Formalising the L4 microkernel API

Rafal Kolanski **Gerwin Klein**

National ICT Australia Ltd. (NICTA)
Locked Bag 6016
The University of New South Wales
Sydney NSW 1466
Australia
Email: {rafal.kolanski|gerwin.klein}@nicta.com.au

Abstract

This paper gives an overview of a pilot project on the specification and verification of the L4 high-performance microkernel. Of the three aspects examined in the project, we describe one in more detail: the formalisation of the kernel's Application Programming Interface using the B Method. We conclude that machine-supported formal verification of software is at a turning point; that it is now feasible, and desirable, to formally verify production-quality operating systems.

Keywords: B Method, Operating System Specification, Software Verification

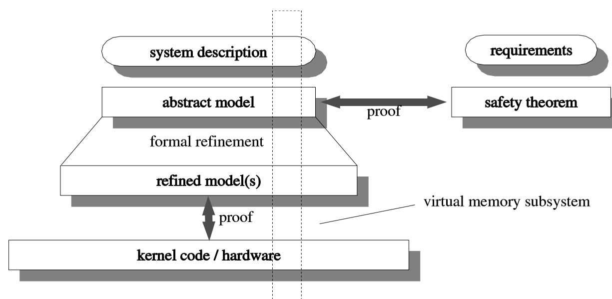

Figure 1: Overview

1 Introduction

The operating system (OS) kernel is defined to be the part of the OS that runs in the privileged mode of the hardware and thus is able to bypass hardware protection mechanisms. A microkernel is a kernel designed to be minimal in code size and concepts.

L4 is a second generation microkernel [14]. It provides the traditional advantages of the microkernel approach to system structure, namely improved reliability and flexibility, while overcoming the performance limitations of the previous generation of microkernels. With implementation sizes in the order of 10,000 lines of C++ and assembler code it is an order of magnitude smaller than Mach and two orders of magnitude smaller than Linux.

The correctness and reliability of any nontrivial system clearly critically depends on the operating system and its kernel. In terms of security, the OS is part of the trusted computing base, that is, the hardware and software necessary for the enforcement of a system's security policy. It has been repeatedly demonstrated that current operating systems fail at correctness, reliability, and security. Microkernels address the problem by applying the principles of minimality and least privilege to OS architecture. To gain confidence in the overall system, it is therefore highly desirable to formally verify the correctness of this design and its implementation.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

The L4 kernel is of a size which makes formalisation and verification feasible. Compared to other OS kernels, L4 is very small; compared to the size of other verification efforts, 10,000 lines of code is still considered a very large and complex system. Our methodology for solving this verification problem is shown in figure 1. It is a classic refinement strategy. We start out from an abstract model of the kernel that is phrased in terms of user concepts as they are explained in the L4 reference manual [13]. This is the level at which most of the safety and security theorems will be shown. We then formally refine this abstract model in multiple property preserving steps towards the implementation of L4. The last step consists of verifying that the C++ and assembler source code of the kernel correctly implements the most concrete refinement level. At the end of this process, we will have shown that the kernel source code satisfies the safety and security properties we have proved about the abstract model.

We conducted a pilot project to judge the feasibility of this verification task. The project investigated three main aspects: a formalisation of the kernel's Application Programming Interface (API) using the B Method (the first horizontal formalisation layer in figure 1), a full refinement proof for a non-trivial subsystem of the kernel using Isabelle/HOL (the vertical slice in figure 1), and a literature survey on formalising safety and security properties on the design level (the right-hand side of figure 1).

In this paper we give an overview of the first of these aspects: the API formalisation using the B Method, depicted as *abstract model* in figure 1. The L4 API provides three basic abstractions: threads, synchronous inter process communication (IPC), and virtual memory management (VMM). Our formalisation covers threads and IPC in detail and contains the basic structure for VMM. The latter has been formalised in depth in the vertical slice part of the project and is already described in earlier publications [20, 10]. Our formalisation is based on release version 0.3 of the L4Ka::Pistachio implementa-

tion [12].

We chose the B Method [1], because there existed a significant amount of experience with this approach among our student population and we wanted to compare at least two different formalisms before embarking on the full verification task. The B Method is a formal development methodology based on set theory with first-order logic. It allows progress from an initial high-level specification all the way to implementation via formal refinement. In this part of the project we have not done any formal refinement, but used the B Method and tool for formalisation only. The B Toolkit [2] allows for animation of the top-level specification which makes validating the specification more convenient. In this mode, the user becomes the implementation of all non-deterministic or undefined aspects.

After reviewing related work in section 2 and introducing B concepts and notation in section 3, we describe our formalisation of the L4 API in section 4. Section 5 gives pointers to further work and concludes.

2 Related Work

Earlier work on operating system kernel formalisation and verification includes PSOS [15] and UCLA Secure Unix [22]. The focus of this work was on capability-based security kernels, allowing security policies such as multi-level security to be enforced. These efforts were hampered by the lack of mechanisation and appropriate tools available at the time and so while the designs were formalised, the full verification proofs were not practical. Later work, such as KIT [3], describes verification of properties such as process isolation to source or object level but with kernels providing far simpler and less general abstractions than modern microkernels. There exists some work in the literature on the modelling of microkernels at the abstract level with varying degrees of completeness. Bevier and Smith [4] specify legal Mach states and describe Mach system calls using temporal logic. Shapiro and Weber [18] give an operational semantics for EROS and prove a confinement security policy. A number of case studies [6, 5, 21] in the literature describe the IPC and scheduling subsystems of microkernels in PROMELA and verify the formal descriptions with the SPIN model checker. These abstractions were not necessarily sound, having been manually constructed from the implementations, and so while useful for discovering concurrency bugs do not provide guarantees of correctness.

The VFiasco project, working with the Fiasco implementation of L4, has published exploratory work on the issues involved in C++ verification at the source level [9]. The VeriSoft project [8] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS that is inspired by, but not very close to L4. While the simplifications are appropriate for the goals of VeriSoft, it is doubtful that the VAMOS kernel will show the necessary performance to be relevant for industrial use.

Spivey uses Z, a predecessor formalism of B, to specify a simple kernel for a safety-critical X-ray diagnostic machine [19]. In abstracting the kernel from its implementation and documenting it for future reimplementations (possibly on different architectures), he finds a flaw in the system that could potentially have caused the X-ray machine to inflict damage.

The more academic approach of using formal design and specfication in the kernel development process up front and *then* proceeding with the implementation is utilised to good effect by Fowler and Wellings [7] for an Ada95 runtime support system in a hard real-time environment. From the verification perspective, this approach is more efficient than the post-hoc formalisation that is commonly found and which we are presenting here. The drawback is that while this process ensures a correct kernel, it is hard to get the runtime performance that separates practical microkernels from impractical ones.

In fact, we propose to do post-hoc formalisation of the existing L4 microkernel whose architecture has proven to deliver the required performance, but for the verification task itself we reserve the freedom to change details in the code base that make the verification process easier.

3 Notation

At the top specification level, the B Method uses *machines*, which represent finite state automata. *Refinements* further refine these, and *implementations* are the most concrete in the chain. Since our formalisation is entirely contained at the top level, we will only describe machine notation. A machine consists of the following sections:

DEFINITIONS They are purely syntactic translations. Any single-letter token counts as a so-called joker and can represent any set of tokens, similar to `#define` in C and C++. One definition cannot use another one within the same machine.

VARIABLES A comma separated list of variables.

SETS Enumerations and abstract sets.

CONSTANTS Declares constant sets, members of sets, or functions (which are also represented as sets).

PROPERTIES Restrictions on sets and constants.

INVARIANT The invariants of the machine, used to define variable types, properties, and relationships.

INITIALISATION Initial values for variables.

OPERATIONS The state transitions of the machine. 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. Operations can have preconditions.

The relationship between machines is restricted. A machine may *INCLUDE* (read-access to everything plus invoking operations), or *SEE* (read-access only to sets and constants) other machines. Write-access is only permitted through operations. If machine X includes Y, it can select which of B's operations are visible when another machine includes X with *PROMOTES*.

Since there is only one name space in B, we use naming conventions to avoid collisions. We use prefixes for all enumerations and a 'd' prefix for most definitions, as well as long classifying names such as *thread_ipc_waiting_timeout*.

Robinson provides a good reference to B syntax [17]. We define a few core notational concepts here and explain other non-standard notation as it occurs.

B supplies built-in sets such as the natural numbers, *NAT* and *NAT1* ($\mathbb{N} - \{0\}$). The library also

provides machines defining sets such as *INTEGER* and $BOOL = \{FALSE, TRUE\}$.

