Towards High-Assurance Multiprocessor Virtualisation

Michael von Tessin
NICTA[*] and University of New South Wales
Sydney, Australia
michael.vontessin@nicta.com.au

Abstract

Virtualisation is increasingly being used in security-critical systems to provide isolation between system components. Being the foundation of any virtualised system, hypervisors need to provide a high degree of assurance with regards to correctness and isolation. Microkernels, such as seL4, can be used as hypervisors. Functional correctness of seL4's uniprocessor C implementation has been formally verified. The framework employed to verify seL4 is tailored to facilitate reasoning about sequential programs. However, we want to be able to use the full power of multiprocessor/multicore systems, and at the same time, leverage the high assurance seL4 already gives us for uniprocessors.

This work-in-progress paper explores possible multiprocessor designs of seL4 and their amenability to verification. For the chosen design, it contributes a formal multiprocessor execution model to lift seL4's uniprocessor model and proofs into a multiprocessor context using only minor modifications. The theorems proving the validity of the lift operation are machine-checked in Isabelle/HOL and walked-through in the paper.

1 Introduction

Virtualisation has gained huge popularity over the past decade, mostly in the desktop and server space to host multiple operating systems on the same hardware and efficiently manage available resources. Seeing operating systems being less and less able to prevent security exploits, virtualisation started to be increasingly used to provide isolation between system components. Today, this is especially true for embedded systems in security- or safety-critical areas [7]. Hypervisors used in these systems have to provide a high degree of assurance with regards to correctness, reliability and isolation. While careful design and extensive testing can increase the degree of assurance, they can never guarantee that a hypervisor is bug-free.

Microkernels are increasingly being used as hypervisors. As a high-performance microkernel in the L4 family, seL4 enables fine-grained dissemination of authority via capabilities which allows controlled communication between otherwise isolated components. The seL4 verification project *L4.verified* [8] has formally verified that the C implementation is functionally correct, i.e. that it refines seL4's *abstract specification* which is written in Haskell-like monadic style [4], shallowly embedded into Isabelle/HOL [9]. Isolation properties—e.g. disconnected subsystems running on top stay disconnected forever—are formulated and proved in a high-level *security model* [3] which talks about how authority is disseminated in seL4's programming model. A proof is currently being undertaken to show that the abstract specification refines the security model.

While being implemented for the ARMv6 and IA-32 (aka x86) architectures, seL4 is formally verified only for ARMv6. I implement and evaluate the multiprocessor version on an IA-32 multicore system, mainly because of their abundant availability as opposed to ARMv6 multicore systems. However, the concepts of this paper are completely architecture-independent.

[*]NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

The seL4 microkernel is implemented for uniprocessors only, is non-preemptible[1] and the framework used for its verification is tailored to facilitate reasoning about sequential programs. However, multiprocessor/multicore[2] systems are now omnipresent in desktops/servers and increasingly being used in the embedded world. We would like to leverage seL4's high assurance on a multiprocessor system as well.

This paper analyses possible multiprocessor designs of seL4 for their tradeoffs and amenability to verification (section 2). For the chosen design, it contributes a formal multiprocessor execution model in section 3, connects it to a monadic abstract specification of seL4's multiprocessor-specific code (section 4), formulates the required isolation properties and walks through the proofs (section 5). In section 6, it connects the multiprocessor abstract specification (1.2 kLOC) to seL4's uniprocessor abstract specification (4.9 kLOC [8]) in order to lift the massive amount of existing uniprocessor proofs (200 kLOC [8]). This enables an overall proof showing that the validity of seL4's refinement proof has been preserved. Related work is discussed in section 7 and section 8 concludes.

All formal specifications and proofs presented in this paper are machine-checked in the interactive theorem prover Isabelle/HOL [9].

2 Multiprocessor Kernel Designs

Formal verification of concurrent systems is hard and becomes intractable very quickly because all interleavings of memory accesses have to be considered. My approach to solving this problem is taking a non-standard attempt of multiprocessor kernel design instead of tackling it only from the formal side. My design goal is to avoid concurrency as much as possible. The two fundamental ways to do that are: (1) to avoid parallelism or (2) to avoid sharing.

The first design, a ***big-lock kernel*** (Figure 1(a)), avoids parallelism by having a big lock around the whole kernel which only allows one CPU to run kernel code at any given time. This has the advantage of being able to make the presence of multiple CPUs transparent and present a superset of seL4's uniprocessor ABI[3] to the system software running on top. As such, system software written for uniprocessor seL4 can run unmodified on top of the multiprocessor version, and at the same time, use all available CPUs. In case of virtualisation, the system software is a *virtual machine monitor* (VMM) running one or more *virtual machines* (VMs) with each one hosting a *guest operating system* (guest OS).

