# Verifying the L4 Virtual Memory Subsystem

Harvey Tuch and Gerwin Klein

National ICT Australia*, Sydney, Australia
{gerwin.klein|harvey.tuch}@nicta.com.au

**Abstract.** We describe aspects of the formalisation and verification of the L4 $\mu$-kernel. Starting from an abstract model of the virtual memory subsystem in L4, we prove safety properties about this model, and then refine the page table abstraction, one part of the model, towards C source code. 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 [12]. 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.

The design of L4 is not only geared towards flexibility and reliability, but also 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

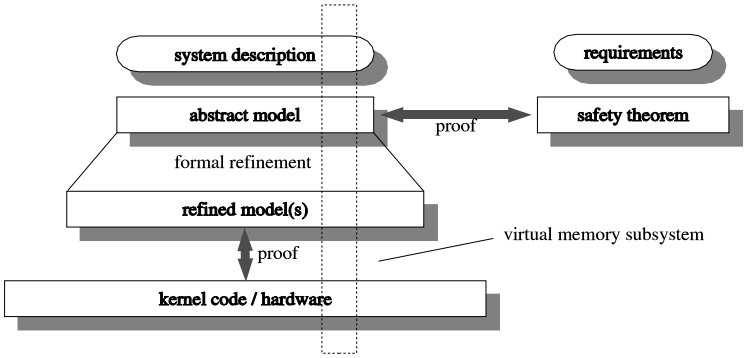


**Fig. 1.** Overview

the L4 reference manual [10]. 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.

In this paper we give an overview of some of the steps in this refinement process. L4 provides three main abstractions: threads, address spaces, and interprocess 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 first show an abstract model of address spaces, describe the framework in which the refinement process proceeds and then concentrate on the implementation of one particular operation of the abstract model. This operation is implemented in the Kernel using page tables of which we again first show an abstract view and then provide an implementation of some of its operations in a programming language that in its level of abstraction is close to C.

Earlier work on operating system kernel formalisation and verification includes PSOS [15] and UCLA Secure Unix [20]. 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 [2], 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 [3] specify legal Mach states and describe Mach system calls using temporal logic. Shapiro and Weber [17] 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 [6, 4, 19] 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 [7].

After introducing our notation in the following section, we first present the abstract conceptual model of virtual memory in L4 in section 3, and then show parts of the refinement of the memory lookup operation in this model towards a page table 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 $\tau$.

The cons of an element $x$ to a list $xs$ is written $x \# xs$, and $[]$ is the empty list. Pairs come with the two projection functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. We identify tuples with pairs nested to the right: $(a, b, c)$ is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ is identical to $'a \times ('b \times 'c)$.

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

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 $'a \Rightarrow 'b\ option$, 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 $\lambda x.\ None$ is written *empty*, and $empty(\ldots)$, where $\ldots$ are updates, abbreviates to $[\ldots]$. For example, $empty(x \mapsto y)$ becomes $[x \mapsto y]$.

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

Records in Isabelle [14], as familiar from programming languages, are essentially tuples with named fields. The type declaration

$$\begin{array}{l}\textbf{record}\ point = \\ \quad X :: nat \\ \quad Y :: nat\end{array}$$

creates a new record type *point* with two components $X$ and $Y$ of type *nat*. The notation $(\!|X{=}0,\ Y{=}0|\!)$ stands for the element of type *point* that has both components set to 0. Isabelle automatically creates two selector functions $X :: point \Rightarrow nat$ and $Y :: point \Rightarrow nat$ such that, e.g. $X\ (\!|X{=}0,\ Y{=}0|\!) = 0$. Updating field $Y$ of a record $p$ with value $n$ is written $p\ (\!|\,Y := n|\!)$. As for function update, multiple record updates separated by comma are admitted.

## 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 address spaces. A first description of this model has already appeared in [9]. For completeness, we repeat parts of it in sections 3.1 and 3.2. The treatment of abstract datatypes in section 3.3 is updated to incorporate operations with output.

### 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 $r_1$.

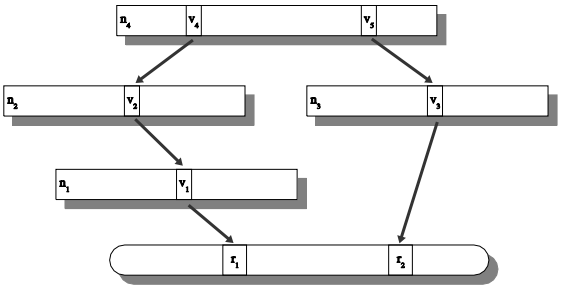


**Fig. 2.** Address Spaces

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 $s \vdash x \rightsquigarrow^1 y$ 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,

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

We write $\_ \vdash \_ \rightsquigarrow^+ \_$ for the transitive and $\_ \vdash \_ \rightsquigarrow^* \_$ 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 $\mathit{Virtual}\ n\ v$.

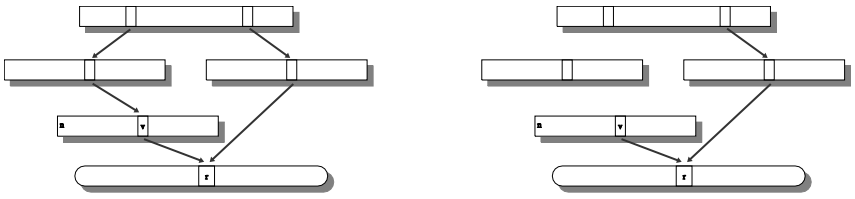

**Fig. 3.** The *unmap* operation (before and after)

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

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

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

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

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*.

