimeout} , \\ & \quad \text{tsWaitingForever} \}] \end{aligned}$$

All the threads in the domain of this variable must either be waiting with a timeout, or waiting forever. No thread with that state must be absent, and no thread that is present may

have a different state. Since the domain is the same as that for `thread_ipc_waiting_for`, the constraint applies there as well.

Zero-timeout not be permitted, since those calls can be resolved immediately without forcing the thread to wait.

Variables: `thread_ipc_polling_on` and `thread_ipc_polling_timeout`

The first function keeps track of which thread a thread is polling on. However, despite the fact the thread it is polling on can be deleted, its range is a subset of *threads*, and not TCB. This is due to the target thread maintaining a list of threads polling on it, which is opposite to the way waiting works. When a thread gets deleted, its incoming queue is unwound. The implementation source code I am basing my specification on [5] further states:

```
// what do we do with these guys?
```

I have opted to do nothing and have the Ipc time out:

$$\begin{aligned} & \text{thread_ipc_polling_on} \in \text{active_threads} \leftrightarrow \text{threads} \wedge \\ & \text{thread_ipc_polling_timeout} \in \text{active_threads} \leftrightarrow \text{TIMEOUT} \wedge \\ & \text{dom} (\text{thread_ipc_polling_timeout}) = \text{thread_state}^{-1} [\{ \text{tsPolling} \}] \wedge \\ & \text{eZeroTimeout} \notin \text{ran} (\text{thread_ipc_polling_timeout}) \wedge \\ & \text{dom} (\text{thread_ipc_polling_on}) \subseteq \text{dom} (\text{thread_ipc_polling_timeout}) \end{aligned}$$

For this reason, there can be some threads in a polling state which are not actually polling for any thread.

The second function (`thread_ipc_polling_timeout`) keeps track of the time-outs for currently polling threads. *All* polling threads must have a timeout, even if the thread they are polling on was deleted, otherwise they will never return to running.

Once more, the zero-timeout is not permitted, since it can be resolved immediately.

Variables: `thread_rcv_waiting_for` and `thread_rcv_waiting_timeout`

These are the future versions of `thread_ipc_waiting_for` and `thread_ipc_waiting_timeout`, for polling threads with a receive phase. When the send phase of IPC succeeds, a receive phase is performed if one was requested. These variables hold those states until that point.

They are very similar to the `thread_ipc_waiting` variables, but:

$$\begin{aligned} & \text{dom} (\text{thread_rcv_waiting_for}) \subseteq \text{dom} (\text{thread_ipc_polling_timeout}) \wedge \\ & \text{dom} (\text{thread_rcv_waiting_timeout}) \cap \text{dom} (\text{thread_ipc_waiting_timeout}) = \{ \} \end{aligned}$$

Not all polling threads will have a receive phase, and no thread must have an entry in both the future and present waiting variables.

Variable: `thread_incoming`

For any thread, the set of threads polling on it is the relational inverse of itself on `thread_ipc_polling_on`:

$$\begin{aligned} & \text{thread_incoming} \in \text{active_threads} \rightarrow \mathbb{P}(\text{threads}) \wedge \\ & \forall tt . (tt \in \text{active_threads} \Rightarrow \text{thread_incoming}(tt) = \\ & \qquad \text{thread_ipc_polling_on}^{-1}[\{tt\}]) \end{aligned}$$

Note: \mathbb{P} has the same meaning as the standard notation for power set.

This variable is very helpful when it comes to modelling the details of IPC, since it enables a much simpler model of incoming thread numbers to be derived (which makes proof obligations simpler, which in turn makes it easier to find inconsistencies).

Variable: `thread_incoming_gnos`

The aforementioned model of incoming thread numbers makes checking whether or not an acceptable thread for receiving is polling much easier.

Please note that not all variables have to make it through refinement. This is especially true here, since L4 does not actually store the thread numbers of threads polling on another thread separately, but it makes the model conceptually simpler.

$$\begin{aligned} & \forall tt . (tt \in \text{active_threads} \Rightarrow \text{thread_incoming_gnos}(tt) = \\ & \qquad \text{thread_gno}[\text{thread_incoming}(tt)]) \end{aligned}$$

The incoming thread numbers for a thread are just the thread number function (`thread_gno`) applied to all incoming threads.

Variable: `thread_error`

The very simple concept of each thread having some error condition which resulted from a previous operation:

$$\text{thread_error} \in \text{active_threads} \rightarrow \text{ERROR}$$

For inactive threads (which cannot execute), the mapping has no meaning and so does not exist. Note that `thread_error` is not the same as the Error TCR [12], since `ERROR` contains `eNoError`, a condition to signify *success*.

In L4, success does not modify the Error TCR, since by convention it is irrelevant if the system call returned successfully. However, the success flag is contained in a register (registers are not specified in my model). This means a refinement of the concept of errors would be necessary.

Initialisation

Initially, no thread is waiting for any other thread or engaged in IPC in any way, meaning that all the `thread_ipc_*` variables as well as the `thread_rcv_*` variables are initialised to empty sets.

Since the interrupt threads, `sigma0` and the root server exist on start up, we want their incoming sets to be present, but empty:

$$\begin{aligned} \text{thread_incoming} & := \{ k\text{Sigma0} , k\text{RootServer} \} \cup k\text{IntThreads} \rightarrow \{ \{ \} \} \parallel \\ \text{thread_incoming_gnos} & := \{ k\text{Sigma0} , k\text{RootServer} \} \cup k\text{IntThreads} \rightarrow \{ \{ \} \} \end{aligned}$$

Any function mapping those threads to the the empty set (there is only one) will satisfy that requirement.

As for the error condition, all existing threads (the same ones as above) start out with the `eNoError` condition:

$$\begin{aligned} \text{thread_error} & := \{ k\text{Sigma0} \mapsto e\text{NoError} , k\text{RootServer} \mapsto e\text{NoError} \} \cup \\ & \quad k\text{IntThreads} \times \{ e\text{NoError} \} \end{aligned}$$

One can see that functions are just sets of mappings in B. The mappings for `sigma0` and the root server are one function, while the Cartesian product of the interrupt threads with `{eNoError}` is the second function. When added together, they enforce the invariant and do what is needed.

5.5.3 Machine `IpcBase`: Operations

ActivateThread2

The new variables introduced in this machine, together with the invariant of the included machines produce a new, larger invariant. Promotion of some operations causes the local invariant to be violated as the operations in lower machines know nothing about it.

Operations which introduce handling of `IpcBase`'s variables to operations from `Thread` or `AddressSpace` have a '2' appended to their end.

The first of these is `ActivateThread2`, which sets up the local variables when the thread is activated, and invokes the original `ActivateThread` (page 45). Their preconditions and parameters are the same, so let us look at the differences.

Recall that in `ActivateThread`, the thread state was set to *waiting forever*, but no mention of who to wait for was made. At the IPC level we can now say which thread the waiting will be for ... its pager:

$$\begin{aligned} \text{thread_ipc_waiting_timeout} (tcb) & := e\text{InfiniteTimeout} \parallel \\ \text{thread_ipc_waiting_for} (tcb) & := \text{thread_gno} (\text{pager}) \end{aligned}$$

The thread needs to receive a message from its pager before starting execution, so it must wait forever for it.

Since the error condition is meaningful for active threads, a condition must be set. No error has occurred, so the `eNoError` condition is used:

$$\text{thread_error} (tcb) := e\text{NoError}$$

Upon activation, the variables pertaining to incoming threads (those polling on the one currently being activated) must be set:

$$\begin{aligned} \text{thread_incoming} (tcb) &:= \text{thread_ipc_polling_on}^{-1} [\{ tcb \}] \parallel \\ \text{thread_incoming_gnos} (tcb) &:= \text{thread_gno} [\text{thread_ipc_polling_on}^{-1} [\{ tcb \}]] \end{aligned}$$

For `thread_incoming`, a new mapping is added: from the thread being activated (`tcb`) to the set of all threads polling on it (via relational inverse).

For `thread_incoming_gnos`, we would like to simply set it to be the numbers of whatever `thread_incoming(tcb)` was set to. Since this is parallel composition, using `thread_incoming` on the right-hand side of an assignment would refer to its *old* value, the expression is repeated and a relational image on `thread_gno` gives the desired result.

CreateActiveThread2

This operation adds the same steps as `ActivateThread` does to `ActivateThread2` to `CreateActiveThread`.

The preconditions and parameters are the same.

DeleteThread2

The preconditions are the same as in `DeleteThread` (page 46), and it still takes a single parameter: `tcb`.

Apart from invoking `DeleteThread`, it has to deal with deletion with regard to the variables in this machine.

Apart from the obvious domain subtraction of `{tcb}` from `thread_ipc_waiting_*`, `thread_ipc_polling_timeout`, `thread_recv_*` and `thread_error`, some variables require a more complex approach:

As mentioned on page 53, L4 stores the polling threads in the receivers incoming queue, so it does not suffice to delete the thread from another's incoming set, since its own incoming set might not be empty:

$$\text{thread_ipc_polling_on} := \{ tcb \} \triangleleft \text{thread_ipc_polling_on} \triangleright \{ tcb \}$$

The application of the domain subtraction (\triangleleft , precedence is left-to-right) removes all mappings denoting this thread is polling on another one (there is only one). Then, the application of range subtraction (\triangleright) removes all mappings denoting another thread is polling on this one. Those threads that were polling on the one being deleted are now stranded until their IPCs time out (this might not be the case with the newest kernels, see page 53).

Range subtraction is defined as follows:

$$r \triangleright S = \{ x \mapsto y \mid x \mapsto y \in r \wedge y \notin S \}$$

This resolves the situation of who is polling on whom. However, the incoming sets still have to be adjusted:

$$\text{thread_incoming} :=$$

$$\begin{aligned}
& \{ aa , bb \mid aa \in \text{dom} (\text{thread_incoming}) - \{ tcb \} \wedge \\
& \quad bb \in \mathbb{P} (TCB) \wedge \\
& \quad bb = \text{thread_incoming} (aa) - \{ tcb \} \} \parallel \\
\text{thread_incoming_gnos} := & \\
& \{ aa , bb \mid aa \in \text{dom} (\text{thread_incoming_gnos}) - \{ tcb \} \wedge \\
& \quad bb \in \mathbb{P} (GLOBAL_TNO) \wedge \\
& \quad bb = \text{thread_incoming_gnos} (aa) - \{ \text{thread_gno} (tcb) \} \}
\end{aligned}$$

The thread has to be removed from both the variables' domains and *simultaneously* be removed from the incoming sets of every active thread in the system. The only way such a broad change can be made in B is to use a *set comprehension*.

The notation is very similar to the list comprehensions used in Haskell, and its definition is as follows:

$\{z \mid P\}$ yields all z that satisfy P

Note that (aa, bb) and $aa \mapsto bb$ are synonymous. Their variation in appearance is the result of the B Toolkit's mark-up tool.

Let us look at the first set comprehension in detail (the second is analogous, but removes a thread number from a different variable). We want the set of all mappings $aa \mapsto bb$, such that:

- The function domain is still represented, with the exception of tcb , whose mappings are removed: $aa \in \text{dom} (\text{thread_incoming_gnos}) - \{ tcb \}$
- The function range still has the same type, i.e. is a subset of TCB: $bb \in \mathbb{P} (GLOBAL_TNO)$
- Finally, we want tcb to be removed from all values thread_incoming may assume.

JustWait

The first of the actual operations enabling IPC. This deals with the case when a thread requests an IPC operation consisting of a receive phase only, but no thread in its incoming sets is available to receive from. This results in the thread waiting.

Given the three parameters tcb (the thread wishing to receive), $timeout$ and $fromSpecifier$ (who it is willing to receive from), the preconditions are as follows:

- the thread can participate in IPC (see `canIPC` definition, page 51)
- $timeout$ is either finite or infinite, but *not zero* (instant time-out, no point in calling this operation)
- $fromSpecifier$ is not $nilthread$, and is either one of the thread numbers known to the system (`thread_gno[threads]`) or it is *anythread*

- a *fromSpecifier* of *anythread* implies that there are no incoming threads for the requester (otherwise, the thread would not need to wait)
- any other *fromSpecifier* is not in the set of incoming thread numbers (again, waiting is what this operation is about)

The work done by the operation is minimal. It updates `thread_ipc_waiting_for` and its time-out equivalent (page 52) to indicate the thread is waiting and who it is waiting for. It also uses `SetState` (page 48) to set the thread's state to *tsWaitingForever* or *tsWaitingTimeout* depending on the value of *timeout*.

SetUpReceivePhaseAndPoll

We have covered what happens when a thread wants to receive and cannot. This operation handles the case of when the operation wants to *send* but cannot (either the remote thread is not waiting, or it is not waiting for the sending thread).

Parameters: *tcb_from* and *tcb_to* (sending and target threads), *poll_timeout*, *recv_timeout* (time-out for the future receive phase), *fromSpecifier* (who the thread is willing to receive from in the receive phase, or *nilthread* if there is no receive phase).

As in `JustWait`, *tcb_from* must be able to perform IPC. The target must be an existing thread. While the poll time-out must not be zero, the receive timeout is only restricted if *fromSpecifier* is not *nilthread*. The actual *fromSpecifier* must be a thread number of an existing thread.

In order to verify that the operation happens in the aforementioned circumstances, the following must be true:

$$\begin{aligned}
 & (tcb_to \in \text{dom} (thread_ipc_waiting_for) \Rightarrow \\
 & thread_ipc_waiting_for (tcb_to) \neq thread_gno (tcb_from) \wedge \\
 & thread_ipc_waiting_for (tcb_to) \neq kAnyGNo)
 \end{aligned}$$

In other words, if the target thread is in a waiting state, then it must not be waiting for *anythread* (since this one will fulfil the criterion) and it must not be waiting for *from_tcb*'s number.

Once that is established, the operation can proceed successfully.

The operation updates `thread_ipc_polling_*` to reflect its polling information, and also adds *from_tcb* and its thread number to the incoming sets of *tcb_to*.

If *fromSpecifier* is not *nilthread*, `thread_rcv_*` is updated with the future waiting information.

JustReceive

Having covered the cases where IPC cannot happen, let us look at the simplest case of IPC occurring: the thread requests an IPC with only a receive phase, and a suitable thread is in its incoming set.

Like JustWait, it takes two parameters (whose meaning is the same): *itcb* (the invoking thread) and *fromSpecifier* (who it wishes to receive from). It does not need a time-out as the operation will go ahead immediately.

The value of *fromSpecifier* must not be *nilthread*, and must either be *anythread* (in which case the incoming set must not be empty) or a thread number already in the incoming set.

The operation may then go ahead. However, it might not succeed due to aspects beyond the control of the current model (such as Xfer time-outs or the operation being aborted halfway). This failure is modelled by non-determinism and is the cause of putting the *thread_error* variable in this machine (see page 54).

The polling thread which is allowed to send is again chosen non-deterministically (since sets have no implicit ordering):

$$\begin{aligned} \text{ANY } tcb_from \quad \text{WHERE } & tcb_from \in thread_incoming (itcb) \wedge \\ & (fromSpecifier \neq kAnyGNo \Rightarrow \\ & thread_gno (tcb_from) \in thread_incoming_gnos (itcb)) \end{aligned}$$

In other words, choose any of the threads in the incoming set, with the extra constraint that if *fromSpecifier* is not *anythread*, that thread's number must be in the set of incoming thread numbers for the receiving thread. The preconditions guarantee that a thread that satisfies this constraint actually exists.

Regardless if the IPC succeeds or fails, the following happens:

$$\begin{aligned} thread_ipc_polling_on & := \{ tcb_from \} \triangleleft thread_ipc_polling_on \quad || \\ thread_ipc_polling_timeout & := \{ tcb_from \} \triangleleft thread_ipc_polling_timeout \end{aligned}$$

The sending thread will no longer be polling at the end of the operation.

$$\begin{aligned} thread_incoming (itcb) & := thread_incoming (itcb) - \{ tcb_from \} \quad || \\ thread_incoming_gnos (itcb) & := thread_incoming_gnos (itcb) - \{ thread_gno (tcb_from) \} \end{aligned}$$

Since it will no longer be polling, it is removed from the receiving thread's incoming sets.

$$\begin{aligned} thread_recv_waiting_timeout & := \{ tcb_from \} \triangleleft thread_recv_waiting_timeout \quad || \\ thread_recv_waiting_for & := \{ tcb_from \} \triangleleft thread_recv_waiting_for \end{aligned}$$

Again, since it will not be polling, the future settings for its receive phase will no longer be applicable. If the IPC succeeds, they will be used to set up the new receive phase for the thread. If IPC fails, they will be discarded.