The disadvantage of this design is that kernel calls scale badly with an increasing number of CPUs. While it seems that having no parallelism in the kernel preserves the validity of seL4's uniprocessor proofs, this is only half the truth, because the uniprocessor kernel code still has to be extended significantly in order to be able to deal with multiple CPUs. Despite still being able to reason about seL4 as a sequential program, we have to add several blocking operations which require one additional thread state each, adding significantly to the complexity of verifying the newly added code.

The second design resembles a ***multikernel*** (Figure 1(b)). It takes the opposite approach and allows full parallelism but avoids sharing by running a separate *instance* of uniprocessor seL4 per CPU. Resources (physical memory, devices) are partitioned between instances, nothing is shared. To start with, I propose a *static setup* of the multikernel design, i.e. the resource partitioning is determined during the bootstrapping of the kernel (when the system is powered on) and never changed thereafter. In this setup, it is possible to avoid writing multiprocessor-specific code for anything other than the kernel's bootstrapping. This means once bootstrapped, each kernel instance is indistinguishable from the uniprocessor

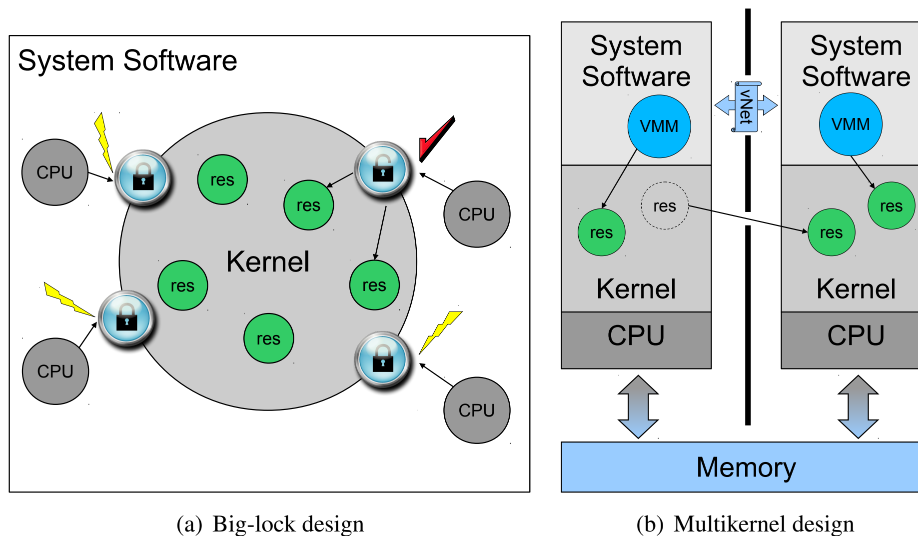
[1] seL4 actually has two explicitly defined preemption points, but because we save a continuation and exit the kernel (early) before taking the pending interrupt, we preserve sequential C code semantics.

[2] For the purpose of this paper, *multiprocessor* and *multicore* can be considered synonyms.

[3] The kernel ABI (Application Binary Interface) is the assembly-level interface the kernel provides to system software running on top.

version running on a uniprocessor system. Because seL4's refinement proof only covers the code of the running kernel and does not talk about its bootstrapping, the static setup preserves the validity of the uniprocessor proofs if we prove that each instance only accesses its own kernel state.

While the static nature is acceptable for embedded virtualisation, we need more flexibility for desktop and server virtualisation. I therefore propose a *one-off dynamic setup*: When the system is powered on, we only start one seL4 instance on one CPU and give it authority over all available resources. System software (e.g. a VMM) is then able to either ask the end user or work out the desired resource partitioning itself before starting the other instances (via a new `StartInstance` system call) which also transfers the authority over the assigned resources to the new instances. This setup allows resources to be assigned dynamically and VMs to be started anytime during runtime by an end user. It introduces no sharing of resources because they are immediately and irrevocably transferred to an instance when starting it. However, the implementation of `StartInstance` will need sharing of some minimal kernel data structures between the current CPU and the CPU to be started in order to coordinate the start and transfer of resources. Because this happens during the runtime phase of the kernel, the uniprocessor proofs' validity can not be automatically preserved anymore.



