Towards Verified Virtual Memory in L4

Gerwin Klein and Harvey Tuch

[1] University of New South Wales, Sydney 2052, Australia
[2] National ICT Australia*, Sydney, Australia
{gerwin.klein|harvey.tuch}@nicta.com.au

Abstract. We report on the initial stage of an on-going verification project: the formalisation and verification of the L4 μ-kernel. We describe an abstract model of the virtual memory subsystem in L4, prove safety properties about this model, and describe refinement of the abstract model towards the implementation of L4. All formalisations and proofs have been carried out in the theorem prover Isabelle.

1 Introduction

L4 is a second generation microkernel based on the principles of minimality, flexibility, and efficiency [10]. 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 about an order of magnitude smaller than Mach and two orders of magnitude smaller than Linux.

The operating system (OS) is clearly one of the most fundamental components of non-trivial systems. The correctness and reliability of the system critically depends on the OS. 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 these requirements of correctness, reliability, and security. Microkernels address this problem by applying the principles of minimality and least privilege to operating system architecture. However, the success of this approach is still predicated on the microkernel being designed and implemented correctly. We can address this by formally modelling and verifying it.

L4 has a design that is not only geared towards flexibility and reliability, but is of a size which makes formalisation and verification feasible. Compared to other operating system 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 Fig. 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

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council

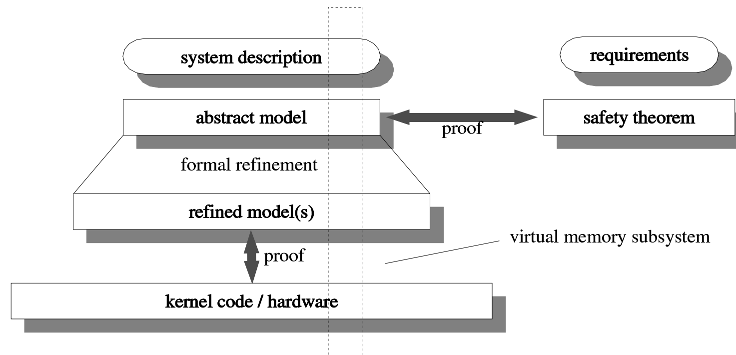

Fig. 1. Overview

the L4 reference manual [1]. 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.

To keep complexity and time manageable, we have decided to take a thin vertical slice out of this refinement process and to test the methodology on one non-trivial subsystem of the kernel initially. This will not give hard safety guarantees about the full system, but it will increase confidence in the implementation and improve understanding of the target subsystem. The goal is to move through the full process quickly and to uncover problems in the interaction of refinement layers and the different formalisms utilised.

In this paper we report on first experiences with this project. L4 provides three main abstractions: threads, address spaces, and inter-process communication (IPC). We have chosen to start with address spaces. This is supported by the virtual memory subsystem of the kernel and is fundamental for implementing separation and security policies on top of L4. We have built an abstract model of address spaces and we show a first refinement of it.

One of the central questions in any verification project is: When exactly is the specification of the system correct? What is the system supposed to do? In this case we have taken the L4 X.2 API description as the main reference [1] and use the L4Ka::Pistachio [8] implementation on the ARM architecture to resolve ambiguities and address implementation issues, in addition to discussions with the developers on the *pistachio-core* mailing list.

As we are mainly trying to test the methodology, we are making some simplifying assumptions in the formalisation. We are also not planning to verify the current implementation of L4Ka::Pistachio. On the contrary, it is a goal and expected outcome of this project that we clarify and simplify the implementation.

If verification makes it necessary, even a complete reimplementation of the L4 X.2 API is possible.

Earlier work on operating system kernel formalisation and verification includes PSOS [11] and UCLA Secure Unix [15]. 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 [13] give an operational semantics for EROS and prove a confinement security policy. Our work differs in that we plan to formally relate our model to the implementation. Some case studies [7,5,14] appear in the literature in which the IPC and scheduling subsystems of microkernels have been described in PROMELA and verified 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. Finally, 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].

After introducing our notation in the following section, we first present an abstract conceptual model of virtual memory in L4 in section 3 and refine it towards an implementation in section 4.

2 Notation

Our meta-language Isabelle/HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t{::}\ \tau$ means that HOL term t has HOL type τ.

$$\textbf{datatype}\ 'a\ \mathit{option} = \mathit{None} \mid \mathit{Some}\ 'a$$

adjoins a new element *None* to a type $'a$. For succinctness we write $\lfloor a \rfloor$ instead of *Some a*.

Function update is written $f(x := y)$ where $f :: {'a} \Rightarrow {'b}$, $x :: {'a}$ and $y :: {'b}$.

Partial functions are modelled as functions of type $'a \Rightarrow {'b}\ \mathit{option}$, where *None* represents undefinedness and $f\, x = \lfloor y \rfloor$ means x is mapped to y. We call such functions *maps*, and abbreviate $f(x{:=}\lfloor y \rfloor)$ to $f(x \mapsto y)$. The map $\lambda x.\ \mathit{None}$ is written *empty*, and $\mathit{empty}(\ldots)$, where $\ldots$ are updates, abbreviates to $[\ldots]$. For example, $\mathit{empty}(x{\mapsto}y)$ becomes $[x \mapsto y]$.