Now then comes the point where the IPC succeeds or fails. This is done using the non-deterministic CHOICE *path1* OR *path2* END construct. During animation, the user is asked to choose the path. During proof, both branches are checked to see if they preserve the invariant.

Let us examine the success path for IPC.

Firstly, the IPC transfer is performed and the error fields for both threads are cleared:

$$\begin{aligned} PerformIPC (tcb_from , itcb) & \quad || \\ thread_error & := thread_error \triangleleft \{ itcb \mapsto eNoError , tcb_from \mapsto eNoError \} \end{aligned}$$

Then, if the sender had a receive phase waiting, set that up (using identical statements to those in JustWait on page 57). Note that the receiving thread's state does not change. It was either running or an activated interrupt thread, and remains so.

If the sender does not have a receive phase waiting, its waiting state is cancelled using UnWait (page 48).

On the other side of the *OR*, the failure path: the sender's waiting state is cancelled using UnWait and an error is picked non-deterministically among the possible unpredictable IPC errors (see 5.2.8) and assigned as an error indicator for both threads.

WakeDestThenWait

If a thread wishes to send and the second thread is waiting, the IPC occurs immediately, the destination thread is woken up, while the source thread starts waiting (if a receive phase was specified). In L4, a thread switch to the destination is also performed.

The precondition combines aspects of the previous IPC operations:

- The destination must be waiting for either the source's thread number or *anythread*;
- The *fromSpecifier* must be that of an existing thread, *nilthread* or *anythread*;
- A non-*nilthread fromSpecifier* indicates a receive phase for the source and so *recv_timeout* must not be zero;
- There is no polling timeout, since the operation goes ahead immediately.

This time there are no common items between the success and failure paths. The non-deterministic CHOICE is made once more.

The success path begins as previously, by performing the IPC transfer and clearing the error indicators for both threads.

If *tcb_from* (the source thread) did not request a receive phase, the operation can be quickly finished by domain subtraction of *tcb_to* from *thread_ipc_waiting_** and using UnWait to cancel its waiting state.

If it *did* request a receive phase, then the situation is more complicated. The destination still has to be removed from *thread_ipc_waiting_**, but now the source thread must also be inserted. The first half of the IF statement is presented below, for when the time-out is infinite. The second half is analogous, but the timeout is finite and so the state will be *tsWaitingTimeout*.

```

thread_ipc_waiting_for := { tcb_to }  $\Leftarrow$  thread_ipc_waiting_for  $\cup$ 
                          { tcb_from  $\mapsto$  fromSpecifier } ||
IF   isInfinite ( recv_timeout ) THEN
  thread_ipc_waiting_timeout :=
  { tcb_to }  $\Leftarrow$  thread_ipc_waiting_timeout  $\cup$  { tcb_from  $\mapsto$  eInfiniteTimeout } ||
  WakeUpAndWait ( tcb_from , tcb_to , tsWaitingForever )

```

WakeUpAndWait (page 49) is used to wake up *tcb_to*, and make *tcb_from* with one of the time-outs.

The **failure** path on the other hand just removes the destination from *thread_ipc_waiting_**, picks an error and sets it as the error attribute for both threads, and also uses UnWait on the destination thread.

ResolveIPC

Given the above operations, the situation where one thread₁ is polling on thread₂, while the latter is waiting for the former. Since neither of them can execute, the kernel needs a way to internally perform the IPC, which is what this operation is for.

ResolveIPC(*tcb_from*, *tcb_to*) requires that:

- Both threads are active;
- The sender is polling and the receiver is waiting;
- The sender is polling on the receiver;
- The receiver either accepts *anythread* or the receiver's thread number;

The non-deterministic IPC part decides between:

- Performing the IPC transfer and clearing the error attribute for both threads or
- Not doing anything, choosing an error non-deterministically and assigning it as the error indicator for both threads.

The rest of the statements are common to both success and failure paths and consist of removing both threads from the state variables (including removing the sender from the receiver's incoming sets). This has been covered in previous operations in this machine.

TimeoutPoll

When the kernel finds a thread that's been polling for longer than its time-out value, a time-out occurs. Since the model abstracts away exact values for time-outs, this must be done non-deterministically. This operation picks any thread which is polling with a non-infinite time-out and times it out. If such a thread does not exist, it does nothing (*skip*).

Of course, non-determinism is not random, it just states that the decision algorithm is not specified at this level. During animation, the user is asked to be that algorithm (see 1.5).

The operation removes the thread from the state variables (covered previously) and sets its error attribute to *eSendTimeout*.

TimeoutWait

This operation is the equivalent of TimeoutPoll, but times out a thread which is waiting with a finite time-out.

SetError

SetError provides a way for operations in higher-level machines to set the error attribute for an active thread without actually doing anything. This is used for example, to signal that a thread lacks necessary privileges to perform an operation.

IpBaseExchangeRegisters

When EXCHANGEREGISTERS functionality was last described (ThreadExchangeRegisters, page 49) only the functionality pertaining directly to threads and state was covered. As the reference manual [12, section 2.3] states, EXCHANGEREGISTERS can be used to cancel or abort ongoing IPCs. Now that the IPC state transitions are available, the IPC functionality in EXCHANGEREGISTERS can be modelled.

It takes one fewer parameter than ThreadExchangeRegisters, since it is the one that decides whether a waiting/polling thread is to be woken up.

The preconditions, with the exception of the *unwait* flag are identical.

The functionality at the IPC level consists of the following bits in *control*:

- If $S = 1$, a currently ongoing send IPC operation will be *aborted*, while an IPC send operation waiting to happen will be *cancelled*;
- If $R = 1$, as above, but for receiving IPC.

In the current model, bits are not used. Instead, the bits are represented by set membership of *ex_S* and *ex_R* in *control*.

If neither are present, the operation invokes ThreadExchangeRegisters with *unwait* set to FALSE (do not change the state) and clears the error attribute.

If *ex_S* is present, the operation is removed from the state variables to do with polling, as well as from the incoming set of the thread it is polling on. Since at the top specification level in B operations happen instantaneously, it is impossible to determine whether the IPC operation was *cancelled* or *aborted*, so a non-deterministic choice is made and becomes the value of the thread's error attribute. ThreadExchangeRegisters is invoked with the *unwait* flag equal to TRUE, forcing the function to be awakened.

If *ex_S* is present, events proceed as above, except the thread is removed from state variables related to *waiting*.

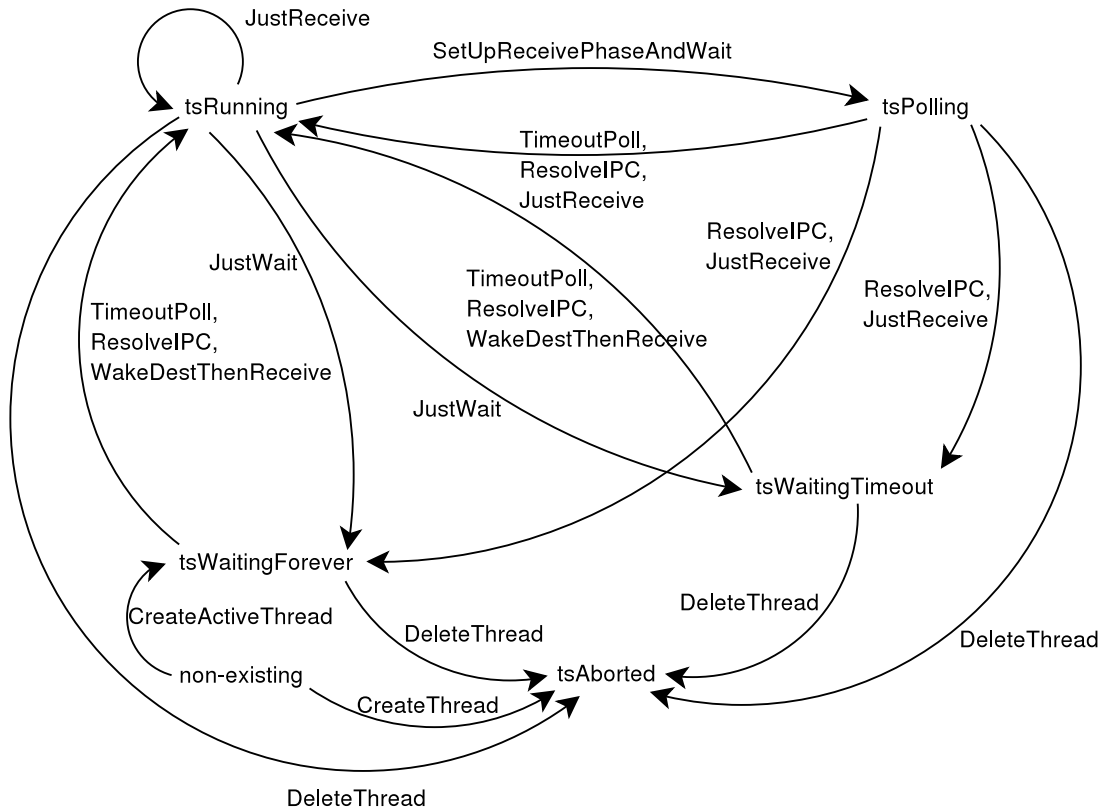


Figure 5.3: Possible state transitions in the model and operations which cause them.

The error additions to Thread's operations

In order for operations which complete successfully to clear the error attribute of a thread, they need to be extended with that functionality at this level. Their preconditions are almost the same as their Thread counterparts', and the operation body invokes them directly. The only difference is that they need to take an extra parameter (*itcb*) in order to know which thread's error attribute should be cleared. The digit '2' has been appended to their names. If it wasn't for the error attribute being in this machine they would have been promoted.

They are: InitialiseAddressSpace2, CreateThread2, SetScheduler2, Migrate2, MigrateAnd-SetScheduler2, ActivateInterrupt2 and DeactivateInterrupt2.

Updated state diagram

Now that all the core functionality present in the model has been defined, a more accurate view of state transitions, such as that in figure 5.3 can be derived.

5.6 API

5.6.1 Machine WeakSyscall

All the functionality that is present in the model has now been defined. The final level is to determine which bit of functionality to use when, and build upon the operations to allow them to take thread numbers (like L4 system calls) instead of internal TCBs as they did before this point.

Since the resulting operations would be extremely long, an extra machine in which operations add functionality but as little error handling capability as possible has been added below the API level. Some of the resulting operations are quite long and complex despite this.

The machine naturally *INCLUDES* IpcBase (see 5.5.2), but it also promotes the following operations:

- SetError - to allow the API machine to handle errors;
- TimeoutWait, TimeoutPoll and ResolveIPC - while not presented in the L4 API, promoting them to the API level allows for more effective animation;
- InitialiseAddressSpace2 and IpcBaseExchangeRegisters - the invoking operations in the API machine are not too long, making the extra level of indirection unnecessary.

To prevent namespace collisions (there is only one, after all), operations in this machine are prefixed with “Weak”.

WeakIpc

This operation has moderate error-handling capability, which makes it very long and riddled with IF statements. See figures 5.4 and 5.5 for the exact flow of statements inside it. All syntax used in these machines has already been introduced and all control flow decisions are based on preconditions of the invoked operations, so diagrams will be of more value than a textual description.

Parameters:

- *itcb* - the thread invoking the operation
- *to_gno* - the thread number of the target thread or *nilthread* if no send phase
- *fromSpecifier* - who *itcb* is willing to receive from, *nilthread* if no receive phase
- *send_timeout* and *recv_timeout* - how long the thread is willing to wait for sending and receiving

Non-trivial preconditions:

- If *to_gno* is not *nilthread*, it must be a number of an existing thread;

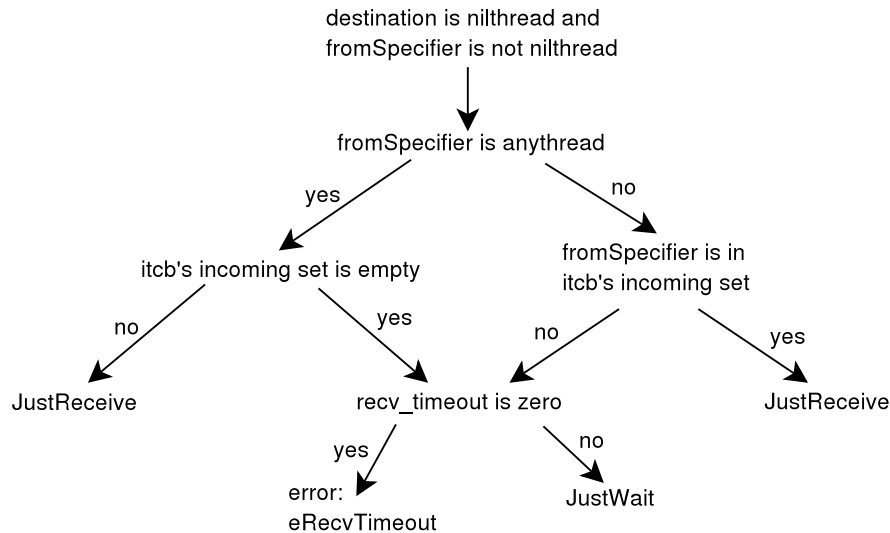


Figure 5.4: The receive phase only section of WeakIpc

- *fromSpecifier* is either the thread number of an existing thread in the system, *nilthread* or *anythread*.

The error-handling capability greatly reduces preconditions.

The body of the operation is divided by a SELECT statement into three cases:

- No send phase and no receive phase — `eNonExistingPartner` is set for *itcb*'s error attribute;
- No send phase, but a receive phase — described in figure 5.4
- A send phase and optional receive phase — described in figure 5.5.

Note that the diagrams denote the *functionality* contained in the operation rather than the exact one-for-one correspondence with the B specification. The B specification is longer due to statement duplication being sometimes unavoidable, and sometimes helpful with generating easier-to-discharge proof obligations. This also makes it more confusing, making simple diagrams more helpful.

WeakDeleteThread

This operation adds the following to DeleteThread2:

- The thread to delete (*dest*) is now chosen using its thread number;
- The invoking thread is passed in as *itcb*;
- The precondition requires that the invoking thread reside in a privileged address space.

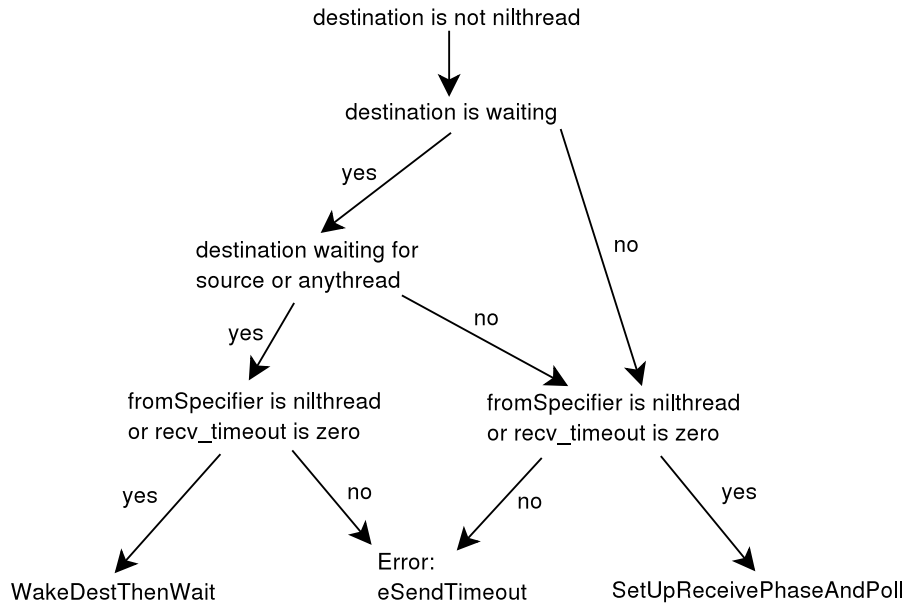


Figure 5.5: The send phase with optional receive phase section of WeakIpc

WeakModifyThread

This operation manages thread activation, setting the scheduler and space migration.