(a) Big-lock design (b) Multikernel design

Figure 1: Multiprocessor kernel designs

Running completely isolated VMs may be desired in a wide range of use cases, but very often, we also need controlled communication between them, e.g. via a virtual network. In order to enable a VMM or a guest OS running in a VM to implement this, we provide a dedicated *shared memory region* to them. Because the seL4 kernel never accesses this region, it does not introduce any concurrency into the kernel.

In summary, the big-lock kernel design sacrifices *scalability* for verifiability whereas the multikernel design sacrifices *flexibility* for verifiability.

3 Formal Multiprocessor Execution Model

On a multiprocessor system, each CPU runs its own instruction *stream* independently from the others. There is no common clock and as such, it cannot be predicted in which global order instructions of different CPUs are executed. In order to reason about global ordering, we need to take into account all possible instruction interleavings between the instructions streams of all CPUs which very quickly results in a combinatorial state explosion. Fortunately, interleavings do not matter unless there is interdepen-

dence between instruction streams. On a multiprocessor system, instruction streams interact with each other by reading/writing physical memory or by sending *inter-processor interrupts* (IPIs) to each other. The memory accesses give us an observable global order because in the architectures considered here the memory bus only allows one CPU to read/write at any given time and therefore effectively synchronises them on each physical memory access[4]. This behaviour imposes an explicit memory-access interleaving on otherwise independent instruction streams. Between two memory accesses, a CPU runs completely independently from all others unless interrupted by an IPI. This allows to abstract from low-level native CPU instructions and only care about three *high-level instructions* that interact with the global system state:

- **Read** and **Write** model all native CPU instructions which read or write physical memory in any way. Native instructions between them are CPU-local and have no effect on the global system state. They merely affect the parameters of the next high-level instruction which then interacts with the global system state.
- **StartCPU** models all native CPU instructions which start another CPU. It models that on most multiprocessor-capable architectures, only one CPU is started when the system is powered on. It is up to this CPU to start the remaining CPUs. This is mostly done by sending a specific, architecture-defined wakeup IPI to the CPUs to be started. Unlike *Read* and *Write*, *StartCPU* synchronises high-level-instruction streams, i.e. its placement has a direct effect on which interleavings are possible and which are not. In order to model the multikernel design (see section 4), *StartCPU* is the only synchronising high-level instruction required. Nonetheless, further high-level instructions to model generic synchronisation primitives such as *compare-and-swap* (CAS) can be added.

The formal definition of a high-level instruction is:

$$\textbf{datatype}\ \textit{instr-t} = \textit{InstrRead p-region-t} \mid \textit{InstrWrite p-region-t} \mid \textit{InstrStartCPU cpu-t}\ ''\textit{instr-t list}''$$

where *cpu-t* is the hardware CPU ID and *p-region-t* is a set of byte-granular physical memory addresses. Each high-level instruction is considered atomic, i.e. the entire memory region given as parameter is read/written at once. This enables modelling that reading and writing word-sized and -aligned values is atomic which is true for most architectures. As a parameter, *InstrStartCPU* takes the high-level-instruction stream (*instr-t list*) to be executed on the CPU to be started.

The global system state is divided into two parts at any point in time: (1) the observed memory-access history (“the past”) which also defines the current memory contents, and (2) the high-level-instruction streams that remain to be executed on all active CPUs (“the future”). Formally, a memory-access history is an *acc-t list* where

$$\textbf{datatype}\ \textit{acc-t} = \textit{AccRead p-region-t cpu-t} \mid \textit{AccWrite p-region-t cpu-t}$$

From a systems point of view, this models exactly what the memory bus observes: a read or write of memory covering one or more memory addresses, issued by a specific CPU.

The global system state (aka **multi**pro**c**essor **s**ystem **s**tate) is defined as:

$$\textbf{record}\ \textit{mpss-t} = \textit{mpss-al} :: \textit{acc-t list}$$

For extensibility reasons, *mpss-al* (**a**ccess **l**ist) is the only field of *mpss-t* which means that our global system state only consists of the memory-access history and does not contain any information about the

[4]On modern architectures where we have caches, it is the cache coherence protocol that synchronises the instruction streams, not the memory bus. Nevertheless, the semantics is exactly the same.

actual values stored in memory. Handling memory content must be done by an abstraction layer talking about lower-level instructions which then have to be mapped to high-level instructions in order to connect this model with that layer (e.g. the monadic *abstract specification* of seL4 that will be presented in section 4). This design choice simplifies models and proofs but, of course, also limits the expressiveness of properties that can be proved. If I was to reason about generic synchronisation primitives such as CAS, the model would at least have to contain information about the values of the synchronisation variables.