Implication is denoted by $\Longrightarrow$ and $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$.

Finally, how are the formulae you see related to the formal Isabelle text? Our motto is

What you see is what we proved!

Isabelle theories can be augmented with LaTeX text which may contain references to Isabelle theorems (by name — see chapter 4 of [12]). We use this presentation mechanism to generate the text for most of the definitions and all of the theorems in this paper automatically.

3 Abstract Address Space Model

The virtual memory subsystem in L4 provides a flexible, hierarchical way of manipulating the mapping from virtual to physical memory pages of address spaces at user-level. We now present a formal model for this. Although the granularity at which L4 maps memory is the page level and does not go down to single addresses, we use the terms *address* and *page* interchangeably in the following.

3.1 Address Spaces

Fig. 2 illustrates the concept of hierarchical mappings. Large boxes depict virtual address spaces. The smaller boxes inside stand for virtual pages in the address space. The rounded box at the bottom is the set of physical pages. The arrows stand for direct mappings which connect pages in one address spaces to addresses in (possibly) other address spaces. In well-behaved states, the transitive closure of mappings always ends in physical pages. The example in Fig. 2 maps virtual page v_1 in space n_1, as well as v_2 in n_2, and v_4 in n_4 to the physical page r_1.

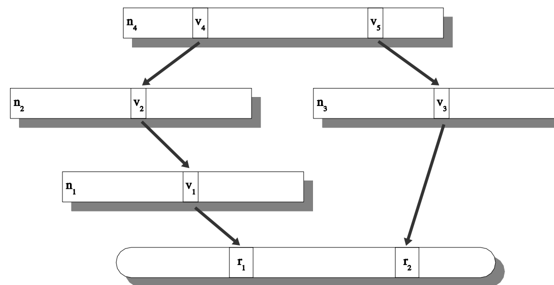

Fig. 2. Address Spaces

Formally, we use the types R for the physical pages (r_1, r_2, etc.), V for virtual pages (v_1, v_2, etc.), and N for the names of address spaces (n_1, n_2, etc.).

A position in this picture is determined uniquely by either naming a virtual page in a virtual address space, or by naming a physical page. We call these the mappings M:

$$\textbf{datatype } M = \mathit{Virtual}\ N\ V \mid \mathit{Real}\ R$$

An address space associates with each virtual page either a mapping, or nothing (the nil page). We implement this in Isabelle by the *option* datatype:

$$\textbf{types } \mathit{space} = V \Rightarrow M\ \mathit{option}$$

The machine state is then a map from address space names to address spaces. Not all names need to be associated with an address space, so we use *option* again:

$$\textbf{types } \mathit{state} = N \Rightarrow \mathit{space}\ \mathit{option}$$

To relate these functions to the arrows in Fig. 2, we use the concept of *paths*. The term $s \vdash x \leadsto^1 y$ means that in state s there is a direct path from position x to position y. There is a direct path from position $\mathit{Virtual}\ n\ v$ to another position y if in state s the address space with name n is defined and maps the virtual page v to y. There can be no paths starting at physical pages. Formally,

$$s \vdash x \leadsto^1 y = (\exists\, n\ v\ \sigma.\ x = \mathit{Virtual}\ n\ v \wedge s\ n = \lfloor \sigma \rfloor \wedge \sigma\ v = \lfloor y \rfloor)$$

We write $_ \vdash _ \leadsto^+ _$ for the transitive and $_ \vdash _ \leadsto^* _$ for the reflexive and transitive closure of the direct path relation.

3.2 Operations

The L4 kernel exports the following basic operations on address spaces: *unmap*, *flush*, *map*, and *grant*. The former two operations remove mappings, the latter two create or move mappings. We explain and define them below.

Fig. 3 illustrates the *unmap* $n\ v$ operation. It is the most fundamental of the operations above. We say a space n unmaps v if it removes all mappings that depend on $\mathit{Virtual}\ n\ v$, or in terms of paths if it removes all edges leading to $\mathit{Virtual}\ n\ v$.

To implement this, we use a function *clear* that, given name n, page v, and address space σ in a state s, returns σ with all v' leading to $\mathit{Virtual}\ n\ v$ mapped to None.

$\mathit{clear} :: N \Rightarrow V \Rightarrow \mathit{state} \Rightarrow \mathit{space} \Rightarrow \mathit{space}$
$\mathit{clear}\ n\ v\ s\ \sigma \equiv$
$\lambda v'.\ \mathsf{case}\ \sigma\ v'\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None}$
$\quad \mid \lfloor m \rfloor \Rightarrow \mathsf{if}\ s \vdash m \leadsto^* \mathit{Virtual}\ n\ v\ \mathsf{then}\ \mathit{None}\ \mathsf{else}\ \lfloor m \rfloor$

An *unmap* $n\ v$ in state s then produces a new state in which each address space is cleared of all paths leading to $\mathit{Virtual}\ n\ v$.