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

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.

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

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 $\leftharpoonup$:

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

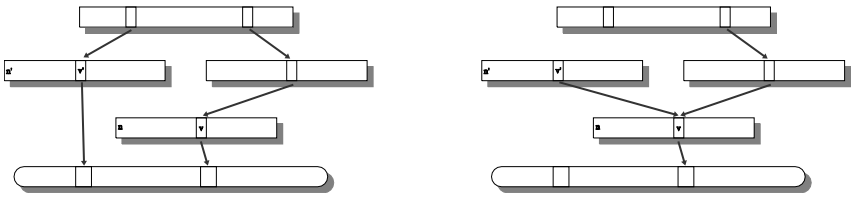

**Fig. 4.** The *map* operation (before and after)

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:

$map :: N \Rightarrow V \Rightarrow N \Rightarrow V \Rightarrow state \Rightarrow state$
$map\ n\ v\ n'\ v'\ s \equiv$ *if* $\neg\ s \vdash Virtual\ n\ v$ *then* $s$ *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 $n,v$.

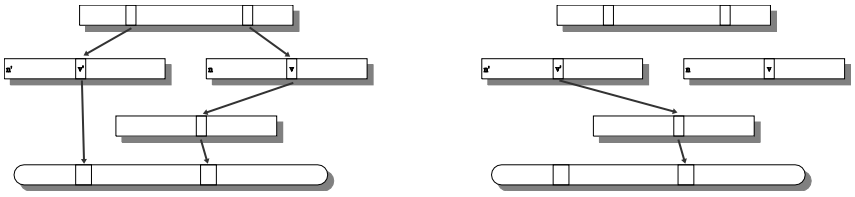

**Fig. 5.** The *grant* operation (before and after)

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

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 $s \vdash n,v \rhd \lfloor r\rfloor$ 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:

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

The model in this section is based on an earlier pen-and-paper formalisation of L4 address spaces by Liedtke [12]. 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 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 with return values of type $'o$:

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

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

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

The abstract model $\mathcal{A}$ in terms of a (*state*, *VMIndex*, *R option*) *DataType* is then:

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

The boot process creates an address space $\sigma_0$ 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 the only operation that returns a value. All other operations return *None*.

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 exist. This condition is ensured automatically in the current L4 implementation as the address spaces are determined by sender and receiver of an IPC operation.

The correctness of the implementation with respect to the abstraction is established by showing the concrete model to be a refinement of the abstract

model. Here refinement is taken to mean *data refinement* [5] and we use the proof technique of simulation. Simulation between an abstract $('a, 'j, 'o)$ *DataType* and a concrete $('c, 'j, 'o)$ *DataType* is formalized as follows.

The step relations for each operation are of type $('a \times 'a \times 'o)$ *set*. It is convenient to have a relation for these operations of type $('a \times 'o) \times ('a \times 'o)$ below, so we introduce the function *up*. This gives the semantics of the operations on the state space $'a \times 'o$. Since the value of the $'o$ component in the pre-state has no effect on the semantics of the operations in an ADT it can be left unrestricted.

$$up\ r \equiv \{((a,\ i),\ b,\ k) \mid (a,\ b,\ k) \in r\}$$

The type of the abstraction relation $r$ is $('a \times 'c)$ *set*. $id_o$ lifts this to $('a \times 'o) \times ('c \times 'o)$.

$$id_o\ r \equiv \{((s,\ k),\ s',\ k') \mid k = k' \wedge (s,\ s') \in r\}$$

A relation $c$ is an L-subset of a relation $a$ under the relation $r$ if the following holds, where $a\ ;\ b$ is relational compososition of $a$ and $b$.

$$r \vdash c \subseteq^L a \equiv r\ ;\ c \subseteq a\ ;\ r$$

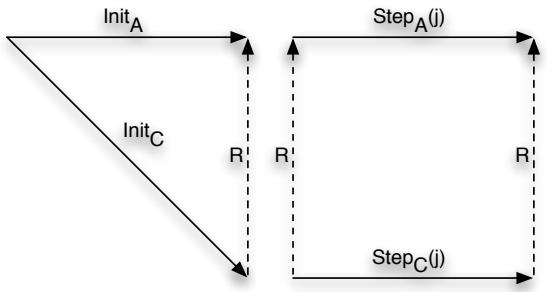


**Fig. 6.** Simulation

A forward simulation exists if the diagrams in Figure 6 commute. That is, there exists a relation such that the initial states of the concrete model are a subset of those in the abstract model under the relational image of the lifted abstraction relation, and if for each step operation the concrete step is an L-subset of the abstract step relation under $r$.

$Lr\ r\ C\ A \equiv$
*let* $r_o = id_o\ r$
*in* $Init\ C \times UNIV \subseteq r_o\ ``\ (Init\ A \times UNIV) \wedge$
$(\forall j.\ r_o \vdash up\ (Step\ C\ j) \subseteq^L up\ (Step\ A\ j))$

We write $C \leq_F A$ when concrete data type $C$ simulates abstract data type $A$:

$$C \leq_F A \equiv \exists r.\ Lr\ r\ C\ A$$

### 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.*

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

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.*

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

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 [1, 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 Page Tables

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.

Below we consider the refinement of one component of the virtual memory subsystem necessary for the implementation of address spaces. The models and interfaces below are based on the existing page table implementation in the L4Ka::Pistachio [11] kernel.