The high-level-instruction streams that remain to be executed on all active CPUs at a given point in time (earlier called "the future") are not directly observable and thus not part of *mpss-t*. Formally, I call them per-**C**PU **i**nstruction **l**ist:

$$\textbf{types}\ \textit{cil-t} = \textit{cpu-t} \Rightarrow \textit{instr-t list}$$

This is a total function from CPU IDs to their high-level instruction streams that are still to be executed. If a CPU is inactive, i.e. is not yet started or does not exist, the function returns the empty list.

Semantics Given a *parallel program* (type *cil-t*), an *operational semantics* defines a mapping from any initial state to all possible final states using a *transition relation* between *configurations*. A configuration is the combination of "the future and the past" which is a tuple of type *cil-t* $\times$ *mpss-t*. The transition relation and its *transition rules* are defined as:

$$\textbf{inductive_set}\ \textit{trans-rel} :: ((\textit{cil-t} \times \textit{mpss-t}) \times (\textit{cil-t} \times \textit{mpss-t}))\ \textit{set}$$
$$hd\ (cil\ c) = InstrRead\ r \Longrightarrow ((cil,\ s),\ cil(c := tl\ (cil\ c)),\ instr\text{-}Read\ r\ c\ s) \in trans\text{-}rel$$
$$hd\ (cil\ c) = InstrWrite\ r \Longrightarrow ((cil,\ s),\ cil(c := tl\ (cil\ c)),\ instr\text{-}Write\ r\ c\ s) \in trans\text{-}rel$$
$$hd\ (cil\ c) = InstrStartCPU\ cn\ il \Longrightarrow ((cil,\ s),\ cil(c := tl\ (cil\ c),\ cn := il),\ s) \in trans\text{-}rel$$

Each transition rule handles one high-level instruction by removing it from *cil* and modifying the global system state accordingly. The transition rule for *InstrStartCPU* does not modify *s* because starting a CPU does not involve accessing memory[5]. Instead, it modifies the *cil* by giving the new CPU a program to be executed. The semantics of *InstrRead* are:

$$instr\text{-}Read :: p\text{-}region\text{-}t \Rightarrow cpu\text{-}t \Rightarrow mpss\text{-}t \Rightarrow mpss\text{-}t$$
$$instr\text{-}Read\ p\text{-}reg\ cpu\text{-}id\ mpss \equiv mpss(|mpss\text{-}al := mpss\text{-}al\ mpss\ @\ [AccRead\ p\text{-}reg\ cpu\text{-}id]|)$$

InstrWrite is defined accordingly. Currently, these instructions only append to the memory-access history (by updating the field *mpss-al* of *mpss*). More semantics to model actual memory contents can be added.

The operational semantics of the multiprocessor execution model is now defined as follows:

$$sem :: mpss\text{-}t \Rightarrow cil\text{-}t \Rightarrow mpss\text{-}t\ set$$
$$sem\ mpss\ cil \equiv \{s.\ ((cil,\ mpss),\ \lambda c.\ [],\ s) \in trans\text{-}rel^*\}$$

It is a function that returns all possible final states (memory-access histories) given an initial state *mpss* and parallel program *cil*. "Final" means that all CPUs have executed their program ($cil = \lambda c.\ []$). "*" denotes the reflexive transitive closure of a relation.

In most use cases of this model, the initial state *mpss* will contain an empty memory-access history and the top-level theorem will say that a desired predicate holds for all observable memory-access histories generated by *sem* when fed with the parallel program in question.

[5]On certain architectures, starting CPUs via IPIs involves writing to a specific region of physical memory. Nevertheless, this region of memory is hard-wired to the CPU's IPI mechanism and these writes never make it to the memory bus which allows me to ignore them in the model.

Summary The multiprocessor execution model (200 LOC in Isabelle) takes a parallel program of type *cil-t* and returns all memory-access histories (*mpss-t set*, or in fact, *acc-t list set*) that can potentially be observed during runtime of the system. The *instr-t list* depicted in Figure 2 is a common example of a program where the system boots with one CPU running which starts the others later.