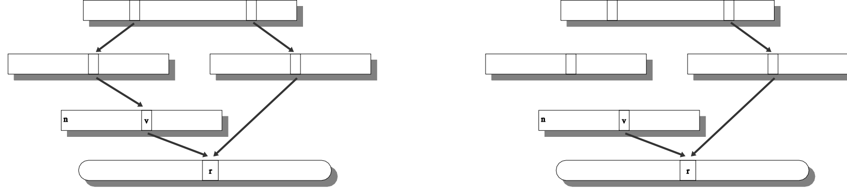

Fig. 3. The *unmap* operation (before and after)

$unmap :: N \Rightarrow V \Rightarrow state \Rightarrow state$
$unmap\ n\ v\ s \equiv \lambda n'.\ \textsf{case}\ s\ n'\ \textsf{of}\ None \Rightarrow None \mid \lfloor\sigma\rfloor \Rightarrow \lfloor clear\ n\ v\ s\ \sigma\rfloor$

For updating a space with name n at page v with a new mapping m we write $n,v \leftarrow m$, where m may be *None*.

$n,v \leftarrow m \equiv \lambda s.\ s(n := \textsf{case}\ s\ n\ \textsf{of}\ None \Rightarrow None \mid \lfloor\sigma\rfloor \Rightarrow \lfloor\sigma(v := m)\rfloor)$

With this, the flush operation is simply *unmap* followed by setting n,v to *None*.

$flush :: N \Rightarrow V \Rightarrow state \Rightarrow state$
$flush\ n\ v \equiv n,v \leftarrow None \circ unmap\ n\ v$

The remaining two operations *map* and *grant* establish new mappings in the receiving address space. To ensure a consistent new state, this new mapping must ultimately be connected to a physical page. We call a mapping m *valid* in state s (written $s \vdash m$) if it is a physical page, or if it is of the form $Virtual\ n\ v$ and is the source of some direct path. We show later that in all reachable states of the system, this definition is equivalent to saying that the mapping leads to a physical page.

$s \vdash m \equiv \textsf{case}\ m\ \textsf{of}\ Virtual\ n\ v \Rightarrow \exists x.\ s \vdash m \leadsto^1 x \mid Real\ r \Rightarrow True$

Before the kernel establishes a new value, the destination is always flushed. This may invalidate the source. The operation only continues if the source is still valid, otherwise it stops. We capture this behaviour in a slightly modified update notation $\leftharpoonup$:

$n,v \leftharpoonup m \equiv \lambda s.\ \textsf{let}\ s_0 = flush\ n\ v\ s\ \textsf{in}\ (\textsf{if}\ s_0 \vdash m\ \textsf{then}\ n,v \leftarrow \lfloor m\rfloor\ \textsf{else}\ id)\ s_0$

In L4, an address space n can *map* a page v to another space n' at page v'. Again, the operation only goes ahead, if the mapping $Virtual\ n\ v$ is valid:

$map :: N \Rightarrow V \Rightarrow N \Rightarrow V \Rightarrow state \Rightarrow state$
$map\ n\ v\ n'\ v'\ s \equiv \textsf{if}\ \neg\ s \vdash Virtual\ n\ v\ \textsf{then}\ s\ \textsf{else}\ (n',v' \leftharpoonup Virtual\ n\ v)\ s$

Fig. 4 shows an example for the *map* operation. Address space n maps page v to n' at v'. The destination n',v' is first flushed and then updated with the new mapping $Virtual\ n\ v$.

A space n can also *grant* a page v to v' in n'. As illustrated in Fig. 5, granting updates n',v' to the value of n at v and flushes the source n,v.

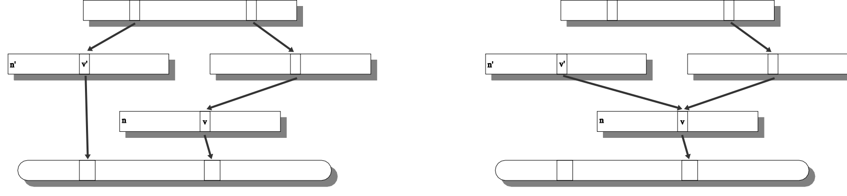

Fig. 4. The *map* operation (before and after)

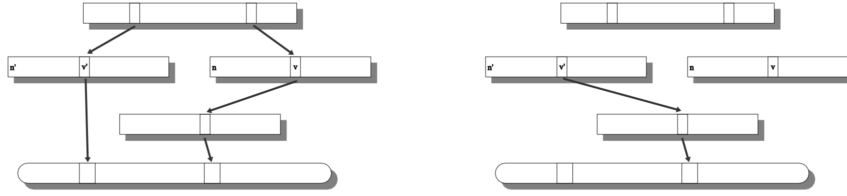

Fig. 5. The *grant* operation (before and after)

$grant :: N \Rightarrow V \Rightarrow N \Rightarrow V \Rightarrow state \Rightarrow state$
$grant\ n\ v\ n'\ v'\ s \equiv$
$\mathsf{if}\ \neg\ s \vdash Virtual\ n\ v\ \mathsf{then}\ s$
$\mathsf{else\ let}\ \lfloor\sigma\rfloor = s\ n;\ \lfloor m\rfloor = \sigma\ v\ \mathsf{in}\ (flush\ n\ v \circ n',v' \leftharpoondown m)\ s$

This concludes the kernel operations on address spaces. We have also modelled the hardware memory management unit (MMU). On this abstract level, all the MMU does is lookup: it determines which physical page needs to be accessed for each virtual page v and address space n. We write $s \vdash n\ v \rhd r$ if lookup of page v in the address space with name n in state s yields the physical page r. As we already have the concepts of paths, this is easily described formally:

$s \vdash n,v \rhd \lfloor r\rfloor = s \vdash Virtual\ n\ v \leadsto^{+} Real\ r$
$s \vdash n,v \rhd None = (\exists\sigma.\ s\ n = \lfloor\sigma\rfloor \wedge \sigma\ v = None) \vee s\ n = None$

The model in this section is based on an earlier pen-and-paper formalisation of L4 address spaces by Liedtke [10]. Formalising it in Isabelle/HOL eliminated problems like the mutual recursive definition of the update and flush functions being not well-founded. It would be well-founded—at least on reachable kernel states—if the model had the property that no loops can be constructed in address spaces. This is not true in the original model. The operation $map\ n\ v\ n'\ v'$ followed by $grant\ n'\ v'\ n\ v$ is a counter example. We also have introduced the formal concept of valid mappings to establish this no-loops property as well as the fact that any page that is mapped at all is mapped to a physical address.

3.3 An abstract data type for virtual memory

In the following we phrase the model of virtual memory and of the MMU hardware in terms of an abstract data type consisting of the type *state* and the operations detailed above. This data type (not to be confused with Isabelle's keyword **datatype**) is used implicitly by any user-level program. Even if the program does not invoke any mapping operations directly, the CPU performs a lookup operation with every memory access.

Putting the operations in terms of an abstract data type enables us to formulate refinement explicitly: if the data type of the abstract address spaces model is replaced with the data type of more concrete models (and finally the implementation) the program will not have any observable differences in behaviour.

Formally we define an abstract data type as a record consisting of an initial set of states and of a transition relation that models execution:

$$\begin{array}{l} \textbf{record}\ ('a,\ 'j)\ \mathit{DataType} = \\ \mathit{Init} :: 'a\ \mathit{set} \\ \mathit{Step} :: 'j \rightarrow ('a \times 'a)\ \mathit{set} \end{array}$$

For our virtual memory model, the operations are enumerated in the index type *VMIndex*:

$$\begin{array}{l} \textbf{datatype}\ \mathit{VMIndex} = \mathsf{create}\ N \mid \mathsf{unmap}\ N\ V \mid \mathsf{flush}\ N\ V \mid \mathsf{map}\ N\ V\ N\ V \\ \qquad \mid \mathsf{grant}\ N\ V\ N\ V \mid \mathsf{lookup}\ N\ V\ (R\ \mathit{option}) \end{array}$$

The definition of the abstract model $\mathcal{A}$ in terms of a data type is then:

$$\begin{array}{l} \mathit{Init}\ \mathcal{A} = \{[\sigma_0 \mapsto \sigma]\ |\sigma.\ \mathit{inj}_p\ \sigma \wedge \mathit{ran}\ \sigma \subseteq \mathit{range}\ \mathit{Real}\} \\ \mathit{Step}\ \mathcal{A}\ (\mathsf{lookup}\ n\ v\ r) = \{(s,\ s') \mid s = s' \wedge s \vdash n{,}v \rhd r\} \\ \mathit{Step}\ \mathcal{A}\ (\mathsf{create}\ n) = \{(s,\ s') \mid s\ n = \mathit{None} \wedge s' = s(n \mapsto \mathit{empty})\} \\ \mathit{Step}\ \mathcal{A}\ (\mathsf{unmap}\ n\ v) = \{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s' = \mathit{unmap}\ n\ v\ s\} \\ \mathit{Step}\ \mathcal{A}\ (\mathsf{flush}\ n\ v) = \{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s' = \mathit{flush}\ n\ v\ s\} \\ \mathit{Step}\ \mathcal{A}\ (\mathsf{map}\ n\ v\ n'\ v') = \{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s\ n' \neq \mathit{None} \wedge s' = \mathit{map}\ n\ v\ n'\ v'\ s\} \\ \mathit{Step}\ \mathcal{A}\ (\mathsf{grant}\ n\ v\ n'\ v') = \\ \{(s,\ s') \mid s\ n \neq \mathit{None} \wedge s\ n' \neq \mathit{None} \wedge s' = \mathit{grant}\ n\ v\ n'\ v'\ s\} \end{array}$$

The boot process creates an address space σ_0 that is an injective mapping from virtual to physical pages. The functions *ran* and *range* return the codomain of a function, where *ran* works on functions $'a \Rightarrow 'b\ \mathit{option}$ and *range* on total functions. Injectivity is constrained to the part of the function that returns $\lfloor x \rfloor$: $\mathit{inj}_p\ f \equiv \mathit{inj\text{-}on}\ f\ \{x \mid \exists\, y.\ f\, x = \lfloor y \rfloor\}$.

The lookup operation is special. In the context of a real system this operation would return a value, since one of the points of the virtual memory abstraction is to provide address translation. If a lookup yields a *None* result the kernel typically raises a *page fault* exception. Since we do not model the larger system, we simplify lookup instead to a subset of the identity relation on *state*.

Creating a new address space n is modelled by updating the state s at n with the predefined map *empty*. The other mapping operations have been defined above. All of them require the address spaces they operate on to be valid. This condition is ensured automatically in the current L4 implementation as the address spaces are determined by sender and receiver of an IPC operation.