Two frequently used operations are the relational image and domain restriction. The relational image of set S under relation r is defined as:

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

If r is a function, this means all y such that $r(x) = y$. The pair (x, y) is denoted $x \mapsto y$.

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

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

The predefined functions dom and ran return the domain and range of relations and functions; card is the the cardinality of a set.

4 The Formalisation

This section describes the formalisation of the L4 microkernel API. As far as possible, we will introduce the kernel concepts together with their formal counterparts. The goals of the formalisation were the following.

Learning Animation of the model to improve and accelerate understanding of L4 internals for new users and developers.

Documentation L4 is continually developed and improved for efficiency. Boiling down the system to its essentials in the form of an abstract model will help to document and clarify the initial intentions and underlying basic mechanisms.

Experimentation The current version of the L4 API lacks an efficient communication restriction for information flow and also is vulnerable to denial of service attacks on kernel memory resources. One of the follow-up projects to this formalisation is revising the L4 API to fix these shortcomings. It is one of the goals of the formalisation presented here to serve as an experiment in kernel modelling to find out which methods work well.

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, kernel operations are system calls. They return values and they return them immediatly.

From the kernel's perspective, however, internal state changes are visible. For instance, the kernel might pick out the parameters from the thread's registers and memory, then pass them to an internal operation which implements the required functionality. 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. The system call also does not simply return a value internally, but instead copies return values into the thread's registers and memory.

To document the kernel behaviour in detail, we chose the second viewpoint. It allows, for example, modelling of thread state transitions in a natural way. Taking the inside view does not mean that we are exposing implementation details of the API — the formalisation remains at the conceptual level of a reference manual. The API for instance already implies that the kernel manages thread control states and the formalisation describes how these states are affected by operations. The formalisation does not describe which data structures are used to implement thread states in the kernel.

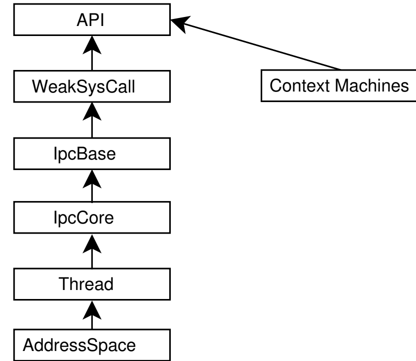

Figure 2: Inclusion diagram for the B development.

Figure 2 shows the module structure of the formalisation. Microkernels strive for the minimal set of functionality that is sufficient to build an OS. This means that the L4 kernel is effectively a single module in which everything is intertwined. We were able to separate out B modules for each of the major subsystems (address spaces, threads, IPC), but as figure 2 shows, they still depend on each other.

In the formalisation, we have placed all types and constants in separate context machines. There is one such context machine for each machine containing operations. In this presentation, we summarise these under the label ContextMachines and describe them in section 4.1. The rest of the presentation follows figure 2 bottom up. In section 4.2 we describe the address space stub (this subsystem has already been formalised separately), in section 4.3 the concept of threads and in section 4.4 the inter process communication subsystem. We leave out the description of *WeakSysCall* which collects these together into the L4 API functions, but does not contain any precondition checking yet. Section 4.5 describes the final interface available to the user. Due to space constraints, our description does not cover all of the formalisation at the same level of detail. The full formalisation is available elsewhere [11].

4.1 Context Machines

This section defines the basic types and constants used in the rest of the formalisation.

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.

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

We start off with the three main limiting aspects of the kernel: the number of threads in the system (*kMaxThreads*), the number of address spaces in the system (*kMaxAddressSpaces*), and the number of threads in an address space (*kMaxThreadsPerSpace*). These three constants have the following properties:

PROPERTIES

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

Each thread must have an address space; an address space can only be created by also creating a

thread [13, 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. Hence, the maxima must be at least 3, too.

In order to talk about address spaces within the model, we define the abstract set of all possible address spaces and restrict them to the maximum number of address spaces in the system:

SETS

ADDRESS_SPACE

PROPERTIES

card (*ADDRESS_SPACE*) =
kMaxAddressSpaces ∧
kRootServerSpace ∈ *ADDRESS_SPACE* ∧
kSigma0Space ∈ *ADDRESS_SPACE* ∧
kKernelSpace ∈ *ADDRESS_SPACE* ∧
kRootServerSpace ≠ *kSigma0Space* ∧
kSigma0Space ≠ *kKernelSpace* ∧
kRootServerSpace ≠ *kKernelSpace*

In the above, we define three new constants. Their function is to reserve three arbitrary members of *ADDRESS_SPACE* for the three core address spaces mentioned before: *kSigma0Space*, *kRootServerSpace*, and *kKernelSpace*.

These address spaces have special status in L4, they are privileged:

DEFINITIONS

dIsPrivilegedSpace (*s*) $\widehat{=}$
s ∈ { *kSigma0Space* , *kRootServerSpace* ,
kKernelSpace }

The following constants describe the control 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 3 presents an overview of the possible transitions between these states. We show a complete diagram below in figure 4, section 4.4.3.

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

- Multiprocessing-related states are missing since our model is too abstract to demonstrate effects of multiple-CPU interaction;
- The halted state is missing. According to discussions with the L4 developers, this state is better modelled by a flag. As defined in [13, section 2.3], halting a thread prevents it from executing in user mode, while ongoing IPC is not affected. This means 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 preserves functionality, but makes for a simpler model;

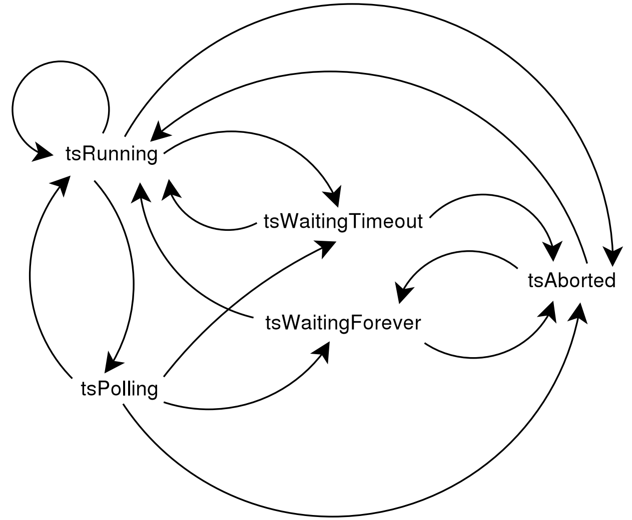

Figure 3: A simplified diagram of possible thread state transitions.

- The aborted state has a slightly different meaning than in the L4 implementation. In L4, all kernel thread control blocks are preallocated 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 our 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.

We now come to the IPC related constants:

DEFINITIONS

canSend (*t*) $\widehat{=}$
thread_state (*t*) ∈ { *tsRunning* , *tsPolling* } **;**
canReceive (*t*) $\widehat{=}$
thread_state (*t*) ∈ { *tsWaitingTimeout* ,
tsWaitingForever }

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 set TCB represents all possible threads creatable in the system. We have chosen this name due to its similarity to the pre allocated Kernel Thread Control Blocks in the system. The constants *kSigma0* and *kRootServer* reserve two distinct members of TCB for sigma0 and the root server threads, respectively.

Additionally, the constant *kIntThreads* reserves a subset of *TCB* for interrupt threads as follows:

kIntThreads ⊂ *TCB* ∧
kIntThreads ≠ {} ∧
card (*kIntThreads*) ≤ *kMaxThreadsPerSpace* ∧
kSigma0 ∉ *kIntThreads* ∧
kRootServer ∉ *kIntThreads*

The constant *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 [13, section 2.3]: *ex_h* represents *h*,

ex_R represents R and so on for all the flags: *hpufisSRH*. We explain the meaning of these flags below when we introduce the corresponding operations.

Now that the thread context is defined, we can model thread identifiers, the user's view of threads. We are leaving out local thread identifiers and thread versions, as they both are mainly a performance optimisation and do not extend the behaviour of the kernel. They can be added as a separate concept during refinement later. Thus, the set *GLOBAL_TNO* 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 $kMaxThreads + 2$. We omit the obvious definition in B.

The last set of constants concerns time-outs. Since the actual values of time-outs are irrelevant at this level of abstraction, we only define three predicates:

isNoTimeout requests an action be taken (or it will fail) immediately

isFiniteTimeout means that the thread will wait or poll for some time until timed out by the kernel, or cancelled by another thread

isInfiniteTimeout indicates that unless the operation is cancelled, the wait will go on indefinitely

We leave out the enumeration of error messages. It suffices to know that there is a set *ERROR* listing all of them. Also, *dIpcFailures* lists the failures during IPC that are beyond the deterministic control of the abstract model. If IPC fails non-deterministically, one of these will occur.