### 4.1 Abstract model

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. It is typically hidden in the processor pipeline by an associative cache, called the *translation-lookaside buffer* (TLB). This holds a subset of mappings from the *page table* data structure which is located in memory. The TLB caches *page table entries* (PTEs), as in Figure 7 — a PTE for a page in the virtual address space specifies the corresponding physical page, access rights, and other page specific information, shown in Figure 8. On a TLB miss a hardware mechanism[1] traverses the page table data structure to perform address translation.

[1] On the ARM architecture. Other architectures might also rely on software mechanisms to achieve the same goal.

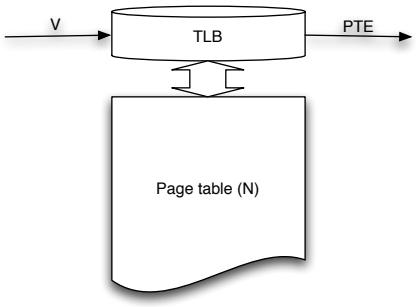


**Fig. 7.** PTE lookup through the TLB

The design of page table implementations is influenced by the direct and indirect performance costs of this operation.

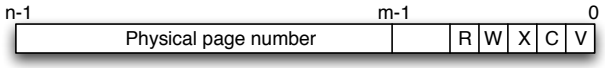


**Fig. 8.** Page table entry (PTE)

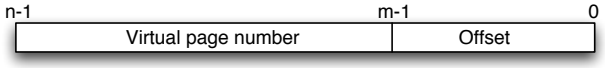


**Fig. 9.** Virtual address

While we treated virtual page numbers and virtual addresses interchangeably in the previous section, this will no longer be sufficient when considering the specifics of page table implementations, since modern TLBs usually support multiple page sizes, called *superpages* [18], in order to improve the coverage of the TLB; a single PTE (and TLB entry) can then cover large regions of the address space. Hence many virtual pages may be associated with a single virtual address. An $n$-bit virtual addresses can be considered as consisting of an $(n-m)$-bit virtual page number and an $m$-bit offset, as in Figure 9, where $2^m$ is in the set of page sizes supported by the the architecture. Mappings are then from $2^m$

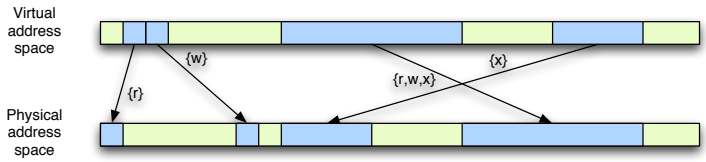


**Fig. 10.** Mappings from virtual to physical pages

sized, aligned regions of the virtual address space to the physical address space, shown in Figure 10.

Virtual addresses are modelled using a theory of fixed-width words in Isabelle, where the word type is a quotient type with equivalence classes derived by taking the natural numbers modulo the word size. The theory is imported from the HOL4 system. We use the *word*32 type for virtual addresses, although nothing should depend on this particular value for word size and the model and proofs presented here should be the same for any size of virtual addresses.

$$\textbf{types}\ V = \mathit{word}32$$

The function *page-bits ps* gives the value of $m$ for page size *ps*. We introduce the type *PTESize* of which values are supported (super)page sizes.

$$\textbf{consts}\ \mathit{page\text{-}bits} :: \mathit{PTESize} \Rightarrow \mathit{nat}$$

The *vpn* for a virtual address is its virtual page number as a *nat*. The function $w2n$ converts from a value of type *word*32 to the corresponding *nat*. $n2w$ does the reverse.

$$\mathit{vpn}\ v\ \mathit{ps} \equiv w2n\ v\ \mathit{div}\ 2\ \hat{}\ \mathit{page\text{-}bits}\ \mathit{ps}$$

At a given page size, two virtual addresses with identical virtual page numbers are of the same page.

$$\mathit{page\text{-}equiv}\ l\ v\ v' \equiv \mathit{vpn}\ v\ l = \mathit{vpn}\ v'\ l$$

*page-set v ps* gives the set of virtual addresses for the page of size *ps* containing the virtual address $v$. Figure 11 gives two example *page-set*s for a virtual address $v$ with superpage sizes $m_0$ and $m_1$.

$$\mathit{page\text{-}set}\ v\ l \equiv \{v' \mid \mathit{page\text{-}equiv}\ l\ v\ v'\}$$

We begin with the description of the state space for the abstract model by introducing several new types. We model page tables as function from $N \times V$ to a pointer to a PTE stored in a heap. We choose to model explicitly at the abstract

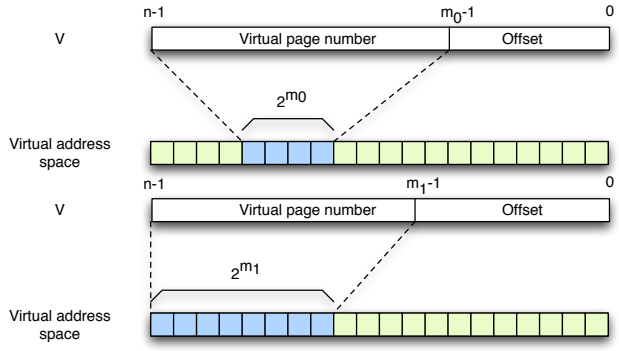


**Fig. 11.** Example *page-set*s

level indirection with respect to PTEs, based on the interface observed in the L4Ka::Pistachio linear page table implementation. This allows for efficient implementation of operations that modify PTEs since unnecessary traversal of the page table can be avoided. The type of PTE pointers is *PTEName*. In addition, a pointer type *TreeListNodeName* is introduced. The page table ADT is utilised by the mapping database (MDB) that stores the map/grant relationships between address spaces as illustrated in section 3.1. In addition to the fields that are usually present in a PTE, the MDB requires that each PTE has the corresponding virtual address and a pointer of type *TreeListNodeName* associated with it.