Parameters:

- *itcb* — invoking thread
- *destNo* — thread number of target thread
- *spaceSpecifier* — a thread number of a thread in the address space we want to specify (or *nilthread* if the address space will not change)
- *schedNo* — thread number for the scheduler (*nilthread* if no modification)
- *pagerNo* — thread number for the pager (*nilthread* if no modification)

The operation does not perform any error handling and its preconditions imply a successful outcome:

- *itcb* is an active thread in a privileged address space;
- *destNo* is a thread number of an existing thread;
- Neither *spaceSpecifier*, *schedNo* nor *pagerNo* may be *nilthread*;
- Modification of the address space (*spaceSpecifier* \neq *nilthread*) implies that *fromSpecifier* is a thread number of an existing thread. The same conditions apply to modifying the pager and scheduler for *pagerNo* and *schedNo* respectively;

- If the thread is to be activated (target is not active and *pagerNo* \neq *nilthread*) then the following conditions must hold: a *schedNo* of *nilthread* implies that the scheduler will not be modified and the thread's existing scheduler must be active, while a *schedNo* that is not *nilthread* implies that the new scheduler must be active. An active scheduler is a prerequisite for activation (see `ActivateThread` on page 45);
- If migration is necessary (the address space indicated by *spaceSpecifier* is different than the thread's current space) requires that the target space for migration can hold the target thread.

Three possible branches in the function body exist. Firstly, if all the modification specifiers (*spaceSpecifier*, *schedNo* and *pagerNo*) are *nilthread*, the function modifies nothing, and so *skips*.

When a pager change is requested (*pagerNo* \neq *nilthread*), all cases include thread activation using `ActivateThread2` (page 55). For all the cases, the target thread is the one whose number is *destNo*, and the pager is the thread with *pagerNo*. For the other two parameters (*space* and *scheduler*), four cases are covered given two possibilities for each:

- If *spaceSpecifier* is *nilthread* no migration is performed, *space* is the address space of the target thread. Otherwise, *space* is the space indicated by *spaceSpecifier*;
- If *schedNo* is *nilthread*, *scheduler* is the thread's current scheduler. Otherwise, it is the thread whose number is *schedNo*.

When no pager change is necessary, the operation either performs a migration (`Migrate2`), a change of scheduler (`SetScheduler2`), or both (`MigrateAndSetScheduler2`), depending on whether *schedNo*, *spaceSpecifier*, or neither are *nilthread* (respectively).

IntThreadControl

Like in the L4 source code [5, `interrupt.cc`], I have put the equivalent of the `THREADCONTROL` system call for interrupt threads in a separate operation. Interrupt threads experience a different view of `THREADCONTROL`: either they are disabled (by setting the thread's pager to itself) or disabled (by setting it to any other thread).

This operation combines `ActivateInterrupt` and `DeactivateInterrupt` (page 48) and provides the functionality of specifying the threads by number.

The operation, as dictated by the `THREADCONTROL` specification [12] is limited to privileged threads only. Again, since there is no non-determinism in its specification, satisfying the preconditions guarantees success.

WeakCreateThread

This operation constitutes the thread-creating portion of `THREADCONTROL`. Its actions are as follows:

- If the pager number supplied is *nilthread*, the thread is created inactive (`CreateThread2`). If the space specifier is equal to the target thread number, a new address space is chosen non-deterministically and passed in as *space*. Otherwise use the address space selected by *spaceSpecifier*;
- If the pager number is not *nilthread*, then proceed as above, but create an active thread (`CreateActiveThread2`) and pass the pager as another parameter.

5.6.2 Machine API

This is the topmost machine in the specification. It `INCLUDES` `WeakSysCall` and all the context machines.

Operations in API are either direct equivalents of L4 system calls, or operations representing system internals for use in animation. Their only real task at this level is to provide pre-condition support to lower-level operations (such as those in `WeakSyscall`) and pick which of these operations to invoke. They are very simple, if sometimes long, and are better examined directly (see appendix A).

What is worth noting however is that the top-level system-call operations still have preconditions: the invoking thread must be active and running, otherwise the system scheduler is fundamentally broken and *nothing* will work.

Chapter 6

Discussion and Critique

Completeness

The model presented in chapter 5 represents nearly all the behaviours in the L4 microkernel at an abstract level. Some behaviours, such as state transitions, are modelled accurately, while others, such as memory management, are modelled using non-determinism.

Since this particular model did not make it into the refinement stages, these behaviours remain non-deterministic.

In retrospect, obtaining the release version of L4 [5] at the beginning of the thesis (February-March 2004) and choosing not to follow updates beyond that point caused some inconveniences. While helpful at the beginning, near the end it was obvious that some features which were unimplemented or ignored in my version *had* in fact been implemented. Apart from the lack of IPC redirectors (see 5.1, page 23) no harm seems to have been done.

Unnecessary constraints

L4 is unlike any system that I have modelled before in B. Unlike those systems (e.g. reservation systems), L4 does not place the same emphasis on the integrity of data contained inside it. For example, a thread's scheduler can be any thread, even one that has not yet been created. The same goes for the pager. This means there can be threads running without pagers and schedulers or worse, with invalid ones. Naturally this is done to improve efficiency (the number one priority in L4): when deleting a thread, looking through all the other threads trying to see which one of those has the deleted thread as a pager is slow. It is better to leave it and re-evaluate the issue when it comes up again (e.g. when a page fault occurs). These situations can be prevented by using thread versions in L4, but those too are left to the person using the kernel to write an OS.

My initial understanding of L4 did not match the above description, and I tried to assert "sane" and "logical" conditions upon the kernel. When closer examination revealed these constraints unnecessary, they had to be removed. Without someone with a great deal of L4 internals experience, it is impossible to be sure all of them were removed.

This is not to say that putting some of those conditions into the kernel would actually be a bad idea. I believe that simple-to-enforce constraints that can prevent problems later (such as a thread being its own pager for non-interrupt threads) could easily be added. My understanding of the view the L4 community holds on the issue is “if you allow one, you allow them all”. Adding these helpful constraints all through the kernel will definitely slow it down a bit.

Structure and animation

In the goals for this model (page 24), the ability to animate it is listed. This, while useful for learning and testing correctness, became a great inconvenience for structuring the development. Animation can only be done at the top specification level. The top specification level only allows parallel composition. This means to animate concepts which are difficult or impossible using parallel composition means to create statement and operation duplication (page 26), highly complex statements (which are difficult to prove later), or non-determinism to model that which cannot be done in parallel.

While following that path will lead to the same destination, it will take a very long time. This is compounded by the fact that introducing large concepts all at once creates difficult to discharge proof obligations, making consistency proof a painful experience.

The other way of doing things is to model the top layer as purely non-deterministic. For example, this means that given a function f and the need to both change $f(a)$ and $f(b)$, instead of using mappings and right overriding, one can simply assign a new function non-deterministically which has the desired properties.

This would render the top-level specification next to useless for animation, since it could only check user input for validity, and inputting whole functions by hand, while very educational, does not allow animation to be efficiently used.

However, the top-level specification will be simpler, hopefully easier to complete, and easier to prove consistent. Then refinement, with its sequential composition can be employed.

While I cannot guarantee that this approach will yield better results than the one employed in this thesis, it could perhaps yield them *faster*.

The future of the B Method is Event B, which takes the alternate solution to a complete extreme, and any further attempts to model systems such as this one should definitely consider using it instead.

Proof and consistency

Presently, all machines below and including Thread have been proven to be consistent. Proof of IpcBase and machines above it has not been completed due to the complexity of the proof obligations (which means they cannot be discharged automatically) and the outdated proving system available in the B Toolkit.

The system requires that when a proof obligation cannot be discharged because a proof path is not available, new rules must be added into the system. Since proof is purely the

rewriting of sequences of tokens, inserting these new rules must be done extremely carefully, or something which is false can be proven as true. Additionally, the automatic prover often goes into an infinite loop on user rules, making discharging proof obligations a very long-winded experience.

There is a better way however. The next incarnation of the B Toolkit, Atelier B [4] provides a completely different method of interactive proof based on suggesting and then proving hypotheses. This means the user need only add rules when *strictly* necessary. Even then, Atelier B provides a way to prove the rules themselves.

Unfortunately, Atelier B is not a free tool, and its licenses are only available while on the university network. Furthermore, it was not functional throughout the year and I have had very little experience with it, not enough to convert the entire project to it so late in the year.

Atelier B and the B Toolkit are not entirely compatible when it comes to the B syntax. Also, the University does not have a licence for the animator plug-in, thereby making one of my primary goals impossible to achieve.

Chapter 7

Conclusion

A formal model of the L4 microkernel API has been presented, along with several aspects of its internal structure and functioning. It is well-documented and may be animated to further check for correctness. Proof of consistency has not been completed, but should not be attempted without moving the development to the new proof system [4]. Refinement has not been attempted, but the specification has been designed in such a way as to make this as painless an experience as possible.

7.1 Future Work

7.1.1 Complete Proof of Consistency

Now that Atelier B is running and available, proving the entire specification consistent can be completed. Moving it over to Atelier B, the animation component is lost, so modifications can be made to make proof even easier. Features can be removed and added into refinements, etc.

7.1.2 Verifying Correctness

I have spend a great deal of time trying to understand the L4 internals and how they apply to the API. Despite best efforts, it is quite likely not everything that the model does reflects what the system should be doing. Occasionally during discussions with L4 personnel, what this the correct action was could not easily be decided.

The difference was that during those meetings no model of L4 existed. A model can resolve many such arguments once it is decided by a majority to be correct. If it is not correct, it can be modified, until agreement is reached.

7.1.3 The L4 Pilot Project

Once the slice [21] is completed, my model can aid in the decision which part of the system will be next to undergo the complete verification procedure. Once this is done, the model can form a basis for the next slices (which will have to be converted to Isabelle/HOL) and then retired. Alternatively it can be kept in use as a live reference to the system.

The new security API [16] is also currently beginning development, which can slot into the new model, be helped or inspired by the model, or even use the model as a guide for what *not to do*.

7.1.4 Further Research Using the B Method

As discussed in 6 (page 69), the fact L4 emphasizes efficiency above all else makes it difficult to think about in terms of formal verification. It is possible to sacrifice some of this efficiency and still obtain a robust kernel. A possible direction for future research is to use Event B to model a slightly different kernel than L4 itself (but still inspired by it), call it BL4. This way some kind of verified kernel might be produced faster than trying to formally verify all the intricacies of speed optimisations within L4, while avoiding the problems associated with classical B.

Bibliography

- [1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] B-Core. The B-Toolkit. 2002.
- [3] William R. Bevier. *A Verified Operating System Kernel*. PhD thesis, University of Texas, 1987.
- [4] ClearSy. Atelier B. 2003.
- [5] L4 development team. L4 pistachio source code, release version 0.3, (generic api code in kernel/src/api/v4/). 2004.
- [6] L⁴Linux development team. L⁴linux home page. 2003.
- [7] Gregory Duval and Jacques Julliand. Modelling and verification of the RUBIS μ -kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.
- [8] Simon Fowler and Andy Wellings. Formal analysis of a real-time kernel specification. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135, pages 440–458, Uppsala, Sweden, 1996. Springer-Verlag.
- [9] Ahmed Helmy. A survey on kernel specification and verification. Technical Report 97-654, University of Southern California, 1997.
- [10] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.
- [11] Kernel.org. Linux source code directory, 2.6.x. 2004.
- [12] *L4 Reference Manual*, January 2004.
- [13] Benjamin James Leslie. Mungi device drivers, 2002.
- [14] Jochen Liedtke. Towards real μ -kernels. *Communications of the ACM*, 39(9):70–77, 1996.
- [15] Jae Yun Moon and Lee Sproull. Essence of distributed work: The case of the linux kernel. *First Monday*, (5), 2000.
- [16] Kevin Elphinstone (NICTA). Future directions in the evolution of the L4 microkernel (presented at the NICTA OS Verification Workshop). 2004.

- [17] University of Cambridge. Isabelle/HOL. 2004.
- [18] J. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS97-26, University of Pennsylvania, Philadelphia, PA, USA, 1997.
- [19] J. Michael Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5):21–28, September 1990.
- [20] A. Tannenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.
- [21] Harvey Tuch and Gerwin Klein (NICTA). Verifying the L4 virtual memory subsystem (presented at the NICTA OS Verification Workshop). 2004.
- [22] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.
- [23] Yuan Yu. *Automated proofs of object code for a widely used microprocessor*. PhD thesis, University of Texas at Austin, 1992.

Appendix A

Final B Specification

A.1 API

MACHINE *API*

SEES

FpageCtx ,
Bool_TYPE

INCLUDES

KernelInformation , *ThreadIdCtx* , *ThreadStateCtx* , *AddressSpaceCtx* ,
TimeoutCtx , *ThreadCtx* , *ErrorCtx* , *WeakSyscall*

PROMOTES

An action performed by the scheduler. Promoted for purposes of animation.

ResolveIPC

OPERATIONS

Note: Certain preconditions must hold even at the top level, such as the fact that a thread which should not be scheduled isn't. If the schedule operation is used, then these conditions will be satisfied. Type information must also be assumed.

Ipc (*itcb* , *to_gno* , *fromSpecifier* , *send_timeout* , *recv_timeout*) $\hat{=}$
PRE $canIPC (itcb) \wedge to_gno \in GLOBAL_TNO \wedge fromSpecifier \in GLOBAL_TNO \wedge$
 $send_timeout \in TIMEOUT \wedge recv_timeout \in TIMEOUT$
THEN
SELECT $to_gno = kAnyGNo$ **THEN**
 $SetError (itcb , eSendNonExistingPartner)$
WHEN $fromSpecifier \notin (thread_gno [threads] \cup \{ kAnyGNo , kNilGNo \})$ **THEN**

```

    SetError ( itcb , eRecvNonExistingPartner )
WHEN   to_gno  $\neq$  kNilGNo  $\wedge$  to_gno  $\notin$  ran ( thread_gno ) THEN
    SetError ( itcb , eSendNonExistingPartner )
ELSE
    WeakIpc ( itcb , to_gno , fromSpecifier , send_timeout , recv_timeout )
END
END ;

```

This is not a system call, but represents the timing out of a single IPC for purposes of animation (this is performed internally to L4 by the scheduler).

```

TimeoutIPC  $\hat{=}$ 
CHOICE
    TimeoutPoll
OR
    TimeoutWait
END ;

```

```

ThreadControl ( itcb , destNo , spaceSpec , schedNo , pagerNo )  $\hat{=}$ 
PRE   itcb  $\in$  active_threads  $\wedge$  thread_state ( itcb ) = tsRunning  $\wedge$ 
    destNo  $\in$  GLOBAL_TNO  $\wedge$ 
    spaceSpec  $\in$  GLOBAL_TNO  $\wedge$ 
    schedNo  $\in$  GLOBAL_TNO  $\wedge$ 
    pagerNo  $\in$  GLOBAL_TNO
THEN

    SELECT    $\neg$  ( dIsPrivilegedSpace ( thread_space ( itcb ) ) ) THEN
        SetError ( itcb , eNoPrivilege )
    WHEN   spaceSpec = kNilGNo THEN
        Thread deletion.

    IF   destNo  $\in$  ran ( thread_gno )  $\wedge$  destNo  $\neq$  thread_gno ( itcb ) THEN
        IF   thread_space ( thread_gno-1 ( destNo ) )  $\neq$  kSigma0Space  $\wedge$ 
            thread_space ( thread_gno-1 ( destNo ) )  $\neq$  kRootServerSpace  $\wedge$ 
            thread_space ( thread_gno-1 ( destNo ) )  $\neq$  kKernelSpace
        THEN
            WeakDeleteThread ( itcb , destNo )
        ELSE
            Cannot delete privileged thread or self

            SetError ( itcb , eNoPrivilege )
        END
    ELSE
        SetError ( itcb , eUnavailableThread )
    END
WHEN   spaceSpec  $\neq$  kNilGNo  $\wedge$  destNo  $\notin$  ran ( thread_gno ) THEN

```

Thread Creation

```
SELECT   spaceSpec = kAnyGNo  THEN
    SetError ( itcb , eInvalidSpace )
WHEN   pagerNo = kAnyGNo  THEN
    SetError ( itcb , eUnavailableThread )
WHEN   schedNo = kNilGNo  THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   schedNo = kAnyGNo  THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   pagerNo ≠ kNilGNo ∧ pagerNo ∉ ran ( thread_gno )  THEN
    SetError ( itcb , eUnavailableThread )
WHEN   schedNo ≠ kNilGNo ∧ schedNo ∉ ran ( thread_gno )  THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   spaceSpec = destNo ∧ spaces = ADDRESS_SPACE  THEN
    SetError ( itcb , eOutOfMemory )
WHEN   spaceSpec ≠ destNo ∧ spaceSpec ∉ ran ( thread_gno )  THEN
    SetError ( itcb , eInvalidSpace )
WHEN   pagerNo ∈ ran ( thread_gno ) ∧
    thread_space ( thread_gno-1 ( spaceSpec ) ) ∉ initialised_spaces
THEN
    SetError ( itcb , eInvalidSpace )
WHEN   pagerNo ∈ ran ( thread_gno ) ∧
    thread_gno-1 ( schedNo ) ∉ active_threads
THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   spaceSpec ∈ ran ( thread_gno ) ∧
    threads_in_space ( thread_space ( thread_gno-1 ( spaceSpec ) ) )
    = kMaxThreadsPerSpace
THEN
    SetError ( itcb , eOutOfMemory )
ELSE
    WeakCreateThread ( itcb , destNo , spaceSpec , schedNo , pagerNo )
END
WHEN   spaceSpec ≠ kNilGNo ∧ destNo ∈ ran ( thread_gno ) ∧
    thread_gno-1 ( destNo ) ∉ kIntThreads  THEN
```