The model incorporates that certain interleavings can never be observed, e.g. all memory accesses performed before executing *StartCPU* are always observed before any memory access performed by the newly started CPU. It also models that two writes from the same CPU are always observed in the original order by all other CPUs. This is not true on architectures with out-of-order execution, such as the IA-32, which require implicit or explicit memory barriers in order to guarantee a desired order. Luckily, on the IA-32, starting a CPU is done with an IPI which is also an implicit memory barrier. Therefore, my model is sufficient because I only rely on *StartCPU* to enfore a global order on memory accesses. Nevertheless, extending the model to generic synchronisation primitives would require explicitly modelling memory barriers. We could model them as high-level instructions and generate a set of all possible orderings of high-level instructions between the barriers.

4 Modelling seL4's Multiprocessor-specific Code

My approach for getting the desired degree of assurance of the new multiprocessor-specific code's correctness is not to prove properties about the C code directly, but instead, work with an *abstract specification* thereof. For each C function, I define its abstract counterpart as a monadic function with the same name, parameters, return value and functionality. First, it allows to specify the desired behaviour at a more abstract level without having to reason about C and system-programming issues at the same time, and second, it makes theorems much easier to understand and greatly simplifies proofs.

I use the same specification language and framework as L4.verified. It is a Haskell-like monadic language with non-deterministic choice, shallowly embedded into Isabelle/HOL. The framework facilitates reasoning about Hoare triples by providing a VCG. Details are described in [4]. This framework only allows reasoning about sequential programs, e.g. when we model accessing variables with it, we know the value of a variable we are reading if we have read or written the same variable before. In a multiprocessor environment we generally do not, because another CPU could have overwritten this variable in the meantime.

In order to lift the framework into a multiprocessor context, I connect the multiprocessor execution model with the abstract specification in the following way (Figure 2):

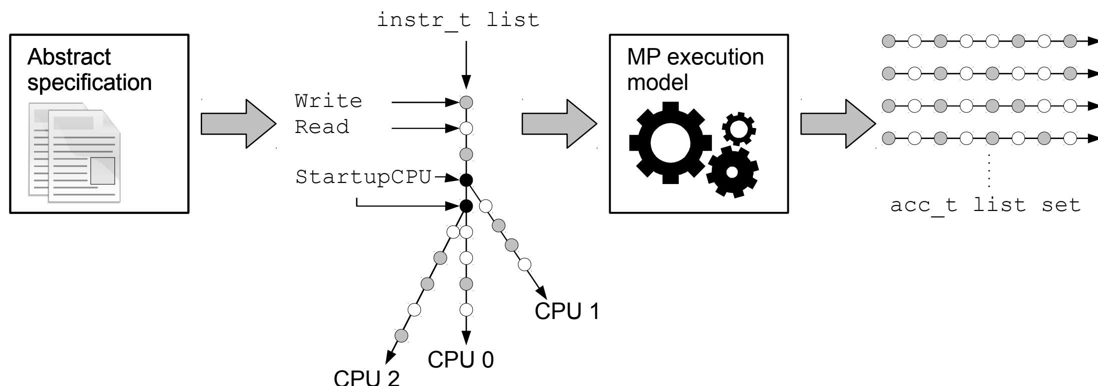


Figure 2: Connection between abstract specification and multiprocessor execution model

1. We **assume** that locally on each CPU, the program observes sequential semantics, e.g. we read the same variable value we have read or written before, and I prove all desired properties about the abstract specification using sequential semantics.
2. We map instructions in the abstract specification to high-level instructions and thereby let it create a high-level instruction list as it "executes".
3. We feed this list to the multiprocessor execution model as a parallel program (of type *cil-t*) and prove that for all possible access histories, the initial assumption was justified: any program running on any CPU has in fact observed sequential semantics.

This decouples reasoning about program-internal behaviour from reasoning about concurrency. It makes modelling and proofs more modular and thus easier but also restricts the kind of parallel programs and properties that can be proved, i.e. proofs about lock-free data structures and algorithms would not be possible because there, program-internal behaviour and concurrency are inherently coupled.

However, it turns out that these limitations are not too restrictive for my purposes, mainly because for modelling the static setup of the multikernel design, new multiprocessor-specific code is restricted to the ***bootstrapping phase*** of the kernel (see Figure 3). The fact that the uniprocessor code implementing the ***runtime phase*** stays the same decouples uniprocessor code from multiprocessor code, bootstrapping specification from runtime specification and the L4.verified project from my project at exactly the one point of changing phases.