4.2 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 this model does not go into the details of memory management, it suffices to model which spaces are used by the system and which of those have been initialised. The model mainly consists of three operations *CreateAddressSpace*, *InitialiseAddressSpace*, and *DeleteAddressSpace*, which we describe below.

The *AddressSpaces* machine *SEES* the context machines described above, importing their abstract sets and constants and introduces two new variables:

spaces representing the address spaces that have been created, and

initialised_spaces representing the address spaces that are created and initialised

Their relationship is defined as follows:

INVARIANT

```
spaces ⊆ ADDRESS_SPACE ∧
initialised_spaces ⊆ spaces
```

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 implies being created.

Every variable in B must be initialised in a manner that establishes the invariant. In the context machines three address spaces were reserved: *kSigma0Space*, *kRootServerSpace*, and *kKernelSpace*. These are the spaces created and initialised by the root task on start up:

```
INITIALISATION
spaces := { kSigma0Space,
   kRootServerSpace, kKernelSpace } ||
initialised_spaces := { kSigma0Space,
   kRootServerSpace, kKernelSpace }
```

The operator $\|$ denotes parallel composition.

Next, we define the three operations that modify the state (variables) of this machine in the *OPERATIONS* clause. Since the operations are designed in such a way that satisfying their preconditions guarantees success, they do not return any values for error reporting.

The three operations are creating an address space, initialising it, and deleting it. Once an address space is initialised it cannot be uninitialised.

```
CreateAddressSpace ( space )  ≙
  PRE   space ∈ ADDRESS_SPACE − spaces
  THEN
    spaces := spaces ∪ { 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. Once the precondition is satisfied, the new identifier is added to the set of created address spaces. In the user visible L4 API there do no exist any real address space identifiers. Address spaces are referred to implicitly by the ID of any thread running in the address space. More than one thread ID can refer to the same address space. Internally, address spaces are identified by just pointers to address space structures which is what the space identifiers in this formalisation correspond to.

Initialisation is easy as well. If the space identifier is one of those already created, the operation succeeds and adds the identifier to the set of initialised address spaces:

```
InitialiseAddressSpace ( space )  ≙
  PRE   space ∈ spaces   THEN
    initialised_spaces := initialised_spaces ∪ { space }
  END
```

The final operation is deletion of an address space. To satisfy the invariant, it suffices that any member of *ADDRESS_SPACE* be passed in. For the operation to make sense, however, invoking it should only have meaning for an existing address space. Additionally, L4 does not allow deletion of privileged threads [12, SYS_THREAD_CONTROL in thread.cc], which means it does also not allow deletion of priviledged address spaces.