Thread Modification - normal.

```
SELECT   spaceSpec = kAnyGNo  THEN
    SetError ( itcb , eInvalidSpace )
WHEN   schedNo = kAnyGNo  THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   pagerNo = kAnyGNo  THEN
    SetError ( itcb , eUnavailableThread )
WHEN   spaceSpec ≠ kNilGNo ∧ spaceSpec ∉ ran ( thread_gno )  THEN
    SetError ( itcb , eInvalidSpace )
```

```

WHEN   schedNo ≠ kNilGNo ∧ schedNo ∉ ran ( thread_gno ) THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   pagerNo ≠ kNilGNo ∧ pagerNo ∉ ran ( thread_gno ) THEN
    SetError ( itcb , eUnavailableThread )
WHEN   thread_gno-1 ( destNo ) ∉ active_threads ∧ pagerNo ≠ kNilGNo ∧
    ¬ ( schedNo ≠ kNilGNo ⇒
    thread_gno-1 ( schedNo ) ∈ active_threads )
THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   thread_gno-1 ( destNo ) ∉ active_threads ∧ pagerNo ≠ kNilGNo ∧
    ¬ ( schedNo = kNilGNo ⇒
    thread_scheduler ( thread_gno-1 ( destNo ) ) ∈ active_threads )
THEN
    SetError ( itcb , eInvalidScheduler )
WHEN   thread_space ( thread_gno-1 ( spaceSpec ) ) ≠ thread_space ( thread_gno-1 ( destNo ) ) ∧
    ¬ ( threads_in_space ( thread_space ( thread_gno-1 ( spaceSpec ) ) ) < kMaxThreadsPerSpace )
THEN
    SetError ( itcb , eOutOfMemory )
ELSE
    WeakModifyThread ( itcb , destNo , spaceSpec , schedNo , pagerNo )
END
WHEN   spaceSpec ≠ kNilGNo ∧ destNo ∈ ran ( thread_gno ) ∧
    thread_gno-1 ( destNo ) ∈ kIntThreads
THEN
    Thread Modification - interrupt.
IF   pagerNo ∉ ran ( thread_gno ) THEN
    SetError ( itcb , eUnavailableThread )
ELSE
    IntThreadControl ( itcb , destNo , pagerNo )
END
END
END ;

```

The redirector parameter is ignored, because it doesn't do anything in the source code, has only a partial description in the reference manual, and remains a hotly contested topic.

```

SpaceControl ( itcb , spaceSpec , control , KernelInterfacePageArea , UtcbaArea ) ≐
PRE   itcb ∈ active_threads ∧ thread_state ( itcb ) = tsRunning ∧
    spaceSpec ∈ TCB ∧ KernelInterfacePageArea ∈ FPAGE ∧
    UtcbaArea ∈ FPAGE ∧ control ∈ ℕ
THEN
SELECT   ¬ ( dIsPrivilegedSpace ( thread_space ( itcb ) ) ) THEN

```

```

    SetError ( tcb , eNoPrivilege )
WHEN   spaceSpec  $\notin$  threads THEN
    SetError ( tcb , eInvalidSpace )
ELSE
    The exact details of memory management are not really crucial. Either they are satisfactory
    and the space is initialised, or it is not.
CHOICE
    InitialiseAddressSpace2 ( tcb , thread_space ( spaceSpec ) )
OR
    ANY   error WHERE   error  $\in$  { eInvalidUtcArea , eInvalidKipArea } THEN
        SetError ( tcb , error )
    END
END
END
END ;

ExchangeRegisters ( tcb , tcb , control , sp , ip , flags , pager , handle )  $\hat{=}$ 
PRE   tcb  $\in$  active_threads  $\wedge$  thread_state ( tcb ) = tsRunning  $\wedge$ 
    tcb  $\in$  TCB  $\wedge$  control  $\subseteq$  EXREGS_FLAGS  $\wedge$  sp  $\in$   $\mathbb{N}$   $\wedge$  ip  $\in$   $\mathbb{N}$   $\wedge$ 
    pager  $\in$  TCB  $\wedge$  flags  $\in$   $\mathbb{N}$   $\wedge$  handle  $\in$   $\mathbb{N}$ 
THEN

    SELECT   tcb  $\notin$  threads THEN
        SetError ( tcb , eInvalidThread )
    WHEN   tcb  $\in$  threads  $\wedge$  thread_space ( tcb )  $\neq$  thread_space ( tcb ) THEN
        SetError ( tcb , eInvalidThread )
    ELSE
        CHOICE
            IpcBaseExchangeRegisters ( tcb , control , pager )
        OR
            ANY   error WHERE   error  $\in$  { eOutOfMemory , eInvalidUtcLocation }
            THEN
                SetError ( tcb , error )
            END
        END
    END
END
END ;

```

Relating to the above, this specification does not deal with internal memory management issues. Also, it does not perform the internal copy-mechanics of IPC, meaning no granting and no mapping occurs. Therefore, unmapping is also a skip operation. This does not imply the model is flawed, merely that the functionality needs to be added in a refinement of IPC. Unmap can then be refined also.

```

Unmap ( tcb , control )  $\hat{=}$ 

```



```

PRE    $itcb \in active\_threads \wedge thread\_state ( itcb ) = tsRunning \wedge$ 
         $control \in \mathbb{N}$ 
THEN
    skip
END   ;

```

This system call donates the rest of this timeslice to another thread. If the thread is “any thread”, then the scheduler picks the target thread. If it is a specific thread number, then control gets transferred to that thread if it exists; if it does not, the invocation follows as if the target was “any thread”. Alas, given a CPU-number-independent abstraction of the system, it is not known which thread is presently executing, so at this level of refinement, nothing visible happens.

```

ThreadSwitch (  $itcb, dest$  )  $\hat{=}$ 
PRE    $itcb \in active\_threads \wedge thread\_state ( itcb ) = tsRunning \wedge$ 
         $dest \in TCB$ 
THEN
     $SetError ( itcb, eNoError )$ 
END   ;

```

Since purely non-deterministic scheduling is currently used, there is no need to store scheduling aspects of threads such as priority, since they will have no effect on the model. As a consequence of this, the Schedule operation must also be purely non-deterministic.

```

Schedule (  $itcb, destNo, timeControl, procControl, prio, preemptControl$  )  $\hat{=}$ 
PRE    $itcb \in active\_threads \wedge thread\_state ( itcb ) = tsRunning \wedge$ 
         $destNo \in GLOBAL\_TNO \wedge$ 
         $timeControl \in \mathbb{N} \wedge procControl \in \mathbb{N} \wedge prio \in \mathbb{N} \wedge$ 
         $preemptControl \in \mathbb{N}$ 
THEN

    IF    $destNo \notin ran ( thread\_gno )$  THEN
         $SetError ( itcb, eInvalidThread )$ 
    ELSIF  $thread\_scheduler ( thread\_gno^{-1} ( destNo ) ) \neq itcb$  THEN

        Must be the thread's scheduler.

         $SetError ( itcb, eNoPrivilege )$ 
    ELSE
        CHOICE
            skip
        OR

            One of the other parameters is invalid.

```

```

        SetError ( itcb , eInvalidParameter )
    END
END
END ;

```

Returns the value of the internal counter in μs . It does not, however, return this value to the kernel, but instead to the thread in some register not available in this specification.

```

SystemClock ( itcb )  $\hat{=}$ 
    PRE  itcb  $\in$  active_threads  $\wedge$  thread_state ( itcb ) = tsRunning  THEN
        SetError ( itcb , eNoError )
    END ;

```

Processor and memory control are only included for completeness, they work at the hardware level and don't do anything to the kernel itself.

```

ProcessorControl ( itcb , procNo , internalFreq , extFreq , voltage )  $\hat{=}$ 
    PRE  itcb  $\in$  active_threads  $\wedge$  thread_state ( itcb ) = tsRunning  $\wedge$ 
        procNo  $\in$   $\mathbb{N}$   $\wedge$  internalFreq  $\in$   $\mathbb{N}$   $\wedge$  extFreq  $\in$   $\mathbb{N}$   $\wedge$  voltage  $\in$   $\mathbb{N}$ 
    THEN
        SELECT   $\neg$  ( dIsPrivilegedSpace ( thread_space ( itcb ) ) )  THEN
            SetError ( itcb , eNoPrivilege )
        ELSE
            SetError ( itcb , eNoError )
        END
    END ;

```

```

MemoryControl ( itcb , attr0 , attr1 , attr2 , attr3 )  $\hat{=}$ 
    PRE  itcb  $\in$  active_threads  $\wedge$  thread_state ( itcb ) = tsRunning  $\wedge$ 
        attr0  $\in$   $\mathbb{N}$   $\wedge$  attr1  $\in$   $\mathbb{N}$   $\wedge$  attr2  $\in$   $\mathbb{N}$   $\wedge$  attr3  $\in$   $\mathbb{N}$ 
    THEN
        SELECT   $\neg$  ( dIsPrivilegedSpace ( thread_space ( itcb ) ) )  THEN
            SetError ( itcb , eNoPrivilege )
        ELSE
            SetError ( itcb , eNoError )
        END
    END
END

```

END

A.2 WeakSyscall

MACHINE *WeakSyscall*

SEES

KernelInformation , *ThreadIdCtx* , *Bool_TYPE* , *ThreadStateCtx* , *AddressSpaceCtx* ,
TimeoutCtx , *ThreadCtx* , *ErrorCtx*

INCLUDES *IpcBase*

PROMOTES

SetError ,
TimeoutWait ,
TimeoutPoll ,
ResolveIPC ,
InitialiseAddressSpace2 ,
IpcBaseExchangeRegisters

OPERATIONS

WeakIpc (*itcb* , *to_gno* , *fromSpecifier* , *send_timeout* , *recv_timeout*) $\hat{=}$

PRE *canIPC* (*itcb*) \wedge

to_gno \in *GLOBAL_TNO* \wedge

(*to_gno* \neq *kNilGNo* \Rightarrow

to_gno \in *ran* (*thread_gno*) \wedge

thread_gno⁻¹ (*to_gno*) \in *threads*) \wedge

fromSpecifier \in *GLOBAL_TNO* \wedge

fromSpecifier \in *thread_gno* [*threads*] \cup { *kAnyGNo* , *kNilGNo* } \wedge

send_timeout \in *TIMEOUT* \wedge

recv_timeout \in *TIMEOUT*

THEN

SELECT *to_gno* = *kNilGNo* \wedge *fromSpecifier* = *kNilGNo* **THEN**

SetError (*itcb* , *eSendNonExistingPartner*)

WHEN *to_gno* = *kNilGNo* \wedge *fromSpecifier* \neq *kNilGNo* **THEN**

IF *fromSpecifier* = *kAnyGNo* **THEN**

IF *thread_incoming* (*itcb*) \neq { } **THEN**

JustReceive (*itcb* , *fromSpecifier*)

ELSIF \neg (*isNoTimeout* (*recv_timeout*)) **THEN**

JustWait (*itcb* , *recv_timeout* , *fromSpecifier*)

ELSE

SetError (*itcb* , *eRecvTimeout*)

END

ELSE

```

IF fromSpecifier ∈ thread_incoming_gnos ( itcb ) THEN
  JustReceive ( itcb , fromSpecifier )
ELSIF ¬ ( isNoTimeout ( recv_timeout ) ) THEN
  JustWait ( itcb , recv_timeout , fromSpecifier )
ELSE
  SetError ( itcb , eRecvTimeout )
END
END
WHEN to_gno ≠ kNilGNo THEN
  ANY to WHERE to ∈ threads ∧ thread_gno-1 ( to_gno ) = to THEN

  IF to ∈ dom ( thread_ipc_waiting_for ) THEN
    IF thread_ipc_waiting_for ( to ) ∈ { thread_gno ( itcb ) , kAnyGNo } THEN

      IF fromSpecifier ≠ kNilGNo ⇒
        ¬ ( isNoTimeout ( recv_timeout ) ) THEN
          WakeDestThenWait ( itcb , to , recv_timeout ,
            fromSpecifier )
        ELSE
          SetError ( itcb , eSendTimeout )
        END
      ELSE

        IF fromSpecifier ≠ kNilGNo ⇒
          ¬ ( isNoTimeout ( recv_timeout ) ) THEN
            SetUpReceivePhaseAndPoll ( itcb , to , send_timeout ,
              recv_timeout , fromSpecifier )
          ELSE
            SetError ( itcb , eSendTimeout )
          END
        END
      END
    ELSE
      IF fromSpecifier ≠ kNilGNo ⇒
        ¬ ( isNoTimeout ( recv_timeout ) ) THEN
          SetUpReceivePhaseAndPoll ( itcb , to , send_timeout ,
            recv_timeout , fromSpecifier )
        ELSE
          SetError ( itcb , eSendTimeout )
        END
      END
    END
  END
END
END ;

```

Not a syscall, but constitutes the deletion part of ThreadControl.

WeakDeleteThread (*itcb* , *dest*) $\hat{=}$
PRE $itcb \in active_threads \wedge dIsPrivilegedSpace (thread_space (itcb)) \wedge$
 $dest \in ran (thread_gno) \wedge$
 $thread_space (thread_gno^{-1} (dest)) \neq kSigma0Space \wedge$
 $thread_space (thread_gno^{-1} (dest)) \neq kRootServerSpace \wedge$
 $thread_space (thread_gno^{-1} (dest)) \neq kKernelSpace$ **THEN**
 $DeleteThread2 (thread_gno^{-1} (dest))$
END ;

Not a syscall, but constitutes the modification part of ThreadControl

WeakModifyThread (*itcb* , *destNo* , *spaceSpecifier* , *schedNo* , *pagerNo*) $\hat{=}$
PRE $itcb \in active_threads \wedge dIsPrivilegedSpace (thread_space (itcb)) \wedge$
 $destNo \in GLOBAL_TNO \wedge destNo \in ran (thread_gno) \wedge$
 $spaceSpecifier \in GLOBAL_TNO \wedge$
 $schedNo \in GLOBAL_TNO \wedge$
 $pagerNo \in GLOBAL_TNO \wedge$
 $spaceSpecifier \neq kAnyGNo \wedge$
 $schedNo \neq kAnyGNo \wedge$
 $pagerNo \neq kAnyGNo \wedge$

If we are changing one of the values, it must be to a valid thread.

$(spaceSpecifier \neq kNilGNo \Rightarrow spaceSpecifier \in ran (thread_gno)) \wedge$
 $(schedNo \neq kNilGNo \Rightarrow schedNo \in ran (thread_gno)) \wedge$
 $(pagerNo \neq kNilGNo \Rightarrow pagerNo \in ran (thread_gno)) \wedge$

Can't activate with a non-running scheduler

$(thread_gno^{-1} (destNo) \notin active_threads \wedge pagerNo \neq kNilGNo \Rightarrow$
 $(schedNo \neq kNilGNo \Rightarrow thread_gno^{-1} (schedNo) \in active_threads) \wedge$
 $(schedNo = kNilGNo \Rightarrow$
 $thread_scheduler (thread_gno^{-1} (destNo)) \in active_threads)) \wedge$

Migration, if necessary, must succeed.