An abstract PTE is modelled as a record type. *Paddr* contains the physical page number for the page, $R, W, X$ specify the access rights for this mapping, and *Cached* indicates whether the data accessed through this mapping may be stored in the data or instruction caches. As part of the MDB required interface we also conceptually associate the two additional fields *MapNode* and *Vaddr* with the PTE as explained above.

$$
\begin{array}{l}
\textbf{record}\ PTE_a = \\
\quad Paddr :: R \\
\quad R :: bool \\
\quad W :: bool \\
\quad X :: bool \\
\quad Cached :: bool \\
\quad MapNode :: TreeListNodeName \\
\quad Vaddr :: V
\end{array}
$$

The state space is then a partial function from $N \times V$ to the PTE pointer for the mapping and the size of the mapping, and a heap for PTEs. In addition, the $N$ field of *PTState* stores which address spaces are currently active (have been created).

**types** $PageTable = N \times V \Rightarrow (PTEName \times PTESize)\ option$
**types** $PTEHeap = PTEName \Rightarrow PTE_a$

**record** $PTState =$
$N :: N\ list$
$Heap :: PTEHeap$
$PageTable :: PageTable$

The operations provided by the page table ADT and their return types are enumerated in the following two type declarations.

**datatype** $PTIndex =$ insert $N\ V\ PTESize$
| lookup $N\ V$
| getpaddr $PTEName\ PTESize$
| setpaddr $PTEName\ R\ PTESize$
| setlinknode $PTEName\ TreeListNodeName\ V\ PTESize$
| getmapnode $PTEName\ V\ PTESize$
| createspace

**datatype** $PTResult = RInsert\ (PTEName \times PTESize)\ option$
| $RLookup\ (PTEName \times PTESize)\ option$
| $RGetPaddr\ R \mid RSetPaddr \mid RSetLinkNode$
| $RGetMapNode\ TreeListNodeName \mid RCreateSpace\ N$

The ADT definition follows, with the semantics of these operations described further below. The *get* and *set* operations for the physical page number (getpaddr and setpaddr) are given, but omitted for the other fields ($R$,$W$,$X$,$Cached$) with the exception of the last two, since they are identical in all but name.

$Init\ \mathcal{P} = \{x \mid N\ x = [] \wedge PageTable\ x = empty\}$
$Step\ \mathcal{P}$ createspace $= create\text{-}space_a$
$Step\ \mathcal{P}$ (lookup $n\ v$) $= lookup_a\ n\ v$
$Step\ \mathcal{P}$ (insert $n\ v\ ps$) $= insert_a\ n\ v\ ps$
$Step\ \mathcal{P}$ (setpaddr $p\ r\ ps$) $= set\text{-}paddr_a\ p\ r\ ps$
$Step\ \mathcal{P}$ (getpaddr $p\ ps$) $= get\text{-}paddr_a\ p\ ps$
$Step\ \mathcal{P}$ (setlinknode $p\ m\ v\ ps$) $= set\text{-}link\text{-}node_a\ p\ m\ v\ ps$
$Step\ \mathcal{P}$ (getmapnode $p\ v\ ps$) $= get\text{-}map\text{-}node_a\ p\ v\ ps$

In the initial state there are no valid address spaces and hence no mappings.

Before mappings can be added, new address spaces need to be created. The createspace operation picks and returns the name of the new address space non-deterministically. It must be distinct from the name of any existing address space.

$create\text{-}space_a \equiv$
$\{(s,\ s',\ r) \mid \exists n.\ n \notin set\ (N\ s) \wedge s' = s(\!|N := n\ \#\ N\ s|\!) \wedge r = RCreateSpace\ n\}$

Lookup returns the PTE pointer and size of the mapping that contains $v$ assuming a valid address space $n$ is specified. The onus is on the caller to supply a valid address space to avoid a potentially unnecessary check for validity on each invocation of this operation.

$$lookup_a\ n\ v \equiv$$
$$\{(s,\ s',\ r) \mid n \in set\ (N\ s) \longrightarrow s' = s \wedge r = RLookup\ (PageTable\ s\ (n,\ v))\}$$

Insertion of new mappings is the most complicated of the operations in this model. Assuming a valid address space argument is supplied, there are two possibilities here, depending on whether the mapping overlaps an existing mapping. If an overlap exists then a conflicting mapping is not inserted. Conflicts occur if the *page-set* for $v$ at the given page size $ps$ has a non-empty intersection with the set of currently mapped virtual addresses. Figure 12 shows two examples of this.

$$valid\text{-}vaddr\ pt\ n \equiv \{x \mid pt\ (n,\ x) \neq None\}$$

$$conflict\ n\ v\ ps\ pt \equiv valid\text{-}vaddr\ pt\ n \cap page\text{-}set\ v\ ps \neq \{\}$$

If there is no conflict, *update-page-table* gives the new state, where an unused location in the PTE heap is selected and all virtual addresses in *page-set v ps* are set to this value with the given page size. The choice of location in the heap is non-deterministic in this model.