```
DeleteAddressSpace ( space )  ≙
  PRE   space ∈ spaces ∧
        ¬ ( dIsPrivilegedSpace ( space ) )
  THEN
    spaces := spaces − { space } ||
    initialised_spaces :=
       initialised_spaces − { space }
  END
```

As address space identifiers are not visible on the user level, address spaces are deleted implicitly when the last thread running in an address space is deleted. That means *DeleteAddressSpace* (as the other operations in this machine) are not visible at the API top level, but rather provide functionality for the rest of the formalisation.

4.3 Threads

Thread functionality is 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 IPC-related aspects of threads, such as which thread is waiting on another.

This section describes the *Thread* machine. Due to the kernel being extremely intertwined, the layered approach imposed by B caused a more complex structure than what would be expected from a greatly simplified specification.

We abstract away from the concept of processors and a currently running thread. One can look at it as a magical machine on which each thread has its own processor to execute on. That means, from a thread's perspective, suspension from execution is essentially transparent in our model.

The machine *INCLUDES* all the functionality in *AddressSpace*, and *PROMOTES* the *InitialiseAddressSpace* operation so that higher-level machines can call it whenever an address space needs to be created.

4.3.1 Variables

In the invariant, we proceed to define the meaning of the machine and its variables. We begin with threads and their subsets:

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

Where *threads* are the threads that have been created, *active_threads* are those that have been activated. The privileged threads are implicitly initialised when the kernel starts and cannot be uninitialised. Halted threads may not enter user mode. Interrupt threads have been assigned different halting semantics by the L4 kernel designers. See *thread_state* below.

All threads in L4 are uniquely identified by their thread numbers, with two arbitrarily reserved to represent any thread and no thread respectively:

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

We define a total injective function mapping threads to thread numbers, but excluding reserved numbers from its range. Note that in B, instead of a type declaration, we say that a function is a member of the set of all functions meeting given constraints.

Next we define the relationship between the existing threads and address spaces:

$$\begin{aligned}
&thread_space \in threads \twoheadrightarrow spaces \wedge \\
&thread_space\ (\ kSigma0\ ) = kSigma0Space \wedge \\
&thread_space\ (\ kRootServer\ ) = kRootServerSpace \wedge \\
&thread_space\ [\ active_threads\ ] \subseteq initialised_spaces \wedge \\
&thread_space\ [\ kIntThreads\ ] = \{\ kKernelSpace\ \} \wedge \\
&thread_space\ ^{-1}\ [\ \{\ kKernelSpace\ \}\ ] = kIntThreads
\end{aligned}$$

In L4, a created thread must have an address space. In fact, the address space pointer in the TCB is what defines whether a thread exists or not. Furthermore, there are no explicit address space identifiers; one specifies a thread in the address space instead. Hence, address spaces cannot be empty and *thread_space* is total as well as surjective (denoted $\twoheadrightarrow$). For a thread to be active it must reside in an initialised address space.

Interrupt threads are only an abstraction of the underlying hardware interrupts, and cannot actually run or have an implementation. The kernel space is therefore allocated to them.

Each thread has two other threads associated with it. These are the thread's scheduler and pager. The former is permitted to change the thread's scheduling-related properties, while the latter is invoked if the thread causes a page fault.

Let us look at the scheduler first:

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

Interrupt threads, naturally, cannot be scheduled. Since L4 does not keep track of schedulers, the range of *thread_scheduler* is not the set of existing threads, but can in fact be any TCB. The root server is traditionally the scheduler for sigma0 and itself. We believe that this situation should be maintained at all times, since otherwise the privileged threads could lose control of the system. The L4 source code does not, at this time, contain checks for this.

In L4, the process of page faults is resolved via IPC, i.e. a faulting thread needs a target to 'send' to (it is the kernel, however, which really performs the action on the thread's behalf). This target is known as the thread's *pager*:

$$\begin{aligned}
&thread_pager \in threads \nrightarrow TCB \wedge \\
&kSigma0 \notin \mathsf{dom}\ (\ thread_pager\ ) \wedge \\
&\forall\ kk\ .\ (\ kk \in kIntThreads \wedge kk \notin halted_threads \Rightarrow \\
&\qquad thread_pager\ (\ kk\ ) = kk\ ) \wedge \\
&\forall\ kk\ .\ (\ kk \in kIntThreads \wedge kk \in halted_threads \Rightarrow \\
&\qquad thread_pager\ (\ kk\ ) \neq kk\ )
\end{aligned}$$

The function is partial ($\nrightarrow$) since sigma0, being the initial system pager, does not have another pager to fall back on. Additionally, until the thread is activated, the pager field in its TCB is meaningless (in fact, setting a valid pager constitutes activation). Similar to *thread_scheduler* the range of *thread_pager* cannot be enforced, since the thread's pager may have been deleted and is not necessarily valid. Interrupts are enabled by setting the corresponding interrupt thread's *halted* flag and setting its pager to something other than itself. When disabled, the thread's pager must be the thread itself.

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

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

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 3 for a diagram of possible state transitions.

Finally, to make the specification simpler to read, we have a variable *threads_in_space* keeping a separate count of how many threads are in each address space. Its range is $0 \ldots kMaxThreadsPerSpace$.

4.3.2 Initialisation

We begin initialisation by creating sigma0, the root server and the interrupt threads. They are created active:

$$threads := \{ kSigma0 , kRootServer \} \cup kIntThreads \;\; \| \\ active_threads := \{ kSigma0 , kRootServer \} \\ \cup kIntThreads$$

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

$$thread_space := \{ kSigma0 \mapsto kSigma0Space , \\ kRootServer \mapsto kRootServerSpace \} \\ \cup kIntThreads \times \{ kKernelSpace \}$$

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

L4 initialises interrupt threads on first activation transparently to the user. Therefore it is not possible to tell whether an inactive interrupt thread has been initialised. In light of this, all interrupt threads in our model start as existing, but disabled:

$$halted_threads := \{\}$$

Since interrupt threads start out disabled, they are by definition (see section 4.3.1) their own pagers. Additionally, sigma0 is the root server's pager:

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

where id is the identity relation.

The root server starts up as the scheduler for sigma0 and for itself [12, thread.cc]:

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

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

$$thread_state := \{ kSigma0 \mapsto tsRunning , \\ kRootServer \mapsto tsRunning \} \cup \\ kIntThreads \times \{ tsAborted \}$$

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

$$threads_in_space := \{ kSigma0Space \mapsto 1 , \\ kRootServerSpace \mapsto 1 , \\ kKernelSpace \mapsto \mathsf{card} (kIntThreads) \}$$

4.3.3 Operations

As the *Thread* machine contains the core functionality in the formalisation, we will describe its operations in some detail. The machine corresponds to the ThreadControl subsystem of L4. The key operations at this level are *CreateThread*, *ActivateThread*, and *DeleteThread*.

To create a thread we need a free TCB, a thread number, an address space and a scheduler:

CreateThread (*tcb*, *global_tno*, *space*, *scheduler*)

CreateThread creates inactive threads. In order to succeed, the thread must not already exist, the supplied thread number must not be reserved or used for any existing thread. The address space the thread is to be created in, does not need to exist, but it cannot be the kernel space (which is reserved for interrupts):

PRE

$$tcb \in TCB - threads \\ global_tno \in GLOBAL_TNO \land \\ global_tno \notin \mathsf{ran} (thread_gno) \land \\ global_tno \neq kNilGNo \land \\ global_tno \neq kAnyGNo \land space \in ADDRESS_SPACE \land \\ space \neq kKernelSpace$$

Additionally, when no address space is supplied during thread creation, L4 creates a new address space for the new thread. At the level of the thread machine, we model this with the *CreateAddressSpace* operation if the space passed in does not exist. If it does exist, the total number of threads in it must not exceed the limit once this thread is added:

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

Since an inactive thread is being created, no restriction is placed on *scheduler*. If these conditions are satisfied, the operation is guaranteed to succeed. In order to actually create the thread, we either create a new address space with the thread in it, or add one to the address space, and simultaneously set all properties for the new thread:

IF $space \notin spaces$ **THEN**
$CreateAddressSpace (space) \;\; \|$
$threads_in_space (space) := 1$
ELSE
$threads_in_space (space) := threads_in_space (space) + 1$
END $\|$
$threads := threads \cup \{ tcb \} \;\; \|$
$thread_gno (tcb) := global_tno \;\; \|$
$thread_space (tcb) := space \;\; \|$
$thread_scheduler (tcb) := scheduler$

In the description of the following operations, trivial typing preconditions (such as tcb $\in$ TCB) will be omitted. This is an area where we would have found a type system like in Isabelle/HOL useful where type inference takes care of such trivial conditions automatically.

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, or as an *ActivateThread* operation on an inactive thread:

ActivateThread(*tcb*, *space*, *pager*, *scheduler*)

In order for the operation to succeed exactly when activation in L4 succeeds, *tcb* must be an existing but inactive thread and *pager* must exist and be running when the thread starts executing [13, section 2.4]:

PRE

$$pager \in threads \land \\ scheduler \in active_threads$$

L4 allows to migrate threads into new address spaces on activation. If this occurs, we must make sure that the thread fits into the new space:

$$space \in initialised_spaces \land \\ (space \neq thread_space (tcb) \Rightarrow \\ threads_in_space (space) < kMaxThreadsPerSpace)$$

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 the operation *ActivateThread2* in section 4.4.3 below. The migration is performed as follows:

$$
\begin{array}{l}
\textbf{IF} \quad space \neq thread_space\ (\ tcb\ ) \quad \textbf{THEN} \\
\quad thread_space\ (\ tcb\ ) := space \ \ \| \\
\quad threads_in_space := threads_in_space \lhd\!\!- \\
\quad\quad \{\ space \mapsto threads_in_space\ (\ space\ ) + 1\ , \\
\quad\quad\quad thread_space\ (\ tcb\ ) \mapsto \\
\quad\quad\quad threads_in_space\ (\ thread_space\ (\ tcb\ )\ ) - 1\ \} \\
\textbf{END}
\end{array}
$$

The thread counters for the two address spaces (current and target) are updated using *right overriding* (denoted $\lhd\!\!-$). Its definition is:

$$r_1 \lhd\!\!- \ r_2 = r_2 \cup (\ \mathrm{dom}(r_2) \lhd\!\!\!- \ r_1\ )$$

The definition uses another of B's operators, *domain subtraction* ($\lhd\!\!\!-$), defined as:

$$S \lhd\!\!\!- \ r = \{ x \mapsto y \mid x \mapsto y \in r \wedge x \notin S\ \}$$

We define *CreateActiveThread* as a merger of *CreateThread* and *ActivateThread*. The only difference is that migrating the thread is not possible as it does not exist yet. Note that a higher-level operation cannot combine the those two operations due to B's restrictions.

The operation *DeleteThread*, given an existing thread *tcb*, removes it from the set of known, active and halted threads provided the *tcb* is not in one of the privileged address spaces. We also remove it from all thread-related functions in the machine:

$$
\begin{array}{l}
\textbf{DeleteThread}(tcb) \\
\quad thread_space := \{\ tcb\ \} \lhd\!\!\!- \ thread_space \ \ \| \\
\quad thread_state := \{\ tcb\ \} \lhd\!\!\!- \ thread_state \ \ \| \\
\quad thread_pager := \{\ tcb\ \} \lhd\!\!\!- \ thread_pager \ \ \| \\
\quad thread_scheduler := \{\ tcb\ \} \lhd\!\!\!- \ thread_scheduler \ \ \| \\
\quad thread_gno := \{\ tcb\ \} \lhd\!\!\!- \ thread_gno
\end{array}
$$

Furthermore, if the thread is the only one left in the address space, we delete the address space, otherwise we decrement the thread counter.

$$
\begin{array}{l}
\textbf{IF} \quad tcb = thread_space\ ^{-1}\ [\ \{\ thread_space\ (\ tcb\ )\ \}\ ] \\
\textbf{THEN} \\
\quad DeleteAddressSpace\ (\ thread_space\ (\ tcb\ )\ ) \ \ \| \\
\quad threads_in_space := \\
\quad\quad \{\ thread_space\ (\ tcb\ )\ \} \lhd\!\!\!- \ threads_in_space \\
\textbf{ELSE} \\
\quad threads_in_space\ (\ thread_space\ (\ tcb\ )\ ) := \\
\quad\quad threads_in_space\ (\ thread_space\ (\ tcb\ )\ ) - 1 \\
\textbf{END}
\end{array}
$$

Apart from creating, deleting and activating, the ThreadControl API section in L4 contains a number of further operations on threads which we explain below.

Modifying the thread's scheduler is one of them. The *SetScheduler* operation is trivial, assuming only that the thread and the scheduler exist, and updating the thread's scheduler. We omit its definition in B here.

We also omit the *Migrate* operation. It performs the same task as the migration in *ActivateThread*.

As our description of the formalisation progresses it becomes obvious that there is much statements duplication. Indeed, the next operation in the machine is *MigrateAndSetScheduler* which demonstrates the problem again. As mentioned before, this is a restriction of the B system. Writing *Migrate* and *SetScheduler* does not mean a higher-level machine can combine them to get *MigrateAndSetScheduler*. If they are to be combined, statements must be duplicated. We believe this to be a shortcoming of the B Method as used by the B Toolkit [2]. One solution to this is to defer any specific actions to refined machines, then use sequential composition in the refinements. The problem with this is that the resulting top-level machines cannot be meaningfully animated, making validation of the formal model more difficult.

We next define the *SetState* operation, taking *tcb* and *state*, with the restrictions that *tcb* is active, not an interrupt thread (since they have different state semantics) and *state* cannot be *tsAborted* (transition to an aborted state would imply thread deactivation, which L4 does not provide):

$$
\begin{array}{l}
\textbf{SetState}\ (tcb,\ state) \quad \widehat{=} \\
\quad \textbf{PRE} \quad state \in THREAD_STATE \wedge \\
\quad\quad state \neq tsAborted \wedge \\
\quad\quad tcb \in active_threads \wedge \\
\quad\quad tcb \notin kIntThreads \\
\quad \textbf{THEN} \\
\quad\quad thread_state\ (\ tcb\ ) := state \\
\quad \textbf{END}
\end{array}
$$

To implement the ExchangeRegisters [13, section 2.3] system call, we need to specify its semantics with respect to the variables in the *Thread* machine. Its 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 section 4.1)

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

unwait should the target thread be woken; this is to correctly set the thread state if a machine including this one, such as *IpcBase*, uses its equivalent of *ExchangeRegisters* to cancel waiting or polling IPC states (see section 4.4.3).

Since there is also no specification of user-level registers saved in the kernel, IP, SP and FLAGS are not passed in. The semantics of *ExchangeRegisters* means a thread can only invoke it on another thread in its address space. Since no thread can be in the reserved kernel address space, interrupt threads are excluded. The operation sets the pager (if $ex_p \in control$), halts the thread (if $ex_H \in control$, resumes if not) and resets any waiting states ($unwait = \mathrm{TRUE}$)

$$
\begin{array}{l}
\textbf{ThreadExchangeRegisters}(tcb,\ control,\ pager,\ unwait) \quad \widehat{=} \\
\quad \textbf{PRE} \quad tcb \in threads \wedge control \subseteq EXREGS_FLAGS \wedge \\
\quad\quad pager \in TCB \wedge tcb \notin kIntThreads \wedge unwait \in BOOL \\
\quad \textbf{THEN} \\
\quad\quad \textbf{IF} \quad ex_p \in control \quad \textbf{THEN} \\
\quad\quad\quad thread_pager\ (\ tcb\ ) := pager \\
\quad\quad \textbf{END} \quad \| \\
\quad\quad \textbf{IF} \quad ex_h \in control \quad \textbf{THEN} \\
\quad\quad\quad \textbf{IF} \quad ex_H \in control \quad \textbf{THEN} \\
\quad\quad\quad\quad halted_threads := halted_threads - \{\ tcb\ \} \\
\quad\quad\quad \textbf{ELSE} \\
\quad\quad\quad\quad halted_threads := halted_threads \cup \{\ tcb\ \} \\
\quad\quad\quad \textbf{END} \\
\quad\quad \textbf{END} \quad \| \\
\quad\quad \textbf{IF} \quad unwait = TRUE \quad \textbf{THEN} \\
\quad\quad\quad \textbf{IF} \quad tcb \in active_threads \quad \textbf{THEN} \\
\quad\quad\quad\quad thread_state\ (\ tcb\ ) := tsRunning \\
\quad\quad\quad \textbf{ELSE} \\
\quad\quad\quad\quad thread_state\ (\ tcb\ ) := tsAborted \\
\quad\quad\quad \textbf{END} \\
\quad\quad \textbf{END} \\
\quad \textbf{END}
\end{array}
$$

There are two special operations for interrupt threads: *ActivateInterrupt* and *DeactivateInterrupt*. In L4, activating an interrupt thread means setting it to *halted* and setting its pager to a value other than itself:

```
ActivateInterrupt(tcb, handler)  ≙
  PRE   tcb ∈ kIntThreads ∧ handler ∈ TCB ∧
    handler ≠ tcb
  THEN
    halted_threads := halted_threads ∪ { tcb }  ||
    thread_pager ( tcb ) := handler
  END
```

To deactivate an interrupt thread, we perform the opposite: the pager is set to itself and the *halted* flag is reset. The definition is analogous to *ActivateInterrupt* and we omit it here.

We will now describe helper-operations enabling IPC-related state transitions. The IPC state transitions themselves are the subject of the next section. The first operation is *Unwait*, which reverts a waiting or polling (waiting to send) thread to its normal state. For normal threads this is *running*; for inactive and interrupt threads it is *aborted*.

```
UnWait(tcb)  ≙
  PRE   tcb ∈ threads   THEN
    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
  END
```

A *SELECT* statement non-deterministically selects one of the cases whose condition is true and evaluates the statement contained in the *THEN* clause. If no condition is true, the *ELSE* clause is evaluated.

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

- the other thread is not waiting: the running thread polls — *SetState* is used
- the other thread is waiting, no receive phase is included: the IPC occurs, the remote thread is woken with *UnWait*
- as above, but a non-trivial receive phase is included: the IPC occurs, the remote thread is woken, but the current thread starts waiting — *WakeUpAndWait* is used

The *WakeUpAndWait* operation takes a running thread, a waiting thread and a the wait state the sending thread is to assume. Both threads must be active, the first must be running (*isRunning* tests for equality with *tsRunning*), the second must be waiting (*tsWaitingForever* or *tsWaitingTimeout*):

```
WakeUpAndWait(running_tcb, waiting_tcb, wait_state)  ≙
  PRE   running_tcb ∈ active_threads ∧
    waiting_tcb ∈ active_threads ∧
    isWaiting ( wait_state ) ∧
    isRunning ( thread_state ( running_tcb ) ) ∧
    isWaiting ( thread_state ( waiting_tcb ) )
  THEN
  IF   waiting_tcb ∈ kIntThreads   THEN
    thread_state := thread_state <+
                    { running_tcb ↦ wait_state ,
                      waiting_tcb ↦ tsAborted }
  ELSE
    thread_state := thread_state <+
                    { running_tcb ↦ wait_state ,
                      waiting_tcb ↦ tsRunning }
    END
  END
```

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. To handle this case, we use the *DualWakeUp* operation, which simply takes a polling and waiting thread and essentially performs an *Unwait* on each. We omit the obvious formal definition.

4.4 Inter Process Communication

This section describes the machines *IpcCore* and *IpcBase* which cover the IPC-related operations in L4.

Inter process communication is the core component of L4. Nearly all aspects of the system are abstracted by IPC when possible, including donation and leasing of memory to other processes. IPC is *synchronous*; 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 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 receiver's 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.

At first glance, the *IpcCore* machine does not seem very useful, as it has no state and contains only a single place-holder operation *PerformIpc*, which does nothing. 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 specification. They could be added in a later refinement step.

We decided to leave out MRs at this level of the specification, because they contain too much implementation detail. When an IPC is performed, MRs do not merely get transferred, but can also contain information on memory maps and grants which would have duplicated the efforts of the VM subsystem part of the verification pilot project.

The machine contains a useful definition *canIPC* representing 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): the thread must be active, running and not halted (except for interrupt threads, which must be halted to be enabled):

```
canIPC(t)  ≙
  t ∈ active_threads ∧
  ( t ∈ kIntThreads ⇒ t ∈ halted_threads ) ∧
  ( t ∉ kIntThreads ⇒ thread_state ( t ) = tsRunning ∧
                      t ∉ halted_threads )
```

The *IpcBase* 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. The IPC operations are nondeterministic, i.e. there are situations which might cause them to fail which are not explicitly contained in this specification. This means that the first possibility of failure is in this machine. The operations

in the previous machines always succeeded given the preconditions. In L4, the error condition is stored in the Error Thread Control Register, which means we have to specify some form of this TCR in the *IpcBase* machine.

Unfortunately, B's inability to perform two operations from the same machine in parallel means that the promoted operation cannot clear the Error TCR themselves. As a work-around we introduce new local versions of these operations, which only add that error functionality and shadow all operations which might normally just be promoted.

4.4.1 Variables

The *IpcBase* machine 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 the *Thread* machine.

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 [13, section 5.6]:

$$\begin{aligned} & thread_ipc_waiting_for \in \\ & \quad active_threads \mapsto GLOBAL_TNO \wedge \\ & kNilGNo \notin \mathsf{ran}\ (\ thread_ipc_waiting_for\ ) \end{aligned}$$

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, so *thread_ipc_waiting_for* cannot have *active_threads*, or even *threads* as its permitted range.

We not only need to know *that* a thread is waiting, but also for *how long* it is waiting. We therefore define the *thread_ipc_waiting_timeout* function. Its range is identical to the one of *thread_ipc_waiting_for*, but it specifies the timeout for waiting threads:

$$\begin{aligned} & thread_ipc_waiting_timeout \in \\ & \quad active_threads \mapsto TIMEOUT \wedge \\ & eZeroTimeout \notin \mathsf{ran}\ (\ thread_ipc_waiting_timeout\ ) \wedge \\ & \mathsf{dom}\ (\ thread_ipc_waiting_timeout\ ) = \\ & \quad \mathsf{dom}\ (\ thread_ipc_waiting_for\ ) \wedge \\ & \mathsf{dom}\ (\ thread_ipc_waiting_timeout\ ) = \\ & \quad thread_state^{-1}\ [\ \{\ tsWaitingTimeout\ , \\ & \qquad tsWaitingForever\ \}\ ] \end{aligned}$$

All the threads in the domain must either be waiting with a timeout, or waiting forever. No thread with a waiting state may be absent, and no thread in the function's domain 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 is not permitted, since those calls are resolved immediately without forcing the thread to wait.

Similar to the two functions above, we define polling semantics for threads:

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

In L4, each thread keeps track of which thread it is polling on (if any), as well as keeping a list of threads which are polling on it. The deletion of a thread which another thread is polling on is not well defined (neither in manual nor source code). In our formalisation, we only remove the target thread from the range of *thread_ipc_polling_on* and let the IPC time out. For this reason, there can be some threads in a polling state which are not actually polling for any thread. All polling threads must have a timeout, even if the thread they are polling on was deleted, otherwise they will never return to running.

The convenience function *thread_incoming*, given a thread, yields the threads that are polling on it. It represents each thread's incoming IPC buffer and is just the relational inverse of all threads on *thread_ipc_polling_on*:

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

For some operations, we need the thread numbers (IDs) of incoming threads. Again, to simplify the formalisation we define *thread_incoming_gnos* as:

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

In addition to the above, when the send phase of an IPC succeeds, the receive phase is invoked. If no receive was requested, the thread goes directly back to running. Otherwise the receive is performed, which means if no candidates are available, the thread must wait. We store a timeout for each polling thread. The *thread_recv_waiting* functions only differ from the *thread_ipc_waiting* functions in that not all polling threads will have a receive phase, and no waiting thread may have a future receive phase:

$$\begin{aligned} & \mathsf{dom}\ (\ thread_recv_waiting_for\ ) \subseteq \\ & \quad \mathsf{dom}\ (\ thread_ipc_polling_timeout\ ) \wedge \\ & \mathsf{dom}\ (\ thread_recv_waiting_timeout\ ) \cap \\ & \quad \mathsf{dom}\ (\ thread_ipc_waiting_timeout\ ) = \{\} \end{aligned}$$

The variable *thread_error* represents the concept of each thread having some error condition which resulted from a previous operation:

$$thread_error \in active_threads \rightarrow ERROR$$

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 [13], since *ERROR* contains *eNoError*, a condition to signify success. L4 would put the success/failure result into a register instead.

4.4.2 Initialisation

We now describe, how the variables defined in the section before are initialised. 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_recv_** 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:

$$\begin{aligned} & thread_incoming :\in \\ & \quad \{\ kSigma0\ , kRootServer\ \} \cup kIntThreads \rightarrow \{\ \{\}\ \}\ \ \| \\ & thread_incoming_gnos :\in \\ & \quad \{\ kSigma0\ , kRootServer\ \} \cup kIntThreads \rightarrow \{\ \{\}\ \} \end{aligned}$$

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

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

$$\begin{aligned} & thread_error := \{\ kSigma0 \mapsto eNoError\ , \\ & \qquad kRootServer \mapsto eNoError\ \} \cup \\ & \qquad kIntThreads \times \{\ eNoError\ \} \end{aligned}$$

4.4.3 Operations

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 *2* appended to their name.

The first of these is *ActivateThread2*, which sets up the local variables when the thread is activated, and invokes the original *ActivateThread*. Their preconditions and parameters are the same, except that we now need to specify who a freshly activated thread must wait for an IPC from. In L4, that is its pager. Since it cannot run until the message is received, it will wait forever:

```
ActivateThread2(tcb, space, pager, scheduler)  ≙
  PRE    tcb ∈ threads ∧ tcb ∉ active_threads ∧
    tcb ≠ pager ∧
    thread_space ( tcb ) ∈ initialised_spaces ∧
    pager ∈ active_threads ∧
    scheduler ∈ active_threads ∧
    space ∈ initialised_spaces ∧
    ( space ≠ thread_space ( tcb ) ⇒
    threads_in_space ( space ) < kMaxThreadsPerSpace )
  THEN
    ActivateThread ( tcb , space , pager , scheduler )  ||
    thread_ipc_waiting_timeout ( tcb ) :=
        eInfiniteTimeout  ||
    thread_ipc_waiting_for ( tcb ) := thread_gno ( pager )  ||
    thread_error ( tcb ) := eNoError  ||
    thread_incoming ( tcb ) :=
        thread_ipc_polling_on ⁻¹ [ { tcb } ]  ||
    thread_incoming_gnos ( tcb ) :=
        thread_gno [ thread_ipc_polling_on ⁻¹ [ { tcb } ] ]
  END
```

We need to set the thread's error condition to some value and since no error has occurred, that is *eNoError*. Additionally, some threads may already be polling for this thread ID, so the *thread_incoming* and *thread_incoming_gnos* functions are updated appropriately.

The *CreateActiveThread2* operation is augmented analogously and we omit its definition here.

Next we add necessary statements to *DeleteThread* which clean up the affected variables in this machine. The preconditions and parameters do not change. Apart from the obvious domain subtraction of $\{tcb\}$ from *thread_ipc_waiting_**, *thread_ipc_polling_timeout*, *thread_recv_** and *thread_error*, we need to remove the thread from both the range and domain of *thread_ipc_polling_on*:

```
thread_ipc_polling_on :=
  { tcb } ⩤ thread_ipc_polling_on ⩥ { tcb }
```

The application of the domain subtraction (⩤, precedence is left-to-right) removes all mappings denoting that this thread is polling on another one (there is only one). Then, the application of range subtraction (⩥) removes all mappings denoting that 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. Range subtraction is defined canonically:

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

This resolves the situation of who is polling on whom. The thread still needs to be removed from the incoming sets of other threads:

```
thread_incoming :=
  { aa , bb  |
      aa ∈ dom ( thread_incoming ) − { tcb } ∧
      bb = thread_incoming ( aa ) − { tcb } }
```

We do the removal via a set comprehension which keeps only other threads' incoming sets, but also removes the deleted thread from them. We apply the same technique to modify *thread_incoming_gnos*.

Operations from previous machines that complete successfully need to clear the error attribute of the thread they are operating on. Hence, we extended these operations with that functionality at the *IpcBase* machine level. Their preconditions are almost the same as their *Thread* counterparts', and the operation body invokes them directly. The only difference is that they take an extra parameter (*itcb*) that says which thread's error attribute should be cleared. We omit the B definition of these machines. They are: *InitialiseAddressSpace2*, *CreateThread2*, *SetScheduler2*, *Migrate2*, *MigrateAndSetScheduler2*, *ActivateInterrupt2* and *DeactivateInterrupt2*.

We now cover the operations directly related to IPC. In all operations, the invoking thread is checked with the *canIpc* definition of the *IpcCore* machine.

The first of the operations enabling IPC is *JustWait*. It is invoked 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, thus causing the receiver thread to wait. 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 *canIpc*; *timeout* is either finite or infinite, but *not zero* (instant time-out); *fromSpecifier* is not *nilthread* and is either *anythread* or a known thread. Also, the we exclude all conditions which would cause an immediate IPC reception to occur: if *fromSpecifier* is *anythread*, the requester must not have incoming threads; for other specifiers, they must not be in the incoming thread numbers. The work done by the operation is minimal. It updates *thread_ipc_waiting_for* and its time-out equivalent to indicate the thread is waiting and who it is waiting for. It also uses *SetState* to set the thread's state to *tsWaitingForever* or *tsWaitingTimeout* depending on the value of *timeout*.

```
JustWait(tcb, timeout, fromSpecifier)  ≙
  PRE    canIPC ( tcb ) ∧ timeout ∈ TIMEOUT ∧
    ¬ ( isNoTimeout ( timeout ) ) ∧
    fromSpecifier ∈ GLOBAL_TNO ∧
    fromSpecifier ≠ kNilGNo ∧
    fromSpecifier ∈
        thread_gno [ threads ] ∪ { kAnyGNo } ∧
    ( fromSpecifier = kAnyGNo ⇒
        thread_incoming ( tcb ) = {} ) ∧
    ( fromSpecifier ≠ kAnyGNo ⇒
    fromSpecifier ∉ thread_incoming_gnos ( tcb ) )
  THEN
    thread_ipc_waiting_for ( tcb ) := fromSpecifier  ||
    thread_ipc_waiting_timeout ( tcb ) := timeout  ||
     IF    isInfinite ( timeout )    THEN
        SetState ( tcb , tsWaitingForever )
     ELSE
        SetState ( tcb , tsWaitingTimeout )
     END
  END
```

We have covered threads that want to receive, but cannot. The *SetUpReceivePhaseAndPoll* 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). The operation takes *tcb_from* and *tcb_to* (sending and target

threads), *poll_timeout*, *recv_timeout* (time-out for the future receive phase) and a *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 time-out is only restricted if *fromSpecifier* is not *nilthread*. The actual *fromSpecifier* must be a thread number of an existing thread.