$(spaceSpecifier \neq kNilGNo \wedge$
 $thread_space (thread_gno^{-1} (spaceSpecifier)) \neq thread_space (thread_gno^{-1} (destNo)) \Rightarrow$
 $threads_in_space (thread_space (thread_gno^{-1} (spaceSpecifier))) < kMaxThreadsPerSpace)$
THEN

Due to being unable to call operations in underlying machines in parallel, even when they don't access the same variables, this must be done.

```

SELECT   spaceSpecifier = kNilGNo  $\wedge$  schedNo = kNilGNo  $\wedge$  pagerNo = kNilGNo
THEN
  skip
WHEN   pagerNo  $\neq$  kNilGNo THEN

  Activation.

  IF   spaceSpecifier = kNilGNo  $\wedge$  schedNo = kNilGNo THEN
    ActivateThread2 ( thread_gno-1 ( destNo ) ,
      thread_space ( thread_gno-1 ( destNo ) ) ,
      thread_gno-1 ( pagerNo ) ,
      thread_scheduler ( thread_gno-1 ( destNo ) ) )
  ELSIF spaceSpecifier = kNilGNo  $\wedge$  schedNo  $\neq$  kNilGNo THEN
    ActivateThread2 ( thread_gno-1 ( destNo ) ,
      thread_space ( thread_gno-1 ( destNo ) ) ,
      thread_gno-1 ( pagerNo ) ,
      thread_gno-1 ( schedNo ) )
  ELSIF spaceSpecifier  $\neq$  kNilGNo  $\wedge$  schedNo = kNilGNo THEN
    ActivateThread2 ( thread_gno-1 ( destNo ) ,
      thread_space ( thread_gno-1 ( spaceSpecifier ) ) ,
      thread_gno-1 ( pagerNo ) ,
      thread_scheduler ( thread_gno-1 ( destNo ) ) )
  ELSE
    ActivateThread2 ( thread_gno-1 ( destNo ) ,
      thread_space ( thread_gno-1 ( spaceSpecifier ) ) ,
      thread_gno-1 ( pagerNo ) ,
      thread_gno-1 ( schedNo ) )
  END
ELSE

  Modification, space migration.

  IF   spaceSpecifier = kNilGNo THEN
    SetScheduler2 ( itcb , thread_gno-1 ( destNo ) , thread_gno-1 ( schedNo ) )
  ELSIF schedNo = kNilGNo THEN
    Migrate2 ( itcb , thread_gno-1 ( destNo ) ,
      thread_space ( thread_gno-1 ( spaceSpecifier ) ) )
  ELSE
    MigrateAndSetScheduler2 ( itcb , thread_gno-1 ( destNo ) ,
      thread_space ( thread_gno-1 ( spaceSpecifier ) ) ,
      thread_gno-1 ( schedNo ) )
  END
END
END ;

```

ThreadControl for interrupt threads.

```

IntThreadControl ( itcb , destNo , handlerNo )  $\hat{=}$ 
  PRE   itcb  $\in$  active_threads  $\wedge$  dIsPrivilegedSpace ( thread_space ( itcb ) )  $\wedge$ 
         destNo  $\in$  GLOBAL_TNO  $\wedge$  handlerNo  $\in$  GLOBAL_TNO  $\wedge$ 
         destNo  $\in$  ran ( thread_gno )  $\wedge$ 
         handlerNo  $\in$  ran ( thread_gno )  $\wedge$ 
         thread_gno-1 ( destNo )  $\in$  kIntThreads
  THEN

```

Creation or deletion doesn't really make sense here. While the L4 source code defines creation the reference manual does not mention this. Furthermore, lazy creation of UTCBs for interrupt threads is probably yet another performance optimisation.

```

  ANY dest WHERE dest  $\in$  threads  $\wedge$  thread_gno ( dest ) = destNo THEN
    IF destNo = handlerNo THEN
      DeactivateInterrupt2 ( itcb , dest )
    ELSE
      ANY handler WHERE handler  $\in$  threads  $\wedge$ 
        thread_gno ( dest ) = handlerNo THEN
          ActivateInterrupt2 ( itcb , dest , handler )
        END
      END
    END
  END
END ;

```

Not a syscall, but constitutes the creation part of ThreadControl

```

WeakCreateThread ( itcb , destNo , spaceSpecifier , schedNo , pagerNo )  $\hat{=}$ 
  PRE   itcb  $\in$  active_threads  $\wedge$  dIsPrivilegedSpace ( thread_space ( itcb ) )  $\wedge$ 
         destNo  $\in$  GLOBAL_TNO  $\wedge$  destNo  $\notin$  ran ( thread_gno )  $\wedge$ 
         spaceSpecifier  $\in$  GLOBAL_TNO  $\wedge$ 
         schedNo  $\in$  GLOBAL_TNO  $\wedge$ 
         spaceSpecifier  $\neq$  kAnyGNo  $\wedge$ 
         spaceSpecifier  $\neq$  kNilGNo  $\wedge$ 
         schedNo  $\neq$  kAnyGNo  $\wedge$ 
         schedNo  $\neq$  kNilGNo  $\wedge$ 
         pagerNo  $\neq$  kAnyGNo  $\wedge$ 
         pagerNo  $\in$  ran ( thread_gno )  $\cup$  { kNilGNo }  $\wedge$ 
         schedNo  $\in$  ran ( thread_gno )  $\wedge$ 

```

Resources must be available for thread and possibly space creation.

```

threads  $\neq$  TCB  $\wedge$ 
( spaceSpecifier = destNo  $\Rightarrow$  spaces  $\neq$  ADDRESS_SPACE )  $\wedge$ 

```

To create a new address space with a thread in it, set $destNo = spaceSpecifier$. Otherwise $spaceSpecifier$ must be valid.

$(spaceSpecifier \neq destNo \Rightarrow spaceSpecifier \in \text{ran} (thread_gno)) \wedge$

Creating an active thread requires an initialised space and a running scheduler.

$(pagerNo \in \text{ran} (thread_gno) \Rightarrow$

$thread_space (thread_gno^{-1} (spaceSpecifier)) \in initialised_spaces \wedge$

$thread_gno^{-1} (schedNo) \in active_threads) \wedge$

Can't exceed $kMaxThreadsPerSpace$

$(spaceSpecifier \in \text{ran} (thread_gno) \Rightarrow$

$threads_in_space (thread_space (thread_gno^{-1} (spaceSpecifier))) < kMaxThreadsPerSpace)$

THEN

ANY tcb **WHERE** $tcb \in TCB - threads$ **THEN**

IF $pagerNo = kNilGNo$ **THEN**

Create inactive.

IF $spaceSpecifier = destNo$ **THEN**

New space requested. Pick one.

ANY $space$ **WHERE** $space \in ADDRESS_SPACE - spaces$ **THEN**

$CreateThread2 (itcb , tcb , destNo , space ,$

$thread_gno^{-1} (schedNo))$

END

ELSE

Use the space specified by $spaceSpecifier$.

$CreateThread2 (itcb , tcb , destNo ,$

$thread_space (thread_gno^{-1} (spaceSpecifier)) ,$

$thread_gno^{-1} (schedNo))$

END

ELSE

Create active.

IF $spaceSpecifier = destNo$ **THEN**

New space requested. Pick one.

ANY $space$ **WHERE** $space \in ADDRESS_SPACE - spaces$ **THEN**

$CreateActiveThread2 (tcb , destNo , space ,$

$thread_gno^{-1} (schedNo) , thread_gno^{-1} (pagerNo))$

END

ELSE

Use the space specified by $spaceSpecifier$.

$CreateActiveThread2 (tcb , destNo ,$

$thread_space (thread_gno^{-1} (spaceSpecifier)) ,$

$thread_gno^{-1} (schedNo) , thread_gno^{-1} (pagerNo))$

END

END

END

END

END

A.3 IpcBase

MACHINE *IpcBase*

Operations enabling IPC to occur.

SEES

KernelInformation , *ThreadIdCtx* , *Bool_TYPE* , *ThreadStateCtx* , *AddressSpaceCtx* ,
TimeoutCtx , *ThreadCtx* , *ErrorCtx*

INCLUDES *IpcCore*

VARIABLES

Thread number a thread is waiting for.

thread_ipc_waiting_for ,
thread_ipc_waiting_timeout ,
thread_ipc_polling_on ,
thread_ipc_polling_timeout ,

When the thread gets awakened from polling by the target, and there is a receive phase, these will have been set to what the values of *thread_ipc_waiting_timeout* and *thread_ipc_waiting_for* will be set to upon entering a waiting state on IPC send phase success.

thread_recv_waiting_for ,
thread_recv_waiting_timeout ,

the queue of threads polling on this one

thread_incoming ,

the queue of thread numbers polling on this one

thread_incoming_gnos ,

represents the Error TCR

thread_error

INVARIANT

Threads waiting for IPCs must be waiting for an IPC from someone. Not waiting, ie. waiting for NilThread precludes from membership.

$$\begin{aligned} & \text{thread_ipc_waiting_for} \in \text{active_threads} \leftrightarrow \text{GLOBAL_TNO} \wedge \\ & k\text{NilGNo} \notin \text{ran}(\text{thread_ipc_waiting_for}) \wedge \\ & \text{thread_ipc_waiting_timeout} \in \text{active_threads} \leftrightarrow \text{TIMEOUT} \wedge \\ & e\text{ZeroTimeout} \notin \text{ran}(\text{thread_ipc_waiting_timeout}) \wedge \\ & \text{dom}(\text{thread_ipc_waiting_timeout}) = \text{dom}(\text{thread_ipc_waiting_for}) \wedge \\ & \text{dom}(\text{thread_ipc_waiting_timeout}) = \text{thread_state}^{-1}[\{ts\text{WaitingTimeout}, \\ & \quad ts\text{WaitingForever}\}] \wedge \end{aligned}$$

Threads polling must poll on a thread.

$$\begin{aligned} & thread_ipc_polling_on \in active_threads \leftrightarrow threads \wedge \\ & thread_ipc_polling_timeout \in active_threads \leftrightarrow TIMEOUT \wedge \\ & dom (thread_ipc_polling_timeout) = thread_state^{-1} [\{ tsPolling \}] \wedge \\ & eZeroTimeout \notin ran (thread_ipc_polling_timeout) \wedge \\ & dom (thread_ipc_polling_on) \subseteq dom (thread_ipc_polling_timeout) \wedge \end{aligned}$$

Post-polling receive phase status.

$$\begin{aligned} & thread_recv_waiting_for \in active_threads \leftrightarrow GLOBAL_TNO \wedge \\ & thread_recv_waiting_timeout \in active_threads \leftrightarrow TIMEOUT \wedge \\ & dom (thread_recv_waiting_for) \subseteq dom (thread_ipc_polling_timeout) \wedge \\ & dom (thread_recv_waiting_for) = dom (thread_recv_waiting_timeout) \wedge \\ & kNilGNo \notin ran (thread_recv_waiting_for) \wedge \\ & eZeroTimeout \notin ran (thread_recv_waiting_timeout) \wedge \end{aligned}$$

Not all polling threads will advance to a receive phase.

$$dom (thread_recv_waiting_timeout) \subseteq thread_state^{-1} [\{ tsPolling \}] \wedge$$

Future and actual waiting must not interfere with each other.

$$dom (thread_recv_waiting_timeout) \cap dom (thread_ipc_waiting_timeout) = \{ \}$$

When a thread is deleted, it must be removed from the range of these as well as the domains.

$$\begin{aligned} & thread_incoming \in active_threads \rightarrow \mathbb{P} (threads) \wedge \\ & \forall tt . (tt \in active_threads \Rightarrow thread_incoming (tt) = \\ & thread_ipc_polling_on^{-1} [\{ tt \}]) \wedge \\ & thread_incoming_gnos \in active_threads \rightarrow \mathbb{P} (GLOBAL_TNO) \wedge \\ & \forall tt . (tt \in active_threads \Rightarrow thread_incoming_gnos (tt) = \\ & thread_gno [thread_incoming (tt)]) \wedge \end{aligned}$$

The error status of an inactive thread has no implications.

$$thread_error \in active_threads \rightarrow ERROR$$

ASSERTIONS

$$\begin{aligned} & thread_state [dom (thread_ipc_waiting_timeout)] \subseteq \{ tsWaitingForever , tsWaitingTimeout \} \wedge \\ & thread_state [dom (thread_ipc_waiting_for)] \subseteq \{ tsWaitingForever , tsWaitingTimeout \} \end{aligned}$$

INITIALISATION

$$\begin{aligned} & thread_ipc_waiting_for := \{ \} \parallel \\ & thread_ipc_waiting_timeout := \{ \} \parallel \\ & thread_ipc_polling_on := \{ \} \parallel \\ & thread_ipc_polling_timeout := \{ \} \parallel \\ & thread_recv_waiting_for := \{ \} \parallel \\ & thread_recv_waiting_timeout := \{ \} \parallel \\ & thread_incoming := \{ kSigma0 , kRootServer \} \cup kIntThreads \rightarrow \{ \{ \} \} \parallel \\ & thread_incoming_gnos := \{ kSigma0 , kRootServer \} \cup kIntThreads \rightarrow \{ \{ \} \} \parallel \end{aligned}$$

$thread_error := \{ kSigma0 \mapsto eNoError, kRootServer \mapsto eNoError \} \cup kIntThreads \times \{ eNoError \}$

OPERATIONS

Activate thread. Requires a pager. Adds IPC functionality to ActivateThread in Thread.mch. Preconditions are the same.

ActivateThread2 ($tcb, space, pager, scheduler$) $\hat{=}$

PRE $tcb \in threads \wedge tcb \notin active_threads \wedge$
 $tcb \neq pager \wedge$
 $thread_space (tcb) \in initialised_spaces \wedge$
 $pager \in active_threads \wedge$
 $scheduler \in active_threads \wedge$
 $space \in initialised_spaces \wedge$
 $(space \neq thread_space (tcb) \Rightarrow$
 $threads_in_space (space) < kMaxThreadsPerSpace)$

THEN

$ActivateThread (tcb, space, pager, scheduler) \parallel$
 $thread_ipc_waiting_timeout (tcb) := eInfiniteTimeout \parallel$
 $thread_ipc_waiting_for (tcb) := thread_gno (pager) \parallel$
 $thread_error (tcb) := eNoError \parallel$

There is a risk some threads might already be polling on this one.

$thread_incoming (tcb) := thread_ipc_polling_on^{-1} [\{ tcb \}] \parallel$
 $thread_incoming_gnos (tcb) := thread_gno [thread_ipc_polling_on^{-1} [\{ tcb \}]]$

END ;

Create active thread. Adds IPC functionality. Preconditions are the same.

CreateActiveThread2 ($tcb, global_tno, space, scheduler, pager$) $\hat{=}$

PRE $tcb \in TCB - threads \wedge$
 $global_tno \in GLOBAL_TNO \wedge$
 $global_tno \notin ran (thread_gno) \wedge$
 $global_tno \neq kNilGNo \wedge$
 $global_tno \neq kAnyGNo \wedge$
 $scheduler \in active_threads \wedge$
 $pager \in threads \wedge$
 $tcb \neq pager \wedge$
 $space \in initialised_spaces \wedge$
 $space \neq kKernelSpace \wedge$
 $(space \in spaces \Rightarrow threads_in_space (space) < kMaxThreadsPerSpace)$

THEN

$CreateActiveThread (tcb, global_tno, space, scheduler, pager) \parallel$
 $thread_ipc_waiting_timeout (tcb) := eInfiniteTimeout \parallel$
 $thread_ipc_waiting_for (tcb) := thread_gno (pager) \parallel$
 $thread_error (tcb) := eNoError \parallel$

There is a risk some threads might already be polling on this one.

```

thread_incoming ( tcb ) := thread_ipc_polling_on -1 [ { tcb } ] ||
thread_incoming_gnos ( tcb ) := thread_gno [ thread_ipc_polling_on -1 [ { tcb } ] ]
END ;

```

Extends DeleteThread with cleaning up all IPC status information.

```

DeleteThread2 ( tcb ) ≐
PRE   tcb ∈ threads ∧ thread_space ( tcb ) ≠ kSigma0Space ∧
        thread_space ( tcb ) ≠ kRootServerSpace ∧
        thread_space ( tcb ) ≠ kKernelSpace
THEN
DeleteThread ( tcb ) ||
thread_ipc_waiting_for := { tcb } ≪ thread_ipc_waiting_for ||
thread_ipc_waiting_timeout := { tcb } ≪ thread_ipc_waiting_timeout ||

```

Remove all mappings for both this thread and all threads polling on this one, but not the timeout.

```

thread_ipc_polling_on := { tcb } ≪ thread_ipc_polling_on ≻ { tcb } ||
thread_ipc_polling_timeout := { tcb } ≪ thread_ipc_polling_timeout ||
thread_recv_waiting_for := { tcb } ≪ thread_recv_waiting_for ||
thread_recv_waiting_timeout := { tcb } ≪ thread_recv_waiting_timeout ||
thread_incoming :=
{ aa , bb | aa ∈ dom ( thread_incoming ) - { tcb } ∧
  bb ∈ ℙ ( TCB ) ∧
  bb = thread_incoming ( aa ) - { tcb } } ||
thread_incoming_gnos :=
{ aa , bb | aa ∈ dom ( thread_incoming_gnos ) - { tcb } ∧
  bb ∈ ℙ ( GLOBAL_TNO ) ∧
  bb = thread_incoming_gnos ( aa ) - { thread_gno ( tcb ) } } ||
thread_error := { tcb } ≪ thread_error
END ;

```

If no one matching fromSpecifier is polling on the thread, let it assume a waiting state. Timing out occurs via a separate operation.

```

JustWait ( tcb , timeout , fromSpecifier ) ≐
PRE   canIPC ( tcb ) ∧

```

Zero timeout is pointless.

```

timeout ∈ TIMEOUT ∧ ¬ ( isNoTimeout ( timeout ) ) ∧

```

Waiting for no one is also pointless.

$$\begin{aligned} & \text{fromSpecifier} \in \text{GLOBAL_TNO} \wedge \\ & \text{fromSpecifier} \neq \text{kNilGNo} \wedge \end{aligned}$$

If not waiting for just anyone, must be waiting for an existing thread.

$$\text{fromSpecifier} \in \text{thread_gno} [\text{threads}] \cup \{ \text{kAnyGNo} \} \wedge$$

No one matching fromSpecifier must be polling.

$$\begin{aligned} & (\text{fromSpecifier} = \text{kAnyGNo} \Rightarrow \text{thread_incoming} (\text{tcb}) = \{ \}) \wedge \\ & (\text{fromSpecifier} \neq \text{kAnyGNo} \Rightarrow \\ & \text{fromSpecifier} \notin \text{thread_incoming_gnos} (\text{tcb})) \end{aligned}$$

THEN

$$\begin{aligned} & \text{thread_ipc_waiting_for} (\text{tcb}) := \text{fromSpecifier} \parallel \\ & \text{thread_ipc_waiting_timeout} (\text{tcb}) := \text{timeout} \parallel \end{aligned}$$

IF *isInfinite* (*timeout*) **THEN**

$$\text{SetState} (\text{tcb} , \text{tsWaitingForever})$$

ELSE

$$\text{SetState} (\text{tcb} , \text{tsWaitingTimeout})$$

END

END ;

Target is not waiting for sender, and so must poll. If a receive phase is included, set it up. Timing out occurs via a different operation.