3.4 Properties

We have shown a number of safety properties about the abstract address space model. They are formulated as invariants over the abstract datatype. A set of states I is an invariant if it contains all initial states and if execution of any operation in a state of I again leads to a state in I. We write $\mathcal{D} \models I$ when I is an invariant of data type $\mathcal{D}$.

Theorem 1. *There are no loops in the address space structure.*

$$\mathcal{A} \models \{s \mid \forall x.\ \neg\ s \vdash x \leadsto^{+} x\}$$

The proof is by case distinction on the operations and proceeds by observing how each operation changes existing paths. Theorem 1 is significant for implementing the lookup function efficiently. It also ensures that internal kernel functions can walk the corresponding data structures naively. Together with the properties below it says that address spaces always have a tree structure.

Theorem 2. *All valid pages translate to physical pages.*

$$\mathcal{A} \models \{s \mid \forall x.\ s \vdash x \longrightarrow (\exists r.\ s \vdash x \leadsto^{*} \mathit{Real}\ r)\}$$

The proof is again by case distinction on the operations. Together with the following theorem we obtain that address lookup is a total function on data type $\mathcal{A}$.

Theorem 3. *The lookup relation is a function.*

$$[\![s \vdash n{,}v \rhd r;\ s \vdash n{,}v \rhd r']\!] \Longrightarrow r = r'$$

This theorem follows directly from the fact that paths are built on functions.

That address lookup is a total function may sound like merely a nice formal property, but it is quite literally an important safety property in reality. Undefined behaviour, possibly physical damage, may result if two conflicting TLB entries are present for the same virtual address. The current ARM reference manual [2, p. B3-26] explicitly warns against this scenario.

3.5 Simplifications and Assumptions

The current model makes the following simplifications and assumptions.

- The L4Ka::Pistachio API stipulates two regions per address space that are shared between the user and kernel, the *kernel interface page* (KIP) and *user thread control blocks* (UTCBs). These should have a valid translation from virtual to physical memory pages, but can not be manipulated by the mapping operations.
- The mapping operations in L4 work on regions of the address space rather than individual pages. These regions, known as *flexpages*, are $2^k b, k \geq 0$ aligned and sized where b is the minimum page size on the architecture. This introduces significant complexity in the implementation and has a number of

boundary conditions of interest, so adding this to the abstract model would be beneficial. At the same time, it is possible to create systems using L4 that only use the minimum flexpage size so this omission does not pose a serious limitation to the utility of the model.

- *map* and *grant* are implemented through the IPC primitives in L4 and involve an agreement on the region to be transferred between sender and receiver. This can be added when the IPC abstraction is modelled.
- Flexpages also have associated read, write and execute access rights. At present the model can be considered as providing an all or nothing view of access rights.
- We assume that all of the mapping operations are atomic, which is the case in the current non-preemptable implementation, and a single processor, hence a sequential system.

4 Model Refinement

The model in the previous section provides an abstract model of address spaces in L4 but does not bear much resemblance to the kernel implementation. This is not surprising since the kernel must provide an efficient realisation of the mapping operations and the code supporting this executes under time and space restrictions. For the purpose of source-code verification it is desirable to have a more concrete model of the implementation. This model will be more complex and detailed than the previous model and hence less suited to proving properties such as the absence of loops in paths. By showing the concrete model to be a refinement of the abstract model it is possible to retain the ability to reason and prove properties at the abstract level. In this section we provide a motivation and overview of the implementation in the L4Ka::Pistachio kernel of address spaces, and then describe the refinement of the abstract address spaces model.

4.1 L4Ka::Pistachio Implementation

The implementation of address spaces is provided by the hardware and OS virtual memory mechanisms. The lookup relation corresponds to the virtual-to-physical mapping function provided by the MMU on the CPU. This translation is carried out on every memory access and so is critical to system performance. This is typically hidden in the pipeline by an associative cache, called the *translation-lookaside buffer* (TLB), holding a subset of mappings from the *page table* data structure which is located in memory. On a TLB miss the page table is accessed to perform address translation by a hardware mechanism (on the ARM architecture) that walks the page table data structure. The page table must support fast address translation, since TLB misses are frequent enough to warrant this, but this must be balanced with space considerations. In L4Ka::Pistachio a multi-level hierarchical page table is implemented, of which the ARM hardware

defined page table format, a two-level page table, is an instance. The operations that update mappings must also maintain coherence between the TLB and page table, and also the data and instruction caches and memory on ARM since the caches are virtually-addressed.

In addition to the virtual-to-physical mappings, an implementation of L4 address spaces requires a representation of the mappings between address spaces, the *mapping database* (MDB). This is conceptually quite similar to the abstract model, with paths reversed to give a tree rooted at each physical memory page. The *map*, *grant* and *unmap* operations correspond to system calls and execute with a small, fixed-size kernel stack. Hence it is desirable to avoid recursion. This is achieved in L4Ka::Pistachio by implementing the mapping tree with a linked-list representing the preorder traversal of the tree, augmented with depth information. The list is doubly-linked and there are pointers stored between nodes in the mapping database and nodes associated with the corresponding page table nodes to avoid unnecessary traversals of either data structure in the mapping operations.

4.2 Tree Address Space Model

We first show that a model of address spaces with the mapping database as a forest to be a refinement of the model in Section 3. This is a conceptual step. It is the view that most people working with the kernel implementation adopt.