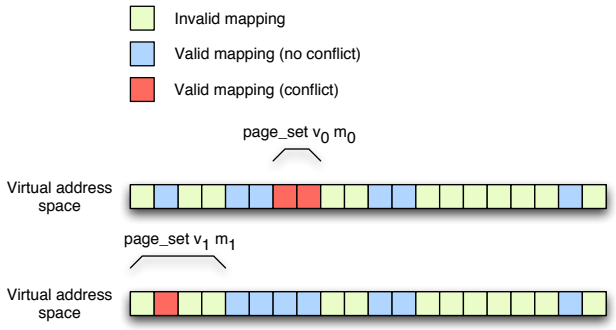


**Fig. 12.** Example conflicts with existing mappings

$$\begin{aligned}
&update\text{-}page\text{-}table\ n\ v\ ps \equiv \\
&\{(s,\ s') \mid \exists p.\ \neg\ \ PageTable\ s \vdash_p p\ \wedge \\
&\qquad s' = \\
&\qquad s(\!|PageTable := \\
&\qquad\quad \lambda(n',\ v'). \\
&\qquad\quad\ \textit{if}\ n' = n \wedge v' \in page\text{-}set\ v\ ps\ \textit{then}\ \lfloor (p,\ ps) \rfloor \\
&\qquad\quad\ \textit{else}\ PageTable\ s\ (n',\ v')|\!)\}
\end{aligned}$$

The result of this operation in the case of a successful insertion is the PTE pointer for the new mapping. If insertion was not successful, but the existing mapping fully contains the new one, a pointer to the existing mapping is returned. Otherwise the operation returns *None*.

$$
\begin{array}{l}
\mathit{insert\text{-}result}\ n\ v\ ps\ s \equiv \\
\mathsf{case}\ \mathit{PageTable}\ s\ (n,\ v)\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \\
\mid \lfloor (p,\ ps') \rfloor \Rightarrow \mathsf{if}\ \mathit{w2n}\ ps < \mathit{w2n}\ ps'\ \mathsf{then}\ \lfloor (p,\ ps') \rfloor\ \mathsf{else}\ \mathit{None}
\end{array}
$$

$$
\begin{array}{l}
\mathit{insert}_a\ n\ v\ ps \equiv \\
\{(s,\ s',\ r) \mid n \in \mathit{set}\ (N\ s) \longrightarrow \\
\qquad (\mathsf{if}\ \mathit{conflict}\ n\ v\ ps\ (\mathit{PageTable}\ s) \\
\qquad\ \mathsf{then}\ (s,\ s') \in \mathit{update\text{-}page\text{-}table}\ n\ v\ ps\ \mathsf{else}\ s = s') \wedge \\
\qquad r = \mathit{RInsert}\ (\mathit{insert\text{-}result}\ n\ v\ ps\ s')\}
\end{array}
$$

The fields of the PTE can be set and retrieved through the heap in the following operations. For each operation it is a precondition that the PTE pointer supplied is valid and the page size at which it is mapped is also correctly provided as an argument. A PTE pointer $p$ is considered valid, $pt \vdash_p p$, if it is in the image of the page table $pt$ for some page size $ps$, $pt \vdash_p p,ps$ if the pair $(p,\ ps)$ is in this image. The notation $f\ `\ A$ stands for the image of set $A$ under $f$.

$$pt \vdash_p p \equiv p \in \mathit{fst}\ `\ \mathit{ran}\ pt$$

$$pt \vdash_p p,ps \equiv (p,\ ps) \in \mathit{ran}\ pt$$

$$
\begin{array}{l}
\mathit{set\text{-}paddr}_a\ p\ \mathit{paddr}\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \longrightarrow \\
\qquad s' = s(\!|\mathit{Heap} := (\mathit{Heap}\ s)(p := \mathit{Heap}\ s\ p(\!|\mathit{Paddr} := \mathit{paddr}|\!))|\!) \wedge \\
\qquad r = \mathit{RSetPaddr}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{get\text{-}paddr}_a\ p\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \longrightarrow s' = s \wedge \\
\qquad r = \mathit{RGetPaddr}\ (\mathit{Paddr}\ (\mathit{Heap}\ s\ p))\}
\end{array}
$$

Associated with each PTE is a *link node*. While not part of the fundamental page table abstraction, this is required for the mapping database and other operations in the kernel. The link node stores the virtual address and a pointer to a corresponding node in the MDB for the mapping. In the implementation this is optimised to be a single field, with the bitwise exclusive-OR of the values stored in the link node. We model this by requiring the complementary value to be passed as an argument to the inspection operations.

$$
\begin{array}{l}
\mathit{set\text{-}link\text{-}node}_a\ p\ m\ v\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \longrightarrow \\
\qquad s' = \\
\qquad s(\!|\mathit{Heap} := (\mathit{Heap}\ s)(p := \mathit{Heap}\ s\ p(\!|\mathit{MapNode} := m,\ \mathit{Vaddr} := v|\!))|\!) \wedge \\
\qquad r = \mathit{RSetPaddr}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{get\text{-}map\text{-}node}_a\ p\ v\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \wedge v = \mathit{Vaddr}\ (\mathit{Heap}\ s\ p) \longrightarrow s' = s \wedge \\
\qquad r = \mathit{RGetMapNode}\ (\mathit{MapNode}\ (\mathit{Heap}\ s\ p))\}
\end{array}
$$

Some features that should be present in a complete page table model have been omitted here and are currently being added to the model. These range from fairly trivial changes to those necessary to increase the generality of the model.

An example of a small omission is the status and cache control bits in the PTE which are not included for brevity. These do not differ conceptually from other fields in the PTE such as permission bits for the purpose of this model. A more important limitation is that on some architectures it may not be possible to insert a mapping even if no conflicts exist due to the page table structure being affected by nearby mappings. A hardware model for the TLB and page table walker can be added to provide a *lookup* operation as described in the abstract address spaces ADT, and to supply semantics for the cache and status bits in the PTE. Finally, we assume translation and protection granularity are identical which is not the case in general, for example sub-page permissions on the ARM architecture.