SetUpReceivePhaseAndPoll (*tcb_from* , *tcb_to* , *poll_timeout* , *recv_timeout* , *fromSpecifier*) $\hat{=}$

PRE *canIPC* (*tcb_from*) \wedge *tcb_to* \in *threads* \wedge

Target must not be waiting for source.

$$\begin{aligned} & (\text{tcb_to} \in \text{dom} (\text{thread_ipc_waiting_for}) \Rightarrow \\ & \text{thread_ipc_waiting_for} (\text{tcb_to}) \neq \text{thread_gno} (\text{tcb_from}) \wedge \\ & \text{thread_ipc_waiting_for} (\text{tcb_to}) \neq \text{kAnyGNo}) \wedge \end{aligned}$$

Waiting for no one is permitted (no recv phase).

$$\text{fromSpecifier} \in \text{GLOBAL_TNO} \wedge$$

Zero timeouts are pointless in this case.

$$\text{poll_timeout} \in \text{TIMEOUT} \wedge \neg (\text{isNoTimeout} (\text{poll_timeout})) \wedge$$

Unless there is no receive phase, whereby we don't care.

$recv_timeout \in TIMEOUT \wedge$
 $(fromSpecifier \neq kNilGNo \Rightarrow \neg (isNoTimeout (recv_timeout))) \wedge$

If there is a receive phase, and not waiting for just anyone, must be waiting for an existing thread.

$fromSpecifier \in thread_gno [threads] \cup \{ kAnyGNo , kNilGNo \}$

THEN

Make the thread poll on the destination.

$thread_ipc_polling_on (tcb_from) := tcb_to \parallel$
 $thread_ipc_polling_timeout (tcb_from) := poll_timeout \parallel$
 $thread_incoming (tcb_to) := thread_incoming (tcb_to) \cup \{ tcb_from \} \parallel$
 $thread_incoming_gnos (tcb_to) := thread_incoming_gnos (tcb_to) \cup \{ thread_gno (tcb_from) \} \parallel$
 $SetState (tcb_from , tsPolling) \parallel$

If there is a receive phase, initialise it.

IF $fromSpecifier \neq kNilGNo$ **THEN**
 $thread_recv_waiting_for (tcb_from) := fromSpecifier \parallel$
 $thread_recv_waiting_timeout (tcb_from) := recv_timeout$

END

END ;

Someone is polling on this thread, and we wish to receive an ipc from one such poller. Timeouts and polling are not in effect as this is instantaneous message pickup.

JustReceive ($itcb$, $fromSpecifier$) $\hat{=}$

PRE $canIPC (itcb) \wedge$

A receive phase is mandatory; must have someone to receive from.

$fromSpecifier \in GLOBAL_TNO \wedge$
 $fromSpecifier \neq kNilGNo \wedge$

That someone must be polling.

$(fromSpecifier \neq kAnyGNo \Rightarrow$
 $fromSpecifier \in thread_incoming_gnos (itcb)) \wedge$
 $(fromSpecifier = kAnyGNo \Rightarrow thread_incoming (itcb) \neq \{ \})$

THEN

ANY tcb_from **WHERE** $tcb_from \in thread_incoming (itcb) \wedge$
 $(fromSpecifier \neq kAnyGNo \Rightarrow$
 $thread_gno (tcb_from) \in thread_incoming_gnos (itcb))$

THEN

```

thread_ipc_polling_on := { tcb_from }  $\Leftarrow$  thread_ipc_polling_on ||
thread_ipc_polling_timeout :=
{ tcb_from }  $\Leftarrow$  thread_ipc_polling_timeout ||
thread_incoming ( itcb ) := thread_incoming ( itcb ) - { tcb_from } ||
thread_incoming_gnos ( itcb ) := thread_incoming_gnos ( itcb ) - { thread_gno ( tcb_from ) } ||
thread_recv_waiting_timeout :=
{ tcb_from }  $\Leftarrow$  thread_recv_waiting_timeout ||
thread_recv_waiting_for :=
{ tcb_from }  $\Leftarrow$  thread_recv_waiting_for ||

```

CHOICE

IPC Succeeds

```

PerformIPC ( tcb_from , itcb ) ||
thread_error := thread_error  $\Leftarrow$  { itcb  $\mapsto$  eNoError , tcb_from  $\mapsto$  eNoError } ||

```

If there is a receive phase pending, initiate it.

```

IF   tcb_from  $\in$  dom ( thread_recv_waiting_for ) THEN
    thread_ipc_waiting_for ( tcb_from ) :=
    thread_recv_waiting_for ( tcb_from ) ||
    thread_ipc_waiting_timeout ( tcb_from ) :=
    thread_recv_waiting_timeout ( tcb_from ) ||
    IF   isInfinite ( thread_recv_waiting_timeout ( tcb_from ) ) THEN
        SetState ( tcb_from , tsWaitingForever )
    ELSE
        SetState ( tcb_from , tsWaitingTimeout )
    END
ELSE
    UnWait ( tcb_from )
END

```

OR

IPC Fails. Both threads resume running with the IPC error.

```

UnWait ( tcb_from ) ||
ANY   error WHERE   error  $\in$  dIpFailures THEN
    thread_error := thread_error  $\Leftarrow$  { itcb  $\mapsto$  error , tcb_from  $\mapsto$  error }
END

```

END

END

END ;

Destination is waiting for source (or anyone). Wake up Destination and wait (if receive phase).

WakeDestThenWait (*tcb_from* , *tcb_to* , *recv_timeout* , *fromSpecifier*) $\hat{=}$

PRE *canIPC* (*tcb_from*) \wedge *tcb_to* \in *threads* \wedge
isWaiting (*thread_state* (*tcb_to*)) \wedge

Target must be waiting for source.

tcb_to \in *dom* (*thread_ipc_waiting_for*) \wedge
thread_ipc_waiting_for (*tcb_to*) \in { *thread_gno* (*tcb_from*) , *kAnyGNo* } \wedge

Waiting for no one is permitted (no recv phase).

fromSpecifier \in *GLOBAL_TNO* \wedge

Poll time out is irrelevant this case.

Unless there is no receive phase, a non-zero timeout must be available.

recv_timeout \in *TIMEOUT* \wedge
(*fromSpecifier* \neq *kNilGNo* \Rightarrow \neg (*isNoTimeout* (*recv_timeout*))) \wedge

If receive phase is included, and not waiting for just anyone, must be waiting for an existing thread.

fromSpecifier \in *thread_gno* [*threads*] \cup { *kAnyGNo* , *kNilGNo* }

THEN

IPC might still fail.

CHOICE

PerformIPC (*tcb_from* , *tcb_to*) \parallel
thread_error := *thread_error* \Leftarrow { *tcb_to* \mapsto *eNoError* , *tcb_from* \mapsto *eNoError* } \parallel

IF *fromSpecifier* = *kNilGNo* **THEN**

No receive phase. Wake up the target

thread_ipc_waiting_for := { *tcb_to* } \Leftarrow *thread_ipc_waiting_for* \parallel
thread_ipc_waiting_timeout :=
{ *tcb_to* } \Leftarrow *thread_ipc_waiting_timeout* \parallel
UnWait (*tcb_to*)

ELSE

Receive phase. Wake up target and wait.

thread_ipc_waiting_for :=
{ *tcb_to* } \Leftarrow *thread_ipc_waiting_for* \cup { *tcb_from* \mapsto *fromSpecifier* } \parallel

IF *isInfinite* (*recv_timeout*) **THEN**

thread_ipc_waiting_timeout :=
{ *tcb_to* } \Leftarrow *thread_ipc_waiting_timeout* \cup { *tcb_from* \mapsto *eInfiniteTimeout* } \parallel
WakeUpAndWait (*tcb_from* , *tcb_to* , *tsWaitingForever*)

ELSE

thread_ipc_waiting_timeout :=
{ *tcb_to* } \Leftarrow *thread_ipc_waiting_timeout* \cup { *tcb_from* \mapsto *eInfiniteTimeout* } \parallel
WakeUpAndWait (*tcb_from* , *tcb_to* , *tsWaitingTimeout*)

END

END

OR

Ipс Fails. No receive phase follows.

```

UnWait ( tcb_to ) ||
thread_ipc_waiting_for := { tcb_to } < thread_ipc_waiting_for ||
thread_ipc_waiting_timeout :=
{ tcb_to } < thread_ipc_waiting_timeout ||
ANY error WHERE error ∈ dIpсFailures THEN
    thread_error := thread_error < { tcb_to ↦ error , tcb_from ↦ error }
END
END
END ;

```

Waiting thread is waiting for the polling thread. This situation is resolved by the scheduler, hence the separate operation.

```

ResolveIPC ( tcb_from , tcb_to ) ≐
PRE   tcb_from ∈ active_threads ∧ tcb_to ∈ active_threads ∧
    isPolling ( thread_state ( tcb_from ) ) ∧
    isWaiting ( thread_state ( tcb_to ) ) ∧
    thread_ipc_polling_on ( tcb_from ) = tcb_to ∧
    ( thread_ipc_waiting_for ( tcb_to ) ≠ kAnyGNo ⇒
    thread_ipc_waiting_for ( tcb_to ) = thread_gno ( tcb_from ) )
THEN
    DualWakeUp ( tcb_from , tcb_to ) ||
    thread_ipc_waiting_for := { tcb_to } < thread_ipc_waiting_for ||
    thread_ipc_waiting_timeout := { tcb_to } < thread_ipc_waiting_timeout ||
    thread_ipc_polling_on := { tcb_from } < thread_ipc_polling_on ||
    thread_ipc_polling_timeout :=
    { tcb_from } < thread_ipc_polling_timeout ||
    thread_incoming ( tcb_to ) := thread_incoming ( tcb_to ) - { tcb_from } ||
    thread_incoming_gnos ( tcb_to ) := thread_incoming_gnos ( tcb_to ) - { thread_gno ( tcb_from ) } ||
CHOICE
    Ipс Succeeds.
    PerformIPC ( tcb_from , tcb_to ) ||
    thread_error := thread_error < { tcb_to ↦ eNoError , tcb_from ↦ eNoError }
OR
    Ipс Fails
    ANY error WHERE error ∈ dIpсFailures THEN
        thread_error := thread_error < { tcb_to ↦ error , tcb_from ↦ error }
    END
END

```

END ;

Invoked by system. Time out a polling thread.

```
TimeoutPoll  $\hat{=}$   
BEGIN  
  IF thread_ipc_polling_timeout  $\triangleright$  { eFiniteTimeout } = {} THEN  
    skip  
  ELSE  
    ANY tcb  
    WHERE  $tcb \in \text{dom} ( \text{thread\_ipc\_polling\_timeout} \triangleright \{ eFiniteTimeout \} )$  THEN  
      UnWait ( tcb ) ||  
      thread_ipc_polling_on := { tcb }  $\triangleleft$  thread_ipc_polling_on ||  
      thread_ipc_polling_timeout :=  
      { tcb }  $\triangleleft$  thread_ipc_polling_timeout ||  
      thread_incoming ( thread_ipc_polling_on ( tcb ) ) :=  
      thread_incoming ( thread_ipc_polling_on ( tcb ) ) - { tcb } ||  
      thread_incoming_gnos ( thread_ipc_polling_on ( tcb ) ) :=  
      thread_incoming_gnos ( thread_ipc_polling_on ( tcb ) ) - { thread_gno ( tcb ) } ||  
      thread_recv_waiting_timeout :=  
      { tcb }  $\triangleleft$  thread_recv_waiting_timeout ||  
      thread_recv_waiting_for :=  
      { tcb }  $\triangleleft$  thread_recv_waiting_for ||  
      thread_error ( tcb ) := eSendTimeout  
    END  
  END  
END ;
```

Invoked by system. Time out a waiting thread.

```
TimeoutWait  $\hat{=}$   
BEGIN  
  IF thread_ipc_waiting_timeout  $\triangleright$  { eFiniteTimeout } = {} THEN  
    skip  
  ELSE  
    ANY tcb  
    WHERE  $tcb \in \text{dom} ( \text{thread\_ipc\_waiting\_timeout} \triangleright \{ eFiniteTimeout \} )$  THEN  
      UnWait ( tcb ) ||  
      thread_ipc_waiting_for := { tcb }  $\triangleleft$  thread_ipc_waiting_for ||  
      thread_ipc_waiting_timeout :=  
      { tcb }  $\triangleleft$  thread_ipc_waiting_timeout ||  
      thread_error ( tcb ) := eRecvTimeout  
    END  
  END
```

```

    END
  END
END ;

SetError ( tcb , error )  $\hat{=}$ 
  PRE   tcb  $\in$  active_threads  $\wedge$  error  $\in$  ERROR THEN
    thread_error ( tcb ) := error
  END ;

```

The IPC extension to the ExchangeRegisters found in Thread

```

IpcBaseExchangeRegisters ( tcb , control , pager )  $\hat{=}$ 
  PRE   tcb  $\in$  threads  $\wedge$  control  $\subseteq$  EXREGS_FLAGS  $\wedge$  pager  $\in$  TCB  $\wedge$ 
    tcb  $\notin$  kIntThreads
  THEN

```

The power to abort or cancel IPCs. Note a thread can't be sending and receiving at the same time.

```

  IF   ex_S  $\in$  control  $\wedge$  tcb  $\in$  dom ( thread_ipc_polling_on ) THEN

```

Cancel sending

```

    ThreadExchangeRegisters ( tcb , control , pager , TRUE ) ||
    thread_ipc_polling_on := { tcb }  $\triangleleft$  thread_ipc_polling_on ||
    thread_ipc_polling_timeout :=
    { tcb }  $\triangleleft$  thread_ipc_polling_timeout ||
    thread_incoming ( thread_ipc_polling_on ( tcb ) ) :=
    thread_incoming ( thread_ipc_polling_on ( tcb ) ) - { tcb } ||
    thread_incoming_gnos ( thread_ipc_polling_on ( tcb ) ) :=
    thread_incoming_gnos ( thread_ipc_polling_on ( tcb ) ) - { thread_gno ( tcb ) } ||
    thread_recv_waiting_timeout :=
    { tcb }  $\triangleleft$  thread_recv_waiting_timeout ||
    thread_recv_waiting_for :=
    { tcb }  $\triangleleft$  thread_recv_waiting_for ||

```

```

    ANY   err WHERE   err  $\in$  { eSendCancelled , eAborted } THEN
      thread_error ( tcb ) := err