```
SetUpReceivePhaseAndPoll(tcb_from, tcb_to, poll_timeout,
    recv_timeout, fromSpecifier)  ≙
  PRE   canIPC ( tcb_from ) ∧ tcb_to ∈ threads ∧
    ( tcb_to ∈ dom ( thread_ipc_waiting_for ) ⇒
        thread_ipc_waiting_for ( tcb_to ) ≠
            thread_gno ( tcb_from ) ∧
        thread_ipc_waiting_for ( tcb_to ) ≠ kAnyGNo ) ∧
    fromSpecifier ∈ GLOBAL_TNO ∧
    poll_timeout ∈ TIMEOUT ∧
    ¬ ( isNoTimeout ( poll_timeout ) ) ∧
    recv_timeout ∈ TIMEOUT ∧
    ( fromSpecifier ≠ kNilGNo ⇒
        ¬ ( isNoTimeout ( recv_timeout ) ) ) ∧
    fromSpecifier ∈ thread_gno [ threads ] ∪
                    { kAnyGNo , kNilGNo }
  THEN
  thread_ipc_polling_on ( tcb_from ) := tcb_to  ||
  thread_ipc_polling_timeout ( tcb_from ) := poll_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 ) }  ||
  SetState ( tcb_from , tsPolling )  ||
  IF   fromSpecifier ≠ kNilGNo   THEN
    thread_recv_waiting_for ( tcb_from ) :=
        fromSpecifier  ||
    thread_recv_waiting_timeout ( tcb_from ) :=
        recv_timeout
  END
END
```

To verify that the operation happens in the aforementioned circumstances, 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:

```
( tcb_to ∈ dom ( thread_ipc_waiting_for ) ⇒
thread_ipc_waiting_for ( tcb_to ) ≠
   thread_gno ( tcb_from ) ∧
thread_ipc_waiting_for ( tcb_to ) ≠ kAnyGNo )
```

Once that is established, the operation can proceed successfully by *updating thread_ipc_polling_** to reflect the thread's polling information, and also adds *from_tcb* and its thread number to the incoming sets of *tcb_to*. If *fromSpecifier* is not *nilthread*, *thread_recv_** is updated with the future waiting information.

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*, *JustReceive* takes two parameters (whose meaning is the same): *itcb* and *fromSpecifier*.

JustReceive(*itcb*, *fromSpecifier*)

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 the operation being aborted halfway by another thread). We model this failure by non-determinism.

```
canIPC ( itcb ) ∧ fromSpecifier ∈ GLOBAL_TNO ∧
fromSpecifier ≠ kNilGNo ∧
( fromSpecifier ≠ kAnyGNo ⇒
   fromSpecifier ∈ thread_incoming_gnos ( itcb ) ) ∧
( fromSpecifier = kAnyGNo ⇒
   thread_incoming ( itcb ) ≠ {} )
```

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

```
ANY    tcb_from
WHERE    tcb_from ∈ thread_incoming ( itcb ) ∧
  ( fromSpecifier ≠ kAnyGNo ⇒
  thread_gno ( tcb_from ) ∈
      thread_incoming_gnos ( itcb ) )
```

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 of the IPC succeeding or failing, the sending thread will no longer be polling at the end of the operation:

```
thread_ipc_polling_on :=
   { tcb_from } ⩤ thread_ipc_polling_on  ||
thread_ipc_polling_timeout :=
   { tcb_from } ⩤ thread_ipc_polling_timeout
thread_incoming ( itcb ) :=
   thread_incoming ( itcb ) − { tcb_from }  ||
thread_incoming_gnos ( itcb ) :=
   thread_incoming_gnos ( itcb ) −
   { thread_gno ( tcb_from ) }
```

Since it will no longer 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:

```
thread_recv_waiting_timeout := { tcb_from } ⩤
   thread_recv_waiting_timeout  ||
thread_recv_waiting_for := { tcb_from } ⩤
   thread_recv_waiting_for
```

We now reach the point where IPC either succeeds or fails. This is performed using the non-deterministic CHOICE *path1* OR *path2* END construct. During animation, the user is asked to choose the path.

On the IPC success path, the IPC transfer is performed and the error fields for both threads are cleared:

```
PerformIPC ( tcb_from , itcb )  ||
   thread_error := thread_error ⩥
   { itcb ↦ eNoError , tcb_from ↦ eNoError }
```

Then, if the sender had a receive phase waiting, it is set up (using identical statements to those in *JustWait*). 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*.

The IPC failure path consists of cancelling the sender's waiting state with *UnWait* and picking an

error non-deterministically among the possible unpredictable IPC errors (see section 4.1) and assigned as an error indicator for both threads.

WakeDestThenWait covers the opposite direction to *JustReceive*: 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.

WakeDestThenWait(*tcb_from*, *tcb_to*, *recv_timeout*, *fromSpecifier*)

The precondition combines aspects of the previous IPC operations: the destination must be waiting for either the source's thread number or *anythread*; *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 success path begins as for *JustReceive* 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 } ⩤ thread_ipc_waiting_for ∪
        { tcb_from ↦ fromSpecifier }  ||
IF  isInfinite ( recv_timeout )  THEN
  thread_ipc_waiting_timeout :=
  { tcb_to } ⩤ thread_ipc_waiting_timeout ∪
        { tcb_from ↦ eInfiniteTimeout }  ||
  WakeUpAndWait ( tcb_from , tcb_to ,
        tsWaitingForever)
ELSE  ...
```

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

We leave out the formal definition of the failure path here. It just removes the destination from *thread_ipc_waiting_**, picks an error, sets it as the error attribute for both threads, and uses *UnWait* on the destination thread.

The *ResolveIPC* operation covers the situation where one thread$_1$ is polling on thread$_2$, while the latter is waiting for the former. Since neither of them is executing, the kernel will perform the IPC. We omit the formal definition of **ResolveIPC**(*tcb_from*, *tcb_to*). It is a combination of *JustReceive* and *WakeDestThenWait*. The main difference is the precondition. It requires that both threads are active, the sender is polling and the receiver is waiting, the sender is polling on the receiver and the receiver either accepts *anythread* or the receiver's thread number.

When the kernel finds a thread that has been polling for longer than its time-out value, a time-out occurs. Since the model abstracts from exact values for time-outs, we let the time-out occur non-deterministically. The *TimeoutPoll* 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. 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. The *TimeoutPoll* operation removes the thread from the state variables and sets its error attribute to *eSendTimeout*. The *TimeoutWait* operation is the equivalent of *TimeoutPoll*, but times out a thread which is waiting with a finite time-out.

```
TimeoutPoll  ≙
  BEGIN
    IF  thread_ipc_polling_timeout ▷
      { eFiniteTimeout } = {}
     THEN
      skip
    ELSE
      ANY  tcb
      WHERE
        tcb ∈ dom ( thread_ipc_polling_timeout ▷
                { eFiniteTimeout } )
      THEN
        UnWait ( tcb )  ||
        thread_ipc_polling_on :=
          { tcb } ⩤ thread_ipc_polling_on  ||
        thread_ipc_polling_timeout :=
          { tcb } ⩤ 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 } ⩤ thread_recv_waiting_timeout  ||
        thread_recv_waiting_for :=
          { tcb } ⩤ thread_recv_waiting_for  ||
        thread_error ( tcb ) := eSendTimeout
      END
    END
  END
```

The *IpcBase* machine also provides an operation *SetError* as a way for operations in higher-level machines to set the error attribute for an active thread. This is used for example, to signal that a thread lacks necessary privileges to perform an operation.

When we described the ExchangeRegisters functionality in section 4.3.3, we only covered the functionality pertaining directly to threads and their control state. As the reference manual [13, section 2.3] states, ExchangeRegisters can be used to cancel or abort ongoing IPCs. Now that the IPC state transitions are available, we can model the IPC functionality in ExchangeRegisters. *IpcBaseExchangeRegisters* takes one fewer parameter than *ThreadExchangeRegisters*. 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.

```
IpcBaseExchangeRegisters(tcb, control, pager)  ≙
  PRE  tcb ∈ threads ∧ control ⊆ EXREGS_FLAGS ∧
    pager ∈ TCB ∧ tcb ∉ kIntThreads
  THEN
    IF
      ex_S ∈ control ∧
      tcb ∈ dom ( thread_ipc_polling_on )
    THEN
      ThreadExchangeRegisters(tcb, control, pager,
                        TRUE)  ||
```

```
            thread_ipc_polling_on :=
                { tcb } ⩤ thread_ipc_polling_on  ||
            thread_ipc_polling_timeout :=
                { tcb } ⩤ 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 } ⩤ thread_recv_waiting_timeout  ||
            thread_recv_waiting_for :=
                { tcb } ⩤ thread_recv_waiting_for  ||
             ANY  err  WHERE
                err ∈ { eSendCancelled , eAborted }
             THEN
                thread_error ( tcb ) := err
             END
        ELSIF
            ex_R ∈ control ∧
            tcb ∈ dom ( thread_ipc_waiting_for )
        THEN
            ThreadExchangeRegisters(tcb, control, pager,
                                    TRUE)  ||
            thread_ipc_waiting_for :=
                { tcb } ⩤ thread_ipc_waiting_for  ||
            thread_ipc_waiting_timeout :=
                { tcb } ⩤ thread_ipc_waiting_timeout  ||
             ANY  err  WHERE
                err ∈ { eRecvCancelled , eAborted }
             THEN
                thread_error ( tcb ) := err
             END
        ELSE
            ThreadExchangeRegisters (tcb, control, pager,
                                     FALSE)  ||
            thread_error ( tcb ) := eNoError
        END
    END
```

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 $R = 1$, likewise, 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. Parallel composition means 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*.