### 4.2 Concrete model

A simple way to implement the page table would be to use a linear array in physical memory, indexed with the virtual page number. This would have the advantage of a fast lookup time, which is desirable as the page table lookup operation is a major component of the TLB refill cost. Unfortunately, this is wasteful of physical memory and does not scale with larger address spaces. For example, consider a 32-bit virtual address space, with 4KB pages and a 24-bit physical address space with 4KB frames[2]. Assume each page table entry is a single word, 4 bytes. The frame number can easily be stored in the PTE, requiring only 12 bits. However, the array will require $2^{20}$ PTE locations, and hence require $2^{22}$ bytes of contiguous storage in physical memory, which may potentially only be used sparsely. In addition, this has poor support for superpages, with large superpages requiring massive duplication of PTEs, making insertion and PTE update operations costly.

Modern architectures and operating systems therefore use data structures that balance the requirement for fast traversal and memory use considerations. These include multi-level page tables, inverse page tables, hashed page tables [8] and guarded page tables [13]. L4Ka::Pistachio implements a multi-level hierarchical page table (MLPT). The page table format defined by the ARM hardware, a two-level page table, is an instance of this.

MLPTs are tree data structures where each node contains an array of a fixed, level dependent, size. Elements of these arrays are either invalid, leaves corresponding to PTEs in the abstract model, or pointers to the next level. They provide both storage for valid PTE heap entries and the mapping function from $V$ to $PTEName$ for an address space $N$. Lookup proceeds by indexing the root table, with a base address equivalent to the address space name, with the most significant $k_t$ bits of the virtual address, where $k_i$ is the number of bits in the index field for level $i$ and the page table has a maximum of $t + 1$ levels. Figure 13 shows a two-level page table with the root page table indexing occurring at level 1. Each entry of the table corresponds to a contiguous region of the address space. If the address space is of size $2^n$ then the array will have $2^{k_t}$ entries and

[2] Physical pages are also called frames.

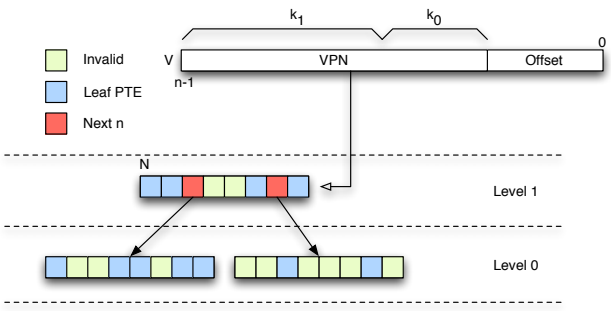


**Fig. 13.** Indexing during lookup and insertion

each entry will map a $2^{n-k_t}$ region. If either an invalid entry or leaf PTE is found at the index then a pointer to this is returned. Otherwise, if the entry points to a table at the next level then the algorithm recurses, with the next table, the $n - k_t$ least significant bits of the virtual address and the next index size, $k_{t-1}$, until either a valid PTE is found or the bottom level is reached and a pointer to the indexed entry is returned. If the returned pointer references a leaf PTE then a valid (in the sense of the abstract model) mapping exists for the virtual address.

Assuming no conflicts, insertion works similarly, with the exception that a new node of appropriate size is allocated and linked to when an invalid node is indexed at a level above the intended insertion point.

We describe the two larger operations, lookup and insertion, from the concrete model below. We utilise the verification environment of Schirmer [16] with custom pretty-printing to provide C-like syntax. Keywords, procedure names, and program variables referring to the current state are printed in typewriter font. Normal Isabelle functions and constants are unchanged. In the Hoare triple $\{\!|\sigma.\ P|\!\}\ s\ \{\!|\ Q\ ^{\sigma}n\ |\!\}$, the name $\sigma$ is bound to the pre-state, and $^{\sigma}n$ refers to the program variable $n$ in state $\sigma$.

The concrete state space has 3 components in its global state — a heap for nodes in the page table *pt-h*, a list of pointers allocated in the heap *v-pt* and the set of currently active address spaces *vN*. We model arrays as lists and hence the type of *pt-h* is given by

$$\textbf{types}\ \mathit{PageTableHeap} = \mathit{PTabName} \Rightarrow \mathit{PTE\ list}$$

where *PTabName* is the type of pointers to page table nodes and *PTE* is defined as the disjoint union:

$$\textbf{datatype}\ \mathit{PTE} = \mathit{Leaf}\ \mathit{PTE}_a \mid \mathit{Next}\ \mathit{PTabName} \mid \mathit{Invalid}$$

The type *PTEName* is now a pointer to an array entry and hence consists of two components — a pointer to the base of the array and an index of type *PTabOffset*.

$$\textbf{types}\ PTEName = PTabName \times PTabOffset$$

For convenience we introduce two abbreviations when working with *PTEName* pointers. The *PTEName* for a virtual address $v$ at level $l$ and node $n$ is given by $\varphi\ n,v,l$. The *PTE* for a *PTEName* pointer in page table heap *pt-h* is written as $\psi$ *pt-h p*.