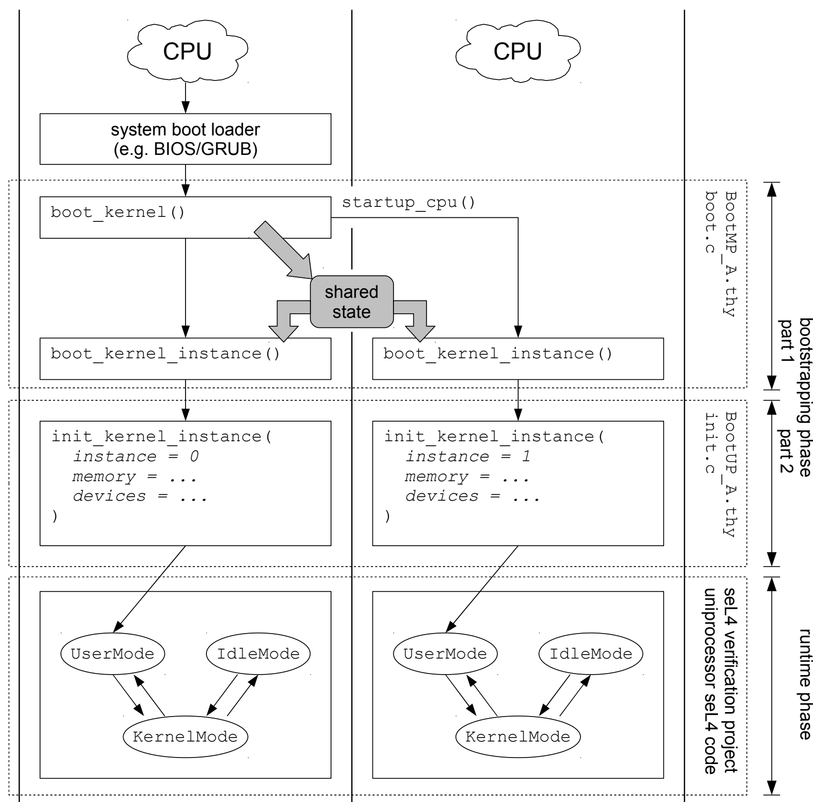


Figure 3: Bootstrapping and runtime phases of the multikernel design's static setup

Abstract Specification The abstract specification of seL4's multiprocessor-specific bootstrapping code runs in a state monad $'a$ *mpms-monad-t* with $'a$ being the type variable of a function's return value. The

multi**p**rocessor **m**onad **s**tate is defined as **record** *mpms-t* with the following fields:

- *mpms-pmem-var* :: *pmem-var-t* models the content of physical memory. The definition of type *pmem-var-t* will be provided later in the subparagraph “Memory Content”.
- *mpms-cpu-id* :: *cpu-t* is the CPU executing the current monadic computation. It reflects the scenario of a C program being able to query the ID of the CPU it is currently running on via an assembly function and letting future execution depend on it, a feature both the C code and its abstract specification make use of.
- *mpms-il* :: *instr-t list* is the high-level-**i**nstruction **l**ist we append to every time we execute a monadic function which is mapped to a high-level instruction. Establishing this mapping simply means having the function perform the append operation explicitly.

The monadic function mapped to *InstrStartCPU* is defined as:

$$\textit{start-cpu} :: \textit{cpu-t} \Rightarrow \textit{unit mpms-monad-t} \Rightarrow \textit{unit mpms-monad-t}$$

It returns *unit mpms-monad-t* which means it is monadic and has no return value (type *unit*). As first parameter, it takes the ID of the CPU to be started (*start-cpu-id*) and as second parameter, the monadic function that the new CPU will execute (*start-fun*). It first calls *start-fun* with a monad state consisting of the current memory contents *mpms-pmem-var*, *mpms-cpu-id* set to *start-cpu-id* and *mpms-il* initialised with the empty list. After *start-fun* returns, *start-cpu* extracts from its monad state the new memory contents and the accumulated high-level instruction list *start-il* which it immediately uses to append *InstrStartCPU start-cpu-id start-il* to *mpms-il*. This recursively assembles a tree-like *instr-t list* structure as depicted in Figure 2 which is nothing else but the parallel program that we feed to the multiprocessor execution model.