This concludes the operations of *IpcBase*. Having defined all the core functionality present in the model, we can now show an accurate view of state transitions in figure 4.

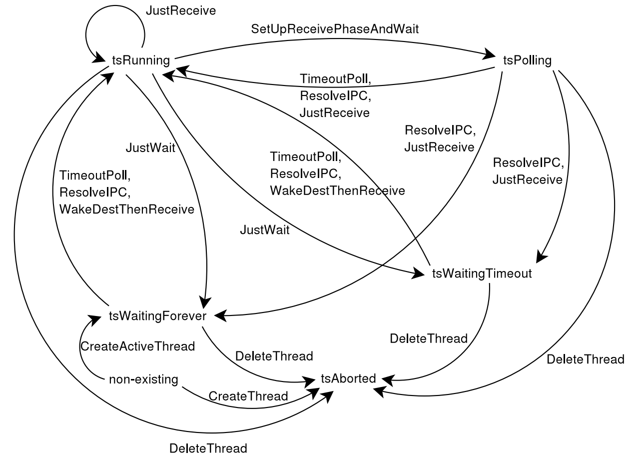

Figure 4: Possible state transitions in the model and operations which cause them.

4.5 API

This is the topmost machine in the specification. It *INCLUDES WeakSysCall* and all 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 precondition 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.

It is worth noting 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.

To give the reader an idea of what such an operation looks like, we present the final version of *ExchangeRegisters*. It augments the *IpcBaseExchangeRegisters* operation with the error-checking necessary to make it succeed, as well as non-deterministically modelling the system call components not within the scope of the formalisation:

```
ExchangeRegisters ( itcb , tcb , control , sp , ip , flags ,
                    pager , handle )  ≙
    PRE  itcb ∈ active_threads ∧
      thread_state ( itcb ) = tsRunning ∧
      tcb ∈ TCB ∧ control ⊆ EXREGS_FLAGS ∧
      sp ∈ ℕ ∧ ip ∈ ℕ ∧
      pager ∈ TCB ∧ flags ∈ ℕ ∧ handle ∈ ℕ
    THEN
        SELECT  tcb ∉ threads  THEN
          SetError ( itcb , eInvalidThread )
        WHEN  tcb ∈ threads ∧
          thread_space ( tcb ) ≠ thread_space ( itcb )
        THEN
          SetError ( itcb , eInvalidThread )
        ELSE
            CHOICE
              IpcBaseExchangeRegisters ( tcb , control , pager )
            OR
                ANY  error
                WHERE  error ∈ { eOutOfMemory ,
                                 eInvalidUtcbLocation }
                THEN
                  SetError ( itcb , error )
                END
            END
        END
    END
```

As mentioned before, the two remaining non-trivial assumptions are that the invoking thread *itcb* must be active and running (otherwise it cannot perform a system call). Via the non-deterministic *SELECT* statement with exclusive conditions, we enforce the preconditions of *IpcBaseExchangeRegisters*: the target thread *tcb* must exist and be in the caller's address space. If the preconditions are met the operation will succeed, but this is not necessarily true of the system call [13, section 2.3]: we may be out of memory or point to a bad memory location. Since the virtual memory subsystem is outside the scope of this formalisation, we model these failures via non-deterministic choice. The instruction and stack pointers, due to no knowledge of memory layout, are ignored; so is the user-defined *handle*, since it has no effect on actions performed by the kernel.

5 Conclusion

In this paper we have described our formalisation of the L4 high-performance microkernel in the B method. The main work on the formalisation was done as the honours thesis project of the first author which equates to an investment of roughly 5 person months. The final formalisation extends to about 2000 lines of B specification.

The goals of the formalisation effort were reached. The model is animatable and can be used as a learning tool. During the project it became apparent that in spite of detailed, good quality documentation, there were a number of ambiguities in the description of the L4 API and even inconsistent expectations towards its behaviour. Using code inspection and discussions with L4 developers those could be resolved, made precise and documented in the model. We are confident that the formalisation provides a good basis for the planned revision of the L4 API that involves formal modelling from the start.

The level of detail that was achieved during the available time frame suggests that formal specification of real-world operating system kernels is entirely feasible, a good opportunity for documentation, and a good starting point for verification of the system.

The context of this work is a pilot project on exactly that: the verification of the L4 microkernel. In other work [20] we have demonstrated the feasibility of this as well. With an investment of about 1.5 person years we were able to specify a significant part of the L4 virtual memory subsystem and fully verify it down to C code, integrated into the kernel, running on real hardware.

Despite the good results and progress we achieved using the B method, we found that a number of restrictions were hindering our work. They are not directly a fault of the B method itself, but more of an incompatibility with our goals. The requirement for animation for instance, precluded some more convenient formalisation mechanisms. Furthermore, the B method is geared towards refinement proofs in multiple steps with code generation at the end. We do agree in principle with this technique, but although the code generation step from B to the C programming language seems appropriate for application code, it bridges too large a gap to effectively control performance critical sections in operating systems code.

We therefore decided to use the other formalism that was successfully applied in the pilot project, Isabelle/HOL [16], for the future, full verification. The main concepts of the B formalisation — a state based description of the L4 API functions will stay the same, only the notation will be different (higher order logic instead of set theory).

In fact, we are not performing a translation from B to Isabelle/HOL, but we are first developing a new version of the L4 API that introduces efficient and flexible security mechanisms. As most of the API will stay unchanged, the hope is that the experience gained in this formalisation will significantly speed up the formal specification process. Moreover, the formal specification this time is integrated with the API design from the start.

We estimate that the full verification of L4 will take about 20 person years, including verification tool development. This effort must be seen in relation to the cost of developing the kernel in the first place, and the potential benefits of verification. The present kernel was written by a three-person team over a period of 8–12 months, with significant improvements since. Furthermore, for most of the developers it was the third in a series of similar kernels they had written, which meant that when starting they had a considerable amount of experience. A realistic estimate of the cost of developing a high-performance implementation of L4 is probably at least 5–10 person years.

Under those circumstances, we argue that the full verification of L4 is highly desirable and provides a good return of investment. The kernel is the lowest and most critical part of any software stack, and any assurances on system behaviour are built on sand as long as the kernel is not shown to behave as expected. Furthermore, formal verification puts pressure on kernel designers to simplify their systems, which has obvious benefits for maintainability and robustness even when not yet formally verified.

Acknowledgements We thank Ken Robinson who supervised the honours thesis, Kevin Elphinstone who assessed it, and the L4 development team for their help and useful discussions. We also thank Ansgar Fehnker for reading drafts of this paper.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] B-Core. The B-Toolkit. http://www.b-core.com/btoolkit.html, 2002.

[3] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[4] W. R. Bevier and L. M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., 1994.

[5] T. Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, 1994.

[6] G. Duval and J. Julliand. Modelling and verification of the RUBIS μ-kernel with SPIN. In *SPIN'95, Workshop on Model Checking of Software*, 1995.

[7] S. Fowler and A. Wellings. Formal analysis of a real-time kernel specification. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135, pages 440–458, Uppsala, Sweden, 1996. Springer-Verlag.

[8] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, Oxford, UK, 2005. to appear.

[9] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.

[10] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[11] R. Kolanski. A formal model of the μ-kernel api using the B method. Honours Thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, 2004.

[12] L4 development team. L4ka::pistachio source code v0.3. http://www.l4ka.org/download/, 2004.

[13] L4Ka Team. *L4 eXperimental Kernel Reference Manual Version X.2r3*, 2004. http://www.l4ka.org.

[14] J. Liedtke. Towards real μ-kernels. *Communications of the ACM*, 39(9):70–77, 1996.

[15] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.

[16] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[17] K. Robinson. A concise summary of the B mathematical toolkit. http://www.cse.unsw.edu.au/~cs2110/B-Summary/, 2005.

[18] J. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS97-26, University of Pennsylvania, Philadelphia, PA, USA, 1997.

[19] J. M. Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5):21–28, September 1990.

[20] H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In G. Klein, editor, *Proc. NICTA FM Workshop on OS Verification*, pages 73–97. Technical Report 0401005T-1, National ICT Australia, 2004.

[21] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *HotOS-VI*, 1997.

[22] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.