The names of address spaces are now synonymous with the root node in the page table.

$$\textbf{types}\ N = PTabName$$

The source code for the page table lookup operation is given in Figure 15. The parameters are found in variables $n$ and $v$. Variables with a `tmp_` prefix are local to this function. Various functions are called inside the body of this function. One such example is `ptab_index` which performs the indexing operation for a given page table level, shown in Figure 14.

```
procedures ptab-index(n,v,l|r2) =
r = v >> hw_pgshifts[w2n l] &
    (1 << (hw_pgshifts[w2n l + 1] - hw_pgshifts[w2n l]) - 1);
r2 = (n, r);
```

**Fig. 14.** Page table indexing code

The function `ptab_index` takes an address space name $n$, a virtual page $v$, and a level $l$ in the page table. The result is returned in variable $r2$. The array `hw_pgshifts` represents *page-bits* from the abstract model. The predicates `pte_is_valid` and `pte_is_subtree` on *PTENames* indicate whether the dereferenced *PTE* has a flag different from *Invalid* or possesses a *Next* tag respectively. For *PTE*s of the form *Next n*, `pte_get_next` gives the pointer to the next level in the page table $n$. It is a precondition on all these functions that the supplied pointer $p$ is valid in the current state $s$, i.e. $p \in set\ (v\text{-}pt\text{-}'\ s)$. We omit the source code of these functions for brevity, with the intention that the source code presented so far provides a sufficient idea of the level of abstraction and language in which these are expressed.

The invariant is necessary to discharge the proof obligations related to the Hoare triple used to show refinement. *wfpt* is a well-formedness predicate on the page table structure, with conditions expressing properties of the tree structure, the size of nodes at different levels, the height of the tree, etc. *table-level* is a relation between page tables nodes and level numbers. $R$ is the abstraction function for the page table — a description of $R$, *pt-lookup-f* and *pt-lookup-g* is provided in Section 4.3.

Page table insertion source code is provided in Figure 16. Two additional functions are present here. `pte_make_subtree` creates a new node, allocating an array $n$ in the heap of size appropriate for the given level and setting the *PTE* referenced by the supplied pointer to *Next* $n$. `pte_make_leaf` sets the *PTE* referenced by the supplied pointer to a *Leaf* value with no access rights. The invariant is similar to that for `pt_lookup`, however the final conjunct describes the changing page table structure as new levels are added.

It should be noted that there is not necessarily any overhead from structuring the code as a series of function calls, since small functions can be either inlined during code generation or flagged as inlineable to the compiler.

```
procedures pt-lookup(n,v|r5) =
{|σ. n ∈ vN ∧ wfpt (pt_h, v_pt, vN)|}
tmp_level = pt-top-level;
tmp_tab = n;
tmp_pte = ptab_index(tmp_tab,v,tmp_level);
tmp_valid = pte_is_valid(tmp_pte);
tmp_subtree = pte_is_subtree(tmp_pte);
while (!(tmp_level == 0) && tmp_valid && tmp_subtree)
/* INV: {|wfpt (pt_h, v_pt, vN) ∧ w2n tmp_level ≤ pt-top-level' ∧
        tmp_valid = pte-is-valid' pt_h tmp_pte ∧
        tmp_subtree = pte-is-subtree' pt_h tmp_pte ∧
        tmp_pte = φ tmp_tab,v,tmp_level ∧
        (tmp_tab, w2n tmp_level) ∈ table-level vN pt_h ∧
        pt-lookup-g tmp_tab v (w2n tmp_level) pt_h =
        pt-lookup-f σ n σ v σ pt-h ∧
        v = σ v ∧ pt_h = σ pt-h ∧ vN = σ vN ∧ v_pt = σ v-pt|}
*/
{
  tmp_level = tmp_level - 1;
  tmp_tab = pte_get_next(tmp_pte);
  tmp_pte = ptab_index(tmp_tab,v,tmp_level);
  tmp_valid = pte_is_valid(tmp_pte);
  tmp_subtree = pte_is_subtree(tmp_pte);
}
r5 = (tmp_pte, tmp_level);
{|r5 = pt-lookup-f σ n σ v σ pt-h ∧ R (pt_h, vN) = R (σ pt-h, σ vN) ∧
wfpt (pt_h, v_pt, vN)|}
```

**Fig. 15.** Page table lookup code

While quite low-level, this is in fact an abstraction of actual page table implementations. In reality, multiple page levels in this model may consist of a single page level at the hardware level, where duplication is used to achieve superpages. Also, PTEs are bitfields and link nodes are stored at a fixed, level dependent, offset from the PTE. Procedures such as `pte_get_next` and `pte_is_subtree`

```
procedures pt-insert(n,v,l|r6) =
⦃σ. n ∈ vN ∧ w2n l ≤ pt-top-level' ∧ wfpt (pt_h, v_pt, vN)⦄
tmp_level = pt-top-level;
tmp_tab = n;
tmp_pte = ptab_index(tmp_tab,v,tmp_level);
tmp_valid = pte_is_valid(tmp_pte);
tmp_subtree = pte_is_subtree(tmp_pte);
while (!(tmp_level == l) && (tmp_subtree || !tmp_valid))
/* INV: ⦃w2n tmp_level ≤ pt-top-level' ∧ wfpt (σ pt-h, σ v-pt, σ vN) ∧ v = σ v ∧
        tmp_subtree = pte-is-subtree' pt_h tmp_pte ∧ vN = σ vN ∧ σ n ∈ vN ∧
        tmp_valid = pte-is-valid' pt_h tmp_pte ∧ wfpt (pt_h, v_pt, vN) ∧
        ¬ w2n tmp_level < w2n l ∧
        (tmp_tab, w2n tmp_level) ∈ table-level vN pt_h ∧
        tmp_pte = φ tmp_tab,v,tmp_level ∧
        pt-lookup-g tmp_tab v (w2n tmp_level) pt_h =
        pt-lookup-f σ n σ v pt_h ∧
        l = σ l ∧ R (pt_h, vN) = R (σ pt-h, σ vN) ∧
        (∀ x l. if pt-lookup-f σ n σ v σ pt-h = (x, l) ∧ w2n tmp_level ≤ w2n l
              then ∃ y. pt-lookup-g tmp_tab v (w2n tmp_level) pt_h =
                        (y, tmp_level)
              else pt_h = σ pt-h ∧ v_pt = σ v-pt)⦄
*/
{
  tmp_level = tmp_level - 1;
  if (!tmp_valid) {
    pte_make_subtree(tmp_level,tmp_pte);
  }
  tmp_tab = pte_get_next(tmp_pte);
  tmp_pte = ptab_index(tmp_tab,v,tmp_level);
  tmp_subtree = pte_is_subtree(tmp_pte);
  tmp_valid = pte_is_valid(tmp_pte);
}
if (!tmp_subtree) {
  if (!tmp_valid) {
    pte_make_leaf(tmp_pte);
  }
  r6 = ⌊(tmp_pte, tmp_level)⌋;
} else {
  r6 = None;
}
⦃r6 = pt-insert_a-out σ n σ v σ l (R (pt_h, vN)) ∧
R (pt_h, vN) ∈ pt-insert_a σ n σ v σ l (R (σ pt-h, σ vN)) ∧ wfpt (pt_h, v_pt, vN)⦄
```