Bootstrapping Because on the C level, the function pointer of *start-fun* is the start instruction pointer given to the new CPU, which is the only information we can transfer to the new CPU, we can neither pass parameters nor handle a return value. This fact inspired splitting the bootstrapping phase into two ***parts*** (see Figure 3):

1. When the system is powered on, *boot-kernel* is executed on the first CPU. It discovers system resources, splits them up between the instances and stores each instance’s configuration by writing to shared state. The configuration contains the assigned resources and a few other system-level parameters necessary to bootstrap an instance. *boot-kernel* now starts the other CPUs, pointing them at *boot-kernel-instance* before calling *boot-kernel-instance* itself. On each CPU (including the first one), *boot-kernel-instance* queries the ID of the CPU it is running on and reads the corresponding instance’s configuration from the shared state before passing it to *init-kernel-instance* via parameters.
2. *init-kernel-instance* can now take a completely instance-local view and is provided with exactly the configuration parameters it needs to bootstrap an instance which includes configuring CPU and kernel devices, creating all kernel-internal data structures and loading/mapping/starting the system software running on top of the kernel.

Bootstrapping Part 1 I call the previously mentioned “shared state” ***m**ulti**p**rocessor **k**ernel **s**tate* and define it as **record** *mpks-t* whose fields are accessed by the monadic functions in exactly the same way as their C counterparts access the fields of a static, globally defined C struct with the same name. The important fields of *mpks-t* are:

- *mpks-avail-p-reg* :: *p-region-t* is the **reg**ion of ***avail**able* **p**hysical memory. In C, *boot-kernel* gets this information from the *system boot loader* (e.g. BIOS/GRUB) and writes it to this field.
- *mpks-ki-p-reg* :: *p-region-t* is the **p**hysical memory **reg**ion where the ***k**ernel **i**mage* is in. The kernel image contains all static kernel data[6] and kernel code, incl. bootstrapping functions such as *boot-kernel*. It is loaded by the system boot loader which also provides this information to *boot-kernel*, which in turn, writes it to this field. It needs this information to exclude the kernel-image memory region from the available memory region.
- *mpks-sh-p-reg* :: *p-region-t* is the **p**hysical memory region that will be provided to each instance's system software as a **sh**ared **reg**ion. *boot-kernel* picks a region from the available region, writes it to this field and then excludes it from the available region.
- *mpks-dev-p-regs* :: *p-region-t set list* stores, for each instance, a set of device memory regions which is essentially the set of devices assigned to the instance[7]. I am modelling the static setup, so *boot-kernel* creates this list by discovering the devices and directly assigning them to instances, according to a static mapping that the end user specified before powering up the system.

Memory Content Memory content is modelled as a partial function from physical addresses to different types of variables:

$$\textbf{types}\ \textit{pmem-var-t} = \textit{paddr-t} \rightharpoonup \textit{var-t}$$
$$\textbf{datatype}\ \textit{var-t} = \textit{VarMpKs mpks-t} \mid \textit{VarUpKs upks-t} \mid \textit{VarKAcc cpu-t}$$

It is now possible to model that the *mpks-t* previously introduced resides at a specific physical memory address. I *underspecify* this address as **consts** *var-mpks*::*paddr-t* which fixes the address but leaves the actual value undefined. In C, this value is chosen by the compiler/linker when building the kernel image. The monadic functions that read/write *mpks-t* contain functions that append *InstrRead* and *InstrWrite* to *mpms-il* which maps them to these high-level instructions. In order to provide a memory region to them, not only a single memory address, I also underspecify **consts** *mpks-size*::*nat*. I assume correctness of compiler/linker and have to axiomatise that $\{\textit{var-mpks}..\textit{var-mpks}+\textit{mpks-size}-1\}$ lies in the kernel-image memory region.

Reading a variable boils down to applying the physical memory address to the function *mpms-pmem-var* while writing it makes use of Isabelle's *function update* feature. Note that the value of a variable is only stored in the first memory address byte, even though the (possibly underspecified) size might be bigger than 1 byte. It reflects the reality in the sense that when using C pointers, we also point at the first memory address byte. Then again, it does not model well what happens when someone accidentally directly modifies the non-first memory address occupied by the variable.

Bootstrapping Part 2 The second variable type (*VarUpKs*) is a **record** *upks-t* (**u**ni**p**rocessor **k**ernel **s**tate) and contains all static kernel data an seL4 instance needs during the runtime phase. Each instance initialises its own *upks-t* state independently when running the function *init-kernel-instance* which is part 2 of the bootstrapping phase (see Figure 3).