```

```

  END

```

```

  ELSIF   ex_R  $\in$  control  $\wedge$  tcb  $\in$  dom ( thread_ipc_waiting_for ) THEN

```

Cancel receiving

```

    ThreadExchangeRegisters ( tcb , control , pager , TRUE ) ||
    thread_ipc_waiting_for := { tcb }  $\triangleleft$  thread_ipc_waiting_for ||
    thread_ipc_waiting_timeout :=

```

```

{ tcb }  $\triangleleft$  thread_ipc_waiting_timeout ||

ANY err WHERE err  $\in$  { eRecvCancelled , eAborted } THEN
  thread_error ( tcb ) := err
END
ELSE
  ThreadExchangeRegisters ( tcb , control , pager , FALSE ) ||
  thread_error ( tcb ) := eNoError
END
END ;

```

Since invoking two operations from the same machine in B not permitted, and we want the error state to be cleared on success, these operations build upon those with similar names in the included machines, but also clear the error in the thread that invoked them.

```

InitialiseAddressSpace2 ( itcb , space )  $\hat{=}$ 
  PRE itcb  $\in$  active_threads  $\wedge$  space  $\in$  spaces THEN
    InitialiseAddressSpace ( space ) ||
    thread_error ( itcb ) := eNoError
  END ;

CreateThread2 ( itcb , tcb , global_tno , space , scheduler )  $\hat{=}$ 
  PRE itcb  $\in$  active_threads  $\wedge$  tcb  $\in$  TCB - threads  $\wedge$ 
    global_tno  $\in$  GLOBAL_TNO  $\wedge$ 
    global_tno  $\notin$  ran ( thread_gno )  $\wedge$ 
    global_tno  $\neq$  kNilGNo  $\wedge$ 
    global_tno  $\neq$  kAnyGNo  $\wedge$ 
    scheduler  $\in$  TCB  $\wedge$ 
    space  $\in$  ADDRESS_SPACE  $\wedge$ 
    space  $\neq$  kKernelSpace  $\wedge$ 
    ( space  $\in$  spaces  $\Rightarrow$  threads_in_space ( space )  $<$  kMaxThreadsPerSpace )
  THEN
    CreateThread ( tcb , global_tno , space , scheduler ) ||
    thread_error ( itcb ) := eNoError
  END ;

SetScheduler2 ( itcb , tcb , scheduler )  $\hat{=}$ 
  PRE itcb  $\in$  active_threads  $\wedge$  tcb  $\in$  threads  $\wedge$  scheduler  $\in$  threads THEN
    SetScheduler ( tcb , scheduler ) ||
    thread_error ( itcb ) := eNoError
  END ;

Migrate2 ( itcb , tcb , space )  $\hat{=}$ 
  PRE itcb  $\in$  active_threads  $\wedge$  tcb  $\in$  threads  $\wedge$  space  $\in$  spaces  $\wedge$ 
    ( space  $\neq$  thread_space ( tcb )  $\Rightarrow$ 
    threads_in_space ( space )  $<$  kMaxThreadsPerSpace )

```

```

THEN
  Migrate ( tcb , space ) ||
  thread_error ( itcb ) := eNoError
END ;

MigrateAndSetScheduler2 ( itcb , tcb , space , scheduler ) ≐
PRE   itcb ∈ active_threads ∧ tcb ∈ threads ∧
  space ∈ spaces ∧ scheduler ∈ threads ∧
  ( space ≠ thread_space ( tcb ) ⇒
  threads_in_space ( space ) < kMaxThreadsPerSpace )
THEN
  MigrateAndSetScheduler ( tcb , space , scheduler ) ||
  thread_error ( itcb ) := eNoError
END ;

ActivateInterrupt2 ( itcb , tcb , handler ) ≐
PRE   itcb ∈ active_threads ∧ tcb ∈ kIntThreads ∧
  handler ∈ TCB ∧ handler ≠ tcb THEN
  ActivateInterrupt ( tcb , handler ) ||
  thread_error ( itcb ) := eNoError
END ;

DeactivateInterrupt2 ( itcb , tcb ) ≐
PRE   itcb ∈ active_threads ∧ tcb ∈ kIntThreads THEN
  DeactivateInterrupt ( tcb ) ||
  thread_error ( itcb ) := eNoError
END
END

```

A.4 IpcCore

MACHINE *IpcCore*

Contains only the actual IPC transfer operation. This is very crude, but at the moment IPC doesn't really have anything to transfer yet (not until MRs are modelled).

SEES

KernelInformation , *ThreadIdCtx* , *Bool_TYPE* , *ThreadStateCtx* , *AddressSpaceCtx* ,
TimeoutCtx , *ThreadCtx*

EXTENDS *Thread*

OPERATIONS

PerformIPC (*from* , *to*) $\hat{=}$
PRE $from \in active_threads \wedge to \in active_threads \wedge$
 $canSend (from) \wedge canReceive (to)$ **THEN**
skip
END

DEFINITIONS

Participation in IPC is limited to active, non-halted threads which are not already participating in IPC. Note that "halted" means the exact opposite for interrupt threads.

$canIPC (t) \hat{=}$
 $t \in active_threads \wedge$
 $(t \in kIntThreads \Rightarrow t \in halted_threads) \wedge$
 $(t \notin kIntThreads \Rightarrow thread_state (t) = tsRunning \wedge t \notin halted_threads)$

END

A.5 Thread

MACHINE *Thread*

SEES

KernelInformation , *ThreadIdCtx* , *Bool_TYPE* , *ThreadStateCtx* , *AddressSpaceCtx* ,
TimeoutCtx , *ThreadCtx*

INCLUDES

AddressSpace

PROMOTES

InitialiseAddressSpace

VARIABLES

threads ,
thread_gno ,
active_threads ,
halted_threads ,
thread_space ,
thread_scheduler ,
thread_pager ,
thread_state ,
threads_in_space

INVARIANT

$threads \subseteq TCB \wedge$
 $thread_gno \in threads \mapsto GLOBAL_TNO \wedge$
 $kAnyGNo \notin \text{ran} (thread_gno) \wedge$
 $kNilGNo \notin \text{ran} (thread_gno) \wedge$
 $halted_threads \subseteq threads \wedge$
 $active_threads \subseteq threads \wedge$

Sigma0 and the Root Server are permanently active threads.

$kSigma0 \in active_threads \wedge$
 $kRootServer \in active_threads \wedge$

So are the interrupt threads.

$kIntThreads \subseteq active_threads \wedge$

No space may exist without a thread in it.

$thread_space \in threads \mapsto spaces \wedge$

A thread must not be active if its address space is uninitialised. However, the space must be initialised before the thread is activated.

$$thread_space [active_threads] \subseteq initialised_spaces \wedge$$

Sigma0 and RootServer get their own spaces, but can create more threads in their respective spaces.

$$thread_space (kSigma0) = kSigma0Space \wedge$$

$$thread_space (kRootServer) = kRootServerSpace \wedge$$

All (and only) interrupt threads reside in the kernel space.

$$thread_space [kIntThreads] = \{ kKernelSpace \} \wedge$$

$$thread_space^{-1} [\{ kKernelSpace \}] = kIntThreads \wedge$$

The only constraint placed on schedulers is that a thread must have a scheduler defined. Whether the scheduler is valid or not is only relevant at thread activation.

Interrupt threads do not have schedulers.

$$thread_scheduler \in threads - kIntThreads \rightarrow TCB \wedge$$

The scheduler of sigma0 is always the root server, which is always its own scheduler.

$$thread_scheduler (kSigma0) = kRootServer \wedge$$

$$thread_scheduler (kRootServer) = kRootServer \wedge$$

To become active, a thread must have a pager; should the pager get deleted, the thread will remain active.

$$thread_pager \in threads \leftrightarrow TCB \wedge$$

Sigma0 does not have a pager. The root server's pager is always Sigma0.

$$kSigma0 \notin \text{dom} (thread_pager) \wedge$$

Non-halted interrupt threads are their own pagers.

$$\forall kk . (kk \in kIntThreads \wedge kk \notin halted_threads \Rightarrow thread_pager (kk) = kk) \wedge$$

$$\forall kk . (kk \in kIntThreads \wedge kk \in halted_threads \Rightarrow thread_pager (kk) \neq kk) \wedge$$

Interrupt threads cannot be “running”.

$$thread_state \in threads \rightarrow THREAD_STATE \wedge$$

$$tsRunning \notin thread_state [kIntThreads] \wedge$$

Being active and aborted is exclusive to interrupt threads.

$$active_threads \cap thread_state^{-1} [\{ tsAborted \}] \subseteq kIntThreads \wedge$$

Only so many threads per address space possible.

$$threads_in_space \in spaces \rightarrow 0 .. kMaxThreadsPerSpace \wedge$$

$$\forall ss . (ss \in spaces \Rightarrow card (thread_space \triangleright \{ ss \}) = threads_in_space (ss))$$

ASSERTIONS

$$thread_scheduler [kIntThreads] = \{ \}$$

INITIALISATION

When the system loads, sigma0 and the root server are started.

$$threads := \{ kSigma0 , kRootServer \} \cup kIntThreads \parallel$$

$$active_threads := \{ kSigma0 , kRootServer \} \cup kIntThreads \parallel$$

$$thread_space := \{ kSigma0 \mapsto kSigma0Space ,$$

$$kRootServer \mapsto kRootServerSpace \} \cup kIntThreads \times \{ kKernelSpace \} \parallel$$

$$thread_gno := \{ kSigma0 , kRootServer \} \cup kIntThreads \mapsto GLOBAL_TNO - \{ kNilGNo , kAnyGNo \} \parallel$$

Interrupt threads start up disabled.

$$halted_threads := \{ \} \parallel$$

sigma0 does not have a pager, but is the pager for root server. Interrupt threads are their own pagers.

$$thread_pager := \{ kRootServer \mapsto kSigma0 \} \cup id (kIntThreads) \parallel$$

root server is scheduler for sigma0 and itself

$$thread_scheduler := \{ kSigma0 \mapsto kRootServer ,$$

$$kRootServer \mapsto kRootServer \} \parallel$$

sigma0 and the root server are initialised as running, while the interrupt threads as aborted.

$$thread_state := \{ kSigma0 \mapsto tsRunning , kRootServer \mapsto tsRunning \} \cup kIntThreads \times \{ tsAborted \} \parallel$$

Set up proper thread counters.

$$threads_in_space := \{ kSigma0Space \mapsto 1 , kRootServerSpace \mapsto 1 ,$$

$$kKernelSpace \mapsto card (kIntThreads) \}$$

OPERATIONS

Create a thread. Need to supply a free tcb, ids, and an address space for the thread to go into.

If address space is not one known to the system, create it.

In accordance with the spec, the scheduler is not checked until an attempt is made to activate the thread.

CreateThread (tcb , global_tno , space , scheduler) $\hat{=}$

PRE tcb \in TCB $-$ threads \wedge
 global_tno \in GLOBAL_TNO \wedge
 global_tno \notin ran (thread_gno) \wedge
 global_tno \neq kNilGNo \wedge
 global_tno \neq kAnyGNo \wedge
 scheduler \in TCB \wedge
 space \in ADDRESS_SPACE \wedge
 space \neq kKernelSpace \wedge
 (space \in spaces \Rightarrow threads_in_space (space) $<$ kMaxThreadsPerSpace)

THEN

If the space doesn't exist, the thread is it's first member.

```

IF   space  $\notin$  spaces   THEN
    CreateAddressSpace ( space )  $\parallel$ 
    threads_in_space ( space ) := 1
ELSE
    threads_in_space ( space ) := threads_in_space ( space ) + 1
END    $\parallel$ 
threads := threads  $\cup$  { tcb }  $\parallel$ 
thread_gno ( tcb ) := global_tno  $\parallel$ 
thread_space ( tcb ) := space  $\parallel$ 
thread_scheduler ( tcb ) := scheduler  $\parallel$ 

```

All threads are created inactive.
Until active, the thread is not ready for running.

```

    thread_state ( tcb ) := tsAborted
END   ;

```

Activate thread. Requires a pager. Higher-level operations should ensure that the waiting forever is for a message from this thread's pager. Migration during activation is possible.

ActivateThread (*tcb* , *space* , *pager* , *scheduler*) $\hat{=}$

```

PRE   tcb  $\in$  threads  $\wedge$  tcb  $\notin$  active_threads  $\wedge$ 
    pager  $\in$  threads  $\wedge$ 

```

Scheduler must exist and be running for activation.

```

scheduler  $\in$  active_threads  $\wedge$ 

```

If migration is necessary, target space must not be full.

```

space  $\in$  initialised_spaces  $\wedge$ 
( space  $\neq$  thread_space ( tcb )  $\Rightarrow$ 
threads_in_space ( space ) < kMaxThreadsPerSpace )

```

THEN

```

    thread_pager ( tcb ) := pager  $\parallel$ 
    thread_scheduler ( tcb ) := scheduler  $\parallel$ 
    active_threads := active_threads  $\cup$  { tcb }  $\parallel$ 

```

Threads will be waiting for wake-up IPC from their pager.

```

    thread_state ( tcb ) := tsWaitingForever  $\parallel$ 

```

Migrate if necessary.

```

IF  $space \neq thread\_space ( tcb )$  THEN
   $thread\_space ( tcb ) := space$  ||
   $threads\_in\_space := threads\_in\_space \Leftarrow \{ space \mapsto threads\_in\_space ( space ) + 1 ,$ 
     $thread\_space ( tcb ) \mapsto threads\_in\_space ( thread\_space ( tcb ) ) - 1 \}$ 
END
END ;

CreateActiveThread (  $tcb , global\_tno , space , scheduler , pager$  )  $\hat{=}$ 
PRE  $tcb \in TCB - threads \wedge$ 
   $global\_tno \in GLOBAL\_TNO \wedge$ 
   $global\_tno \notin ran ( thread\_gno ) \wedge$ 
   $global\_tno \neq kNilGNo \wedge$ 
   $global\_tno \neq kAnyGNo \wedge$ 
   $scheduler \in active\_threads \wedge$ 
   $pager \in threads \wedge$ 
   $space \in initialised\_spaces \wedge$ 
   $space \neq kKernelSpace \wedge$ 
   $( space \in spaces \Rightarrow threads\_in\_space ( space ) < kMaxThreadsPerSpace )$ 
THEN

  If the space doesn't exist, the thread is it's first member.
  IF  $space \notin spaces$  THEN
     $CreateAddressSpace ( space )$  ||
     $threads\_in\_space ( space ) := 1$ 
  ELSE
     $threads\_in\_space ( space ) := threads\_in\_space ( space ) + 1$ 
  END ||
   $threads := threads \cup \{ tcb \}$  ||
   $active\_threads := active\_threads \cup \{ tcb \}$  ||
   $thread\_gno ( tcb ) := global\_tno$  ||
   $thread\_space ( tcb ) := space$  ||
   $thread\_scheduler ( tcb ) := scheduler$  ||
   $thread\_pager ( tcb ) := pager$  ||

  Threads will be waiting for wake-up IPC from their pager.

   $thread\_state ( tcb ) := tsWaitingForever$ 
END ;

```

Deleting threads. Cannot delete a privileged thread. Higher level operations override this to delete other aspects of threads those machines define. If thread is the last one in an address space, delete the address space. No attempt to figure out whose pager or scheduler this thread was.

```

DeleteThread (  $tcb$  )  $\hat{=}$ 
PRE  $tcb \in threads \wedge thread\_space ( tcb ) \neq kSigma0Space \wedge$ 

```

```

thread_space ( tcb ) ≠ kRootServerSpace ∧
thread_space ( tcb ) ≠ kKernelSpace
THEN
  threads := threads - { tcb } ||
  active_threads := active_threads - { tcb } ||
  halted_threads := halted_threads - { tcb } ||
  thread_space := { tcb } ≪ thread_space ||
  thread_state := { tcb } ≪ thread_state ||
  thread_pager := { tcb } ≪ thread_pager ||
  thread_scheduler := { tcb } ≪ thread_scheduler ||
  thread_gno := { tcb } ≪ thread_gno ||
  IF { tcb } = thread_space-1 [ { thread_space ( tcb ) } ] THEN

    Last thread in this space. Address space must be removed too.

    DeleteAddressSpace ( thread_space ( tcb ) ) ||
    threads_in_space := { thread_space ( tcb ) } ≪ threads_in_space
  ELSE
    threads_in_space ( thread_space ( tcb ) ) :=
    threads_in_space ( thread_space ( tcb ) ) - 1
  END
END ;