**Fig. 16.** Page table insertion code

constitute the underlying ADT, of which concrete models should correspond to architecture-specific implementations of page tables.

### 4.3 Refinement

We can define an ADT for the operations in the above model and show refinement using the abstraction relation *r*. *pt-lookup-f* is a functional implementation of page table lookup as described in the previous section.

```
pt-lookup-g n v l pt-h =
(let p = φ n,v,n2w l
 in case ψ pt-h p of
    Next n' ⇒ if l ≠ 0 then pt-lookup-g n' v (l − 1) pt-h else (p, w-0)
    | - ⇒ (p, n2w l))
```

```
pt-lookup-f n v pt-h ≡ pt-lookup-g n v pt-top-level' pt-h
```

The function $R$ maps from concrete page tables to the abstract page table function. This hides the type of nodes other than *Leaf* by returning *None* if the pointer returned by page table lookup does not reference a *Leaf*.

```
R c ≡
let (pt-h, N) = c
in λ(n, v).
     if n ∉ N then None
     else let (p, l) = pt-lookup-f n v pt-h
          in if ψ pt-h p ≠ Invalid then ⌊(p, l)⌋ else None
```

The same set of valid address spaces should be in both concrete and abstract models. Valid *Leaf PTE*s appear at the same location in the abstract heap.

```
r ≡
{(a, c) | set (N a) = vN-' c ∧
 (∀ p. case ψ pt-h-' c p of
       Leaf pte ⇒ pt-h-' c,v-pt-' c ⊢ p ⟶ Heap a p = pte | - ⇒ True) ∧
 PageTable a = R (pt-h-' c, vN-' c)}
```

The conditions in the Hoare triple specifications for the source code ensure that well-formedness holds and is preserved, that the abstraction relation holds on the concrete and abstract states pre- and post-operation, and that the expected values are returned. Using the soundness result of the Hoare logic [16], we get that the concrete implementations on the semantic level correctly simulate the abstract model of page tables.

## 5 Conclusion

We have presented some important aspects of the refinement process in verifying 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 taken the memory lookup operation of this model, and

described its implementation in the kernel using a page table data structure. We have further refined this abstract view of the page table data structure towards an implementation in the C programming language.

While we have not yet completely reached the level of C source code as it is accepted by standard C compilers, it is already apparent that this final step is within reach.

Further work in this direction includes enhancements to the Hoare-logic verification environment, such as the ability to directly use concrete C-syntax within Isabelle, as well as using this verified implementation of page tables as a drop-in replacement for the current L4Ka::Pistachio implementation. Our final goal is a verified, high performance implementation of L4. Since our verified implementation is very close in terms of code and data structures being used to the existing one, we do not expect any decrease in performance.

*Acknowledgements* We thank Espen Skoglund for providing a clean and generic interface of the page table data structure in L4Ka::Pistachio that was nicely amenable to verification.

## References

1. ARM Limited. *ARM Architecture Reference Manual*, Jun 2000.
2. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., Dec 1994.
4. Thierry Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.
5. 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.
6. Gregory Duval and Jacques Julliand. Modelling and verification of the RUBIS $\mu$-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.
7. 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.
8. Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proc. 20th ISCA*, pages 39–50. ACM, 1993.
9. Gerwin Klein and Harvey Tuch. Towards verified virtual memory in L4. In *TPHOLS'04 emerging trends*, Park City, Utah, Sep 2004.
10. *L4 eXperimental Kernel Reference Manual Version X.2*, 2004.
11. L4Ka Team. L4Ka::Pistachio kernel. `http://l4ka.org/projects/pistachio/`.
12. Jochen Liedtke. On $\mu$-kernel construction. In *Proc. 15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.
13. Jochen Liedtke. *On the Realization Of Huge Sparsely-Occupied and Fine-Grained Address Spaces*. Oldenbourg, Munich, Germany, 1996.
14. Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcom Newey, editors, *Proc. Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

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, Computer Science Laboratory, SRI International, 1980.
16. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In Gerwin Klein, editor, *Proc. NICTA workshop on OS verification 2004, Technical Report 0401005T-1*, Sydney, Australia, Oct 2004.
17. J. S. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS-97-26, Distributed Systems Laboratory, University of Pennsylvania, 1997.
18. Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proc. 19th ISCA*. ACM, 1992.
19. 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.
20. 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.