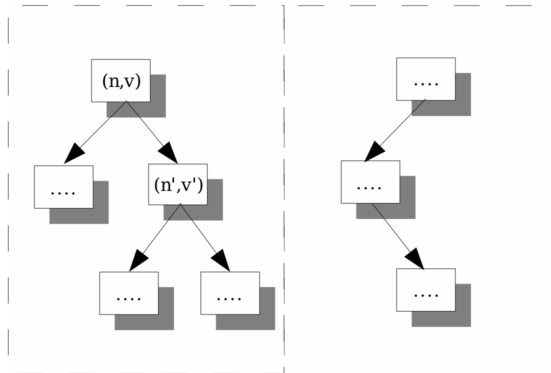

Fig. 6. Forest

$$\textbf{types } MDB = (N \times V) \rightharpoonup (N \times V)\ set$$

A tree here is a partial function from a node to a set of child nodes (see Fig. 6). The function is required to be partial so that nodes with no children and nodes not present in the tree can be distinguished.

$$\textbf{record } state_1 = \\ N :: N\ set \\ M :: R \Rightarrow MDB$$

The N component of the state now contains the names of the valid address spaces and each physical memory page has an associated mapping tree (possibly empty) in the M component of the state.

The direct path relation is defined as

$$s \vdash a \rightsquigarrow^1{}_1\ b = (\exists\, r\ mn.\ M\ s\ r\ a = \lfloor mn \rfloor \wedge b \in mn)$$

A direct path exists between nodes a and b if b is a child of a in a tree r.

Again, we write $_ \vdash _ \rightsquigarrow^+{}_1\ _$ for the transitive and $_ \vdash _ \rightsquigarrow^*{}_1\ _$ for the reflexive and transitive closure of the direct path relation. A path between a and b indicates that b is in the subtree of a.

Lookup in the tree model is written as $s \vdash n{,}v \rhd_1\ r$ and is defined with:

$$s \vdash n{,}v \rhd_1 \lfloor r \rfloor = (M\ s\ r\ (n,\ v) \neq None)$$
$$s \vdash n{,}v \rhd_1\ None = ((\forall\, r.\ M\ s\ r\ (n,\ v) = None) \wedge n \in N\ s \vee n \notin N\ s)$$

Lookup corresponds to tree membership for a node.

The $unmap_1$ operation then simply removes all nodes in the subtree of the target from the tree, except the target, and all references to these nodes from other nodes. The notation $s(\!|M := x|\!)$ denotes update of field M in record s with value x.

$$\begin{aligned}
&unmap_1\ n\ v\ s \equiv \\
&s(\!|M := \lambda r\ x.\ \textsf{case}\ M\ s\ r\ x\ \textsf{of}\ None \Rightarrow None \\
&\qquad\qquad\quad \mid \lfloor mn \rfloor \Rightarrow \\
&\qquad\qquad\qquad \textsf{if}\ s \vdash (n,\ v) \rightsquigarrow^+{}_1\ x\ \textsf{then}\ None \\
&\qquad\qquad\qquad \textsf{else}\ \lfloor \{b \mid b \in mn \wedge \neg\ s \vdash (n,\ v) \rightsquigarrow^+{}_1\ b\} \rfloor |\!)
\end{aligned}$$

Similarly, $flush_1$ removes all nodes in the subtree along with their corresponding references.

$$\begin{aligned}
&flush_1\ n\ v\ s \equiv \\
&s(\!|M := \lambda r\ x.\ \textsf{case}\ M\ s\ r\ x\ \textsf{of}\ None \Rightarrow None \\
&\qquad\qquad\quad \mid \lfloor mn \rfloor \Rightarrow \\
&\qquad\qquad\qquad \textsf{if}\ s \vdash (n,\ v) \rightsquigarrow^*{}_1\ x\ \textsf{then}\ None \\
&\qquad\qquad\qquad \textsf{else}\ \lfloor \{b \mid b \in mn \wedge \neg\ s \vdash (n,\ v) \rightsquigarrow^*{}_1\ b\} \rfloor |\!)
\end{aligned}$$

map_1 is implemented by inserting a new node for the map destination in the tree beneath the map source.

$$\begin{aligned}
&map_1\ n\ v\ n'\ v'\ s \equiv \\
&\textsf{if}\ s \vdash n{,}v \rhd_1\ None\ \textsf{then}\ s \\
&\textsf{else let}\ s' = flush_1\ n'\ v'\ s \\
&\quad \textsf{in if}\ s' \vdash n{,}v \rhd_1\ None\ \textsf{then}\ s'\ \textsf{else}\ update\text{-}map_1\ n\ v\ n'\ v'\ s'
\end{aligned}$$

$$\begin{aligned}
&update\text{-}map_1\ n\ v\ n'\ v'\ s \equiv \\
&s(\!|M := \lambda r.\ \textsf{case}\ M\ s\ r\ (n,\ v)\ \textsf{of}\ None \Rightarrow M\ s\ r \\
&\qquad\qquad \mid \lfloor mn \rfloor \Rightarrow M\ s\ r((n,\ v) \mapsto mn \cup \{(n',\ v')\},\ (n',\ v') \mapsto \{\})|\!)
\end{aligned}$$

In $grant_1$ a node is inserted into the tree for the destination if the prior flush and unmap do not result in the source being removed, and any references to the source are replaced by references to the destination node.