```

The scheduler must actually be a valid thread, as suggested by the Reference Manual section 2.4 ThreadControl.

```

SetScheduler ( tcb , scheduler ) ≐
  PRE tcb ∈ threads ∧ scheduler ∈ threads THEN
    thread_scheduler ( tcb ) := scheduler
  END ;

Migrate ( tcb , space ) ≐
  PRE tcb ∈ threads ∧ space ∈ spaces ∧

    It must fit into the new space.

    ( space ≠ thread_space ( tcb ) ⇒
    threads_in_space ( space ) < kMaxThreadsPerSpace )
  THEN
    IF space ≠ thread_space ( tcb ) THEN
      thread_space ( tcb ) := space ||
      threads_in_space := threads_in_space ≪ { space ↦ threads_in_space ( space ) + 1 ,
      thread_space ( tcb ) ↦ threads_in_space ( thread_space ( tcb ) ) - 1 }
    END
  END ;

MigrateAndSetScheduler ( tcb , space , scheduler ) ≐
  PRE tcb ∈ threads ∧ space ∈ spaces ∧ scheduler ∈ threads ∧

```

It must fit into the new space.

```

( space  $\neq$  thread_space ( tcb )  $\Rightarrow$ 
  threads_in_space ( space ) < kMaxThreadsPerSpace )
THEN
  thread_scheduler ( tcb ) := scheduler ||
IF space  $\neq$  thread_space ( tcb ) THEN
  thread_space ( tcb ) := space ||
  threads_in_space := threads_in_space  $\Leftarrow$  { space  $\mapsto$  threads_in_space ( space ) + 1 ,
    thread_space ( tcb )  $\mapsto$  threads_in_space ( thread_space ( tcb ) ) - 1 }
END
END ;

```

Only allow transitions between Running and Waiting. Use other operations for transitioning from Aborted.

```

SetState ( tcb , state )  $\hat{=}$ 
PRE state  $\in$  THREAD_STATE  $\wedge$  state  $\neq$  tsAborted  $\wedge$  tcb  $\in$  active_threads  $\wedge$ 
  tcb  $\notin$  kIntThreads THEN
  thread_state ( tcb ) := state
END ;

```

Halting an interrupt thread means the thread is activated, and a pager (handler) should be supplied.

```

ActivateInterrupt ( tcb , handler )  $\hat{=}$ 
PRE tcb  $\in$  kIntThreads  $\wedge$  handler  $\in$  TCB  $\wedge$  handler  $\neq$  tcb THEN
  halted_threads := halted_threads  $\cup$  { tcb } ||
  thread_pager ( tcb ) := handler
END ;

```

```

DeactivateInterrupt ( tcb )  $\hat{=}$ 
PRE tcb  $\in$  kIntThreads THEN
  halted_threads := halted_threads - { tcb } ||
  thread_pager ( tcb ) := tcb
END ;

```

If a thread has been waiting or polling, and IPC wants to resume it, various things can happen.

```

UnWait ( tcb )  $\hat{=}$ 
PRE tcb  $\in$  threads THEN
SELECT tcb  $\in$  active_threads  $\wedge$  tcb  $\notin$  kIntThreads THEN
  thread_state ( tcb ) := tsRunning

```

```

WHEN    $tcb \in active\_threads \wedge tcb \in kIntThreads$   THEN
     $thread\_state ( tcb ) := tsAborted$ 
ELSE
     $thread\_state ( tcb ) := tsAborted$ 
END
END   ;

```

When sending to a thread waiting for you, and a receive phase is included, that thread starts running and you wait.

```

WakeUpAndWait (  $running\_tcb$  ,  $waiting\_tcb$  ,  $wait\_state$  )  $\hat{=}$ 
PRE    $running\_tcb \in active\_threads \wedge waiting\_tcb \in active\_threads \wedge$ 
     $isWaiting ( wait\_state ) \wedge$ 
     $isRunning ( thread\_state ( running\_tcb ) ) \wedge$ 
     $isWaiting ( thread\_state ( waiting\_tcb ) )$ 
THEN
    Interrupt threads go back to an aborted status, not a running one.
IF    $waiting\_tcb \in kIntThreads$   THEN
     $thread\_state := thread\_state \triangleleft \{ running\_tcb \mapsto wait\_state ,$ 
     $waiting\_tcb \mapsto tsAborted \}$ 
ELSE
     $thread\_state := thread\_state \triangleleft \{ running\_tcb \mapsto wait\_state ,$ 
     $waiting\_tcb \mapsto tsRunning \}$ 
END
END   ;

```

The thread portion of ExchangeRegisters

```

ThreadExchangeRegisters (  $tcb$  ,  $control$  ,  $pager$  ,  $unwait$  )  $\hat{=}$ 
PRE    $tcb \in threads \wedge control \subseteq EXREGS\_FLAGS \wedge pager \in TCB \wedge$ 
     $tcb \notin kIntThreads \wedge unwait \in BOOL$ 
THEN
IF    $ex\_p \in control$   THEN
     $thread\_pager ( tcb ) := pager$ 
END   ||
IF    $ex\_h \in control$   THEN
IF    $ex\_H \in control$   THEN
     $halted\_threads := halted\_threads - \{ tcb \}$ 
ELSE
     $halted\_threads := halted\_threads \cup \{ tcb \}$ 
END
END   ||

```

```

IF   unwait = TRUE  THEN
  IF   tcb ∈ active_threads  THEN
    thread_state ( tcb ) := tsRunning
  ELSE
    thread_state ( tcb ) := tsAborted
  END
END
END  ;

```

When the scheduler makes an IPC happen between a polling and receiving thread, they both revert to running.

```

DualWakeUp ( polling_tcb , waiting_tcb )  $\hat{=}$ 
PRE   polling_tcb ∈ active_threads  $\wedge$  waiting_tcb ∈ active_threads  $\wedge$ 
  isPolling ( thread_state ( polling_tcb ) )  $\wedge$ 
  isWaiting ( thread_state ( waiting_tcb ) )
THEN

```

Interrupt threads go back to an aborted status, not a running one.

```

SELECT  polling_tcb ∈ kIntThreads  $\wedge$  waiting_tcb ∈ kIntThreads  THEN
  thread_state := thread_state  $\Leftarrow$  { polling_tcb  $\mapsto$  tsAborted , waiting_tcb  $\mapsto$  tsAborted }
WHEN   polling_tcb ∈ kIntThreads  $\wedge$  waiting_tcb  $\notin$  kIntThreads  THEN
  thread_state := thread_state  $\Leftarrow$  { polling_tcb  $\mapsto$  tsAborted , waiting_tcb  $\mapsto$  tsRunning }
WHEN   polling_tcb  $\notin$  kIntThreads  $\wedge$  waiting_tcb ∈ kIntThreads  THEN
  thread_state := thread_state  $\Leftarrow$  { polling_tcb  $\mapsto$  tsRunning , waiting_tcb  $\mapsto$  tsAborted }
WHEN   polling_tcb  $\notin$  kIntThreads  $\wedge$  waiting_tcb  $\notin$  kIntThreads  THEN
  thread_state := thread_state  $\Leftarrow$  { polling_tcb  $\mapsto$  tsRunning , waiting_tcb  $\mapsto$  tsRunning }
END
END
END

```

A.6 AddressSpace

MACHINE *AddressSpace*

SEES

AddressSpaceCtx , *KernelInformation*

VARIABLES

spaces ,

Initialisation data.

initialised_spaces

INVARIANT

$spaces \subseteq ADDRESS_SPACE \wedge$
 $initialised_spaces \subseteq spaces$

INITIALISATION

Initialise privileged address spaces to their predefined constants.

$spaces := \{ kSigma0Space , kRootServerSpace , kKernelSpace \} \parallel$
 $initialised_spaces := \{ kSigma0Space , kRootServerSpace , kKernelSpace \}$

OPERATIONS

Create an address space. Requires an unused address space to be passed in.

CreateAddressSpace (*space*) $\hat{=}$
PRE $space \in ADDRESS_SPACE - spaces$ **THEN**
 $spaces := spaces \cup \{ space \}$
END ;

Initialise the space.

InitialiseAddressSpace (*space*) $\hat{=}$
PRE $space \in spaces$ **THEN**
 $initialised_spaces := initialised_spaces \cup \{ space \}$
END ;

Delete an address space. Cannot delete privileged spaces. They are integral to the system.


```
DeleteAddressSpace ( space )  $\hat{=}$   
  PRE   space  $\in$  spaces  $\wedge \neg$  ( dIsPrivilegedSpace ( space ) )  THEN  
    spaces := spaces - { space } ||  
    initialised_spaces := initialised_spaces - { space }  
  END  
END
```

A.7 KernelInformation

MACHINE *KernelInformation*

The system only supports a given number of threads. Additionally, each address space also has a limit. Since there are two initial threads (roottask, sigma0) in their own address spaces, plus some non-zero number of kernel threads, the absolute lower limit for both maxThreads is 3. Since no address space can exist without a thread in it, the absolute minimum for kMaxThreadsPerSpace is 1.

Three address spaces are initially constructed: kernel, roottask, sigma0, therefore the minimum for address spaces is 3.

CONSTANTS

kMaxThreads ,
kMaxThreadsPerSpace ,
kMaxAddressSpaces

PROPERTIES

$kMaxThreads \in \mathbb{N}_1 \wedge$
 $3 \leq kMaxThreads \wedge$
 $kMaxThreadsPerSpace \in \mathbb{N}_1 \wedge$
 $kMaxAddressSpaces \in \mathbb{N}_1 \wedge$
 $3 \leq kMaxAddressSpaces$

END

A.8 ThreadCtx

MACHINE *ThreadCtx*

SEES *KernelInformation*

SETS

TCB ;

An ExchangeRegisters call gets a subset of these which define what actions will be performed.

$EXREGS_FLAGS = \{ ex_h , ex_p , ex_u , ex_f , ex_i , ex_s , ex_S , ex_R , ex_H \}$

The choice of which TCBs get allocated to the privileged tasks is completely arbitrary, but NOT random, so they are constants defined by the implementor.

CONSTANTS

kSigma0 ,

kRootServer ,

kIntThreads

PROPERTIES

$\text{card} (TCB) = kMaxThreads \wedge$

$kSigma0 \in TCB \wedge$

$kRootServer \in TCB \wedge$

$\neg (kSigma0 = kRootServer) \wedge$

$kIntThreads \subset TCB \wedge$

$kIntThreads \neq \{ \} \wedge$

$\text{card} (kIntThreads) \leq kMaxThreadsPerSpace \wedge$

$kSigma0 \notin kIntThreads \wedge$

$kRootServer \notin kIntThreads$

DEFINITIONS

$\text{canSend} (t) \hat{=} \text{thread_state} (t) \in \{ tsRunning , tsPolling \} ;$

$\text{canReceive} (t) \hat{=} \text{thread_state} (t) \in \{ tsWaitingTimeout , tsWaitingForever \}$

END

A.9 ThreadIdCtx

MACHINE *ThreadIdCtx*

SEES

Bool_TYPE , *KernelInformation*

Since thread versions are an optimisation feature used for thread renaming and local thread ids an IPC optimisation, they are left out here.

SETS

Global thread identifiers.

GLOBAL_TNO

CONSTANTS

Global thread numbers representing no thread and any thread respectively.

kNilGNo ,

kAnyGNo

PROPERTIES

The maximum number of threads in the system does not take into account the extra thread numbers representing any and no thread.

$\text{card} (\text{GLOBAL_TNO}) = kMaxThreads + 2 \wedge$

$kNilGNo \in \text{GLOBAL_TNO} \wedge$

$kAnyGNo \in \text{GLOBAL_TNO} \wedge kAnyGNo \neq kNilGNo$

END

A.10 ThreadStateCtx

MACHINE *ThreadStateCtx*

Simplified.

SETS

$$\begin{aligned} \text{THREAD_STATE} = \{ & \text{tsRunning} , \\ & \text{tsWaitingForever} , \\ & \text{tsWaitingTimeout} , \\ & \text{tsPolling} , \\ & \text{tsAborted} \} \end{aligned}$$

DEFINITIONS

$$\begin{aligned} \text{isRunnable} (s) & \hat{=} s = \text{tsRunning} ; \\ \text{isSending} (s) & \hat{=} s = \{ \text{tsPolling} \} ; \\ \text{isReceiving} (s) & \hat{=} s \in \{ \text{tsWaitingForever} , \text{tsWaitingTimeout} \} ; \\ \text{isAborted} (s) & \hat{=} s = \text{tsAborted} ; \\ \text{isRunning} (s) & \hat{=} s = \text{tsRunning} ; \\ \text{isWaiting} (s) & \hat{=} s \in \{ \text{tsWaitingForever} , \text{tsWaitingTimeout} \} ; \\ \text{isWaitingForever} (s) & \hat{=} s = \text{tsWaitingForever} ; \\ \text{isWaitingWithTimeout} (s) & \hat{=} s = \text{tsWaitingTimeout} ; \\ \text{isPolling} (s) & \hat{=} s = \text{tsPolling} \end{aligned}$$

END

A.11 TimeoutCtx

MACHINE *TimeoutCtx*

SETS

$TIMEOUT = \{ eZeroTimeout, eFiniteTimeout, eInfiniteTimeout \}$

DEFINITIONS

$isFinite (t) \hat{=} t = eFiniteTimeout ;$
 $isInfinite (t) \hat{=} t = eInfiniteTimeout ;$
 $isNoTimeout (t) \hat{=} t = eZeroTimeout$

END

A.12 AddressSpaceCtx

MACHINE *AddressSpaceCtx*

SEES

KernelInformation

SETS

ADDRESS_SPACE

CONSTANTS

kSigma0Space ,

kRootServerSpace ,

kKernelSpace

L4 needs any 3 spaces reserved, but the implementer decides which ones to use (eg. the first 3 might be the most efficient), so they can be treated as constants defined during implementation.

PROPERTIES

$\text{card} (\text{ADDRESS_SPACE}) = kMaxAddressSpaces \wedge$

$kRootServerSpace \in \text{ADDRESS_SPACE} \wedge$

$kSigma0Space \in \text{ADDRESS_SPACE} \wedge$

$kKernelSpace \in \text{ADDRESS_SPACE} \wedge$

$kRootServerSpace \neq kSigma0Space \wedge$

$kSigma0Space \neq kKernelSpace \wedge$

$kRootServerSpace \neq kKernelSpace$

DEFINITIONS

$dIsPrivilegedSpace (s) \hat{=} s \in \{ kSigma0Space , kRootServerSpace , kKernelSpace \}$

END

A.13 FpageCtx

MACHINE *FpageCtx*

An Fpage is represented by the base, size, and set of permissions.

SETS

$$PERMS = \{ pfRead, pfWrite, pfExecute \}$$

DEFINITIONS

$$dFpage (b, s, p) \hat{=} b, s, p ;$$
$$dFpagePerms \hat{=} prj2 (\mathbb{N} \times \mathbb{N}, \mathbb{P} (PERMS)) ;$$
$$dFpageBase (f) \hat{=} prj1 (\mathbb{N}, \mathbb{N}) (prj1 (\mathbb{N} \times \mathbb{N}, \mathbb{P} (PERMS)) (f)) ;$$
$$dFpageSize (f) \hat{=} prj2 (\mathbb{N}, \mathbb{N}) (prj1 (\mathbb{N} \times \mathbb{N}, \mathbb{P} (PERMS)) (f)) ;$$
$$dIsFpage (f) \hat{=} f \in \mathbb{N} \times \mathbb{N} \times \mathbb{P} (PERMS) ;$$
$$FPAGE \hat{=} \mathbb{N} \times \mathbb{N} \times \mathbb{P} (PERMS)$$

END

A.14 ErrorCtx

MACHINE *ErrorCtx*

SETS

$ERROR = \{ eNoError ,$
 $eSendTimeout , eSendNonExistingPartner , eSendCancelled ,$
 $eRecvTimeout , eRecvNonExistingPartner , eRecvCancelled ,$
 $eMsgOverflow , eXferTimeoutSender , eXferTimeoutReceiver ,$
 $eAborted ,$
 $eNoPrivilege , eUnavailableThread , eInvalidSpace ,$
 $eInvalidScheduler , eOutOfMemory ,$
 $eInvalidThread , eInvalidUtcblLocation ,$
 $eInvalidUtcblArea , eInvalidKipArea ,$
 $eInvalidParameter \}$

DEFINITIONS

$dIpcFailures \hat{=} \{ eMsgOverflow , eXferTimeoutSender ,$
 $eXferTimeoutReceiver , eAborted \}$

END

A.15 Bool_TYPE

MACHINE *Bool_TYPE*

SETS *BOOL* = { *FALSE* , *TRUE* }

END