$grant_1\ n\ v\ n'\ v'\ s \equiv$
$\mathsf{if}\ s \vdash n{,}v \rhd_1 None\ \mathsf{then}\ s$
$\mathsf{else\ let}\ s' = flush_1\ n'\ v'\ s$
$\quad \mathsf{in\ if}\ s' \vdash n{,}v \rhd_1 None\ \mathsf{then}\ s'\ \mathsf{else}\ update\text{-}grant_1\ n\ v\ n'\ v'\ s'$

$update\text{-}grant_1\ n\ v\ n'\ v'\ s \equiv$
$\mathsf{let}\ s' = unmap_1\ n\ v\ s$
$\mathsf{in}\ s'(\!|M := \lambda r\ x.\ \mathsf{if}\ x = (n',\ v') \wedge s' \vdash n{,}v \rhd_1 \lfloor r \rfloor\ \mathsf{then}\ \lfloor\{\}\rfloor$
$\quad\quad \mathsf{else\ case}\ M\ s'\ r\ x\ \mathsf{of}\ None \Rightarrow None$
$\quad\quad\quad \mid \lfloor mn \rfloor \Rightarrow$
$\quad\quad\quad\quad \mathsf{if}\ x = (n,\ v)\ \mathsf{then}\ None$
$\quad\quad\quad\quad \mathsf{else}\ \lfloor\{b \mid b \in mn \wedge b \neq (n,\ v) \vee$
$\quad\quad\quad\quad\quad (n,\ v) \in mn \wedge b = (n',\ v')\}\rfloor|\!)$

4.3 Refinement Proof

In this section we again phrase the model presented above in terms of a data type. The tree data type $\mathcal{M}$ is:

$Init\ \mathcal{M} =$
$\{(\!|N = \{\sigma_0\},\ M = \lambda r\ nv.\ \mathsf{if}\ P'\ nv = \lfloor r \rfloor\ \mathsf{then}\ \lfloor\{\}\rfloor\ \mathsf{else}\ None|\!)\ |P'.$
$\ inj_p\ P' \wedge fst\ `\ dom\ P' \subseteq \{\sigma_0\}\}$
$Step\ \mathcal{M}\ (\mathsf{lookup}\ n\ v\ r) = \{(s,\ s') \mid s = s' \wedge s \vdash n{,}v \rhd_1 r\}$
$Step\ \mathcal{M}\ (\mathsf{create}\ n) = \{(s,\ s') \mid n \notin N\ s \wedge s' = s(\!|N := insert\ n\ (N\ s)|\!)\}$
$Step\ \mathcal{M}\ (\mathsf{unmap}\ n\ v) = \{(s,\ s') \mid n \in N\ s \wedge s' = unmap_1\ n\ v\ s\}$
$Step\ \mathcal{M}\ (\mathsf{flush}\ n\ v) = \{(s,\ s') \mid n \in N\ s \wedge s' = flush_1\ n\ v\ s\}$
$Step\ \mathcal{M}\ (\mathsf{map}\ n\ v\ n'\ v') = \{(s,\ s') \mid n \in N\ s \wedge n' \in N\ s \wedge s' = map_1\ n\ v\ n'\ v'\ s\}$
$Step\ \mathcal{M}\ (\mathsf{grant}\ n\ v\ n'\ v') = \{(s,\ s') \mid n \in N\ s \wedge n' \in N\ s \wedge s' = grant_1\ n\ v\ n'\ v'\ s\}$

We show that the tree data type is a refinement of the abstract data type. Here refinement is taken to mean *data refinement* [6] and we use the proof technique of simulation.

We begin with the abstraction relation R_1 between concrete state s_c and abstract states s_a:

$R_1 \equiv$
$\{(s_c,\ s_a) \mid dom\ s_a = N\ s_c \wedge$
$\quad (\forall n\ v\ r.\ s_a \vdash n{,}v \rhd \lfloor r \rfloor = s_c \vdash n{,}v \rhd_1 \lfloor r \rfloor) \wedge$
$\quad (\forall n\ v\ n'\ v'.\ s_a \vdash Virtual\ n\ v \leadsto^1 Virtual\ n'\ v' = s_c \vdash (n',\ v') \leadsto^1{}_1\ (n,\ v))\}$

Here it is clear that the path relation in the tree model is the inverse of the path relation in the abstract model.

We then show that the diagrams in Fig. 7 commute, for all operations. This is achieved by showing forward simulation:

$$C \leq_F A \equiv \exists r.\ Init\ C \subseteq r\ ``\ Init\ A \wedge (\forall j.\ r\ ;;\ Step\ C\ j \subseteq Step\ A\ j\ ;;\ r)$$

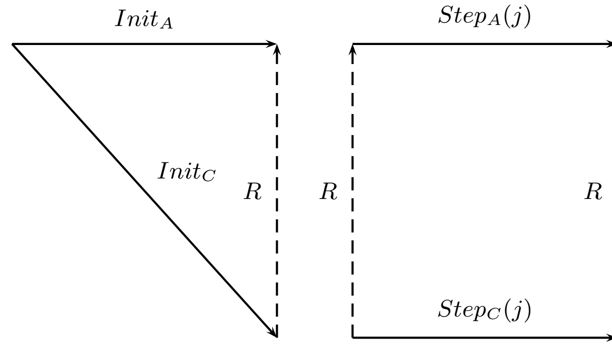

Fig. 7. Simulation

where ‘‘ is the image of a set under a relation, and ;; the composition of two relations.

Theorem 4. *The tree data type simulates the abstract data type*

$$\mathcal{M} \leq_F \mathcal{A}$$

The proof is by case distinction on the operations of the data type. It proceeds by observing how each operation changes the state in terms of the path and lookup relations on the concrete and abstract level. For example, the direct path relation after flush can be shown to be:

$$\mathit{flush}\ n\ v\ s \vdash x \leadsto^1 y = (s \vdash x \leadsto^1 y \wedge \neg\ s \vdash x \leadsto^* \mathit{Virtual}\ n\ v)$$

$$\mathit{flush}_1\ n\ v\ s \vdash x \leadsto^1{}_1\ y = (s \vdash x \leadsto^1{}_1\ y \wedge \neg\ s \vdash (n,\ v) \leadsto^*{}_1\ y)$$

Simulation gives that the properties proved as invariants on the abstract data type also hold on the concrete data type, i.e. the safety properties proved in Section 3.4 also hold on the concrete data type.

Also, since the operations are deterministic, the simulation also holds in the other direction.

Theorem 5. *The abstract data type simulates the tree data type*

$$\mathcal{A} \leq_F \mathcal{M}$$

4.4 Further Refinement

The next step in the refinement process is to implement the forest with a list model. The state space for this is based on the following type:

record *TreeListNode* =
Next :: *TreeListNodeName option*
Prev :: *TreeListNodeName option*
PTE :: *PTEName*
Depth :: *nat*

$$\textbf{record } \mathit{TreeListHeap} = \\ \mathit{Valid} :: \mathit{TreeListNodeName\ set} \\ \mathit{Heap} :: \mathit{TreeListNodeName} \Rightarrow \mathit{TreeListNode}$$

where *TreeListNodeName* and *PTEName* are uninterpreted types. These represent pointers to list nodes and page table entries respectively.

The mapping operations in this model are closer to those in the implementation. Unmap/flush iterate over the subtree unlinking nodes, map inserts a node into the list immediately after the destination node and grant replaces the source node with that of the destination in the list.

The following subtree relation can be used to connect the list to the tree model.

$$s \vdash x \mapsto y = (\mathit{Next}\ (\mathit{Heap}\ s\ x) = \lfloor y \rfloor \wedge x \in \mathit{Valid}\ s)$$

$$[\![s \vdash m \mapsto m';\ \mathit{Depth}\ (\mathit{Heap}\ s\ m) < \mathit{Depth}\ (\mathit{Heap}\ s\ m')]\!] \Longrightarrow s \vdash m \leadsto^T m'$$

$$[\![s \vdash m \leadsto^T m';\ s \vdash m' \mapsto ma;\ \mathit{Depth}\ (\mathit{Heap}\ s\ m) < \mathit{Depth}\ (\mathit{Heap}\ s\ ma)]\!] \\ \Longrightarrow s \vdash m \leadsto^T ma$$

The refinement relation then implies the equivalence of subtrees in the models. We omit the page table and operations here, a complete description of this refinement step will be published in later work.

Further refinement will proceed by independent refinement of the list heap and page table to source level. There are a number of issues to address in this process, including a choice of suitable language for use in the refinement steps once we decompose operations into imperative code.

5 Conclusion

We have presented the initial stage of a refinement process to verify the virtual memory subsystem of the L4 microkernel. We have shown an abstract model of address spaces together with the operations on them that the kernel API offers. We have refined it into a tree-like structure that is conceptually closer to the data structures used in the kernel implementation.

The next step after refining the current stage into a linked list structure and a page table implementation will be source code verification. Even though we have not yet reached the implementation level, the process of building an abstract model and refining it has already had a beneficial impact on the L4 kernel. During the process of developing these models we have encountered and clarified a number of small ambiguities and errors in the reference manual, have identified unnecessary restrictions, and discovered small errors in the implementation.

Our activities in verifying the L4 kernel apart from the memory subsystem include building a complete abstract model of the L4 API that is executable and lends itself to simulation and exploration. We are also looking at how further

safety and security properties like confidentiality and information flow are best formulated in the context of the L4 model we are building.

Acknowledgements We thank Kai Engelhardt, Kevin Elphinstone, Michael Norrish, Adam Wiggins, and the developers on the pistachio-core mailing list for advice and stimulating discussions.

References

1. *L4 eXperimental Kernel Reference Manual Version X.2*, 2004.
2. ARM Limited. *ARM Architecture Reference Manual*, June 2000.
3. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
4. William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., December 1994.
5. Thierry Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.
6. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
7. Gregory Duval and Jacques Julliand. Modelling and verification of the RUBIS μ-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.
8. System Architecture Group. The L4Ka::Pistachio microkernel. White paper, University of Karlsruhe, May 2003.
9. Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.
10. J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.
11. 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, Computer Science Laboratory, SRI International, 1980.
12. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. `http://www.in.tum.de/~nipkow/LNCS2283/`.
13. J. S. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS-97-26, Distributed Systems Laboratory, University of Pennsylvania, 1997.
14. P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.
15. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.