I define a new state monad *'a upms-monad-t*. The **u**ni**p**rocessor **m**onad **s**tate is defined as **types** *upms-t* = *upks-t*. This means that, unlike in the *mpms-monad-t*, functions running in the *upms-monad-t* directly operate on static kernel data. When *boot-kernel-instance* calls *init-kernel-instance*, the *upks-monad-t* is lifted into the *mpks-monad-t*. Upon return, the entire memory region accessed (read or written) by

[6] *Static kernel data* is allocated within the kernel image by the compiler/linker at build time. It is initialised in the bootstrapping phase, exists during the whole runtime phase and is never deallocated.

[7] This reflects the fact that most devices nowadays are memory-mapped, i.e. they are used by accessing the physical memory region where they are mapped to.

init-kernel-instance is marked as "accessed by CPU with ID *cpu-id*" by writing the value *VarKAcc cpu-id* into every accessed memory address. The accessed memory region had been accumulated into a ghost-variable field of the **record** *upks-t* while executing *init-kernel-instance*.

Memory Content Static kernel data does not suffice to implement a non-trivial kernel. The system software running on top of it is dynamic and will create new threads, address spaces etc. which forces the kernel to dynamically allocate kernel memory from the available memory region during the runtime phase. In seL4, authority over physical memory is conferred to system software by means of *capabilities*. By invoking such a capability, the system software can tell the kernel to dynamically allocate *kernel objects* (e.g. threads, page tables for address spaces) within the physical memory region the capability gives authority to. While the system software receives authority over these allocated kernel objects via new capabilities, only the kernel is allowed to directly access them. The only exception is *frame objects* which are directly used by system software to store code and data. It is also possible for system software to explicitly deallocate kernel objects again and reuse the physical memory. This feature is indispensable for implementing a truly dynamic system and accounts for most of the kernel's complexity.

In the abstract specification, allocations and contents of kernel objects are stored in a *kernel heap*, stored in field *upks-kheap* of **record** *upks-t*. It is a partial function from memory addresses to objects:

$$\textbf{types}\ \textit{kheap-t} = \textit{pptr-t} \rightharpoonup \textit{obj-t}$$

Accessing the kernel heap works in the same way as accessing *mpms-pmem-var* (described earlier), except that instead of a *paddr-t*, we use a *pptr-t* which is nothing else than a *paddr-t*, offset by a constant[8]. This implementation detail is made explicit in the model because correctness relies on these addresses being converted in various scenarios which was the cause of kernel bugs in the past.

During kernel bootstrapping, we not only initialise static kernel data but also need to create a minimum set of objects/capabilities needed to bootstrap the system software running on top. This includes the *initial thread* object, page table objects for the initial thread's address space and frame objects where the system software code/data is stored in. In addition, we also create frame objects that cover the shared memory region, hereafter called *shared frames*. These are the frames a VMM or guest OS will use to implement a virtual shared network between instances. After all necessary objects have been created, the kernel creates capabilities that cover the remaining available memory. All these capabilities are given to the system software's initial thread when it is started. Going into the details of seL4's kernel data structures and object types is beyond the scope of this paper, but Table 1 gives a feeling of its complexity.

abstract specification of	semantics of model	C data structures	C functions
bootstrapping part 1	100 LOC	100 LOC	100 LOC
bootstrapping part 2	100 LOC	300 LOC	500 LOC

Table 1: Abstract specification's count of lines of Isabelle code

In seL4, the initial thread gets all authority (capabilities) and is trusted to setup the system and distribute the capabilities correctly, similar to other microkernel-based systems. In the multikernel design, we have multiple initial threads which are responsible for setting up the system in a coordinated manner. They are able to selectively disseminate the shared frames to different subsystems (e.g. VMs in the same security domain) and thereby connect or disconnect them across instances.

I will also investigate kernel-provided communication mechanisms which would allow even more fine-grained control of information flow but also introduce concurrency into the runtime phase.

[8] For performance reasons, the kernel does not access physical memory (*pptr-t*) directly, but instead, via a *kernel window* in virtual memory that maps to physical addresses with a constant offset.

5 Properties Proved

The desired properties to be proved about seL4's multikernel model can be divided into 3 categories:

1. The **refinement proof** of uniprocessor seL4 remains valid (see section 6).
2. The kernel behaves correctly with regards to concurrency. This means that the initial assumption is correct in saying that all monadic functions of seL4's abstract specification observe sequential semantics. I will call this the ***kernel-memory-access theorem***.
3. Kernel objects and capabilities created during bootstrapping and given to the system software's initial thread conform to seL4's multiprocessor ABI. In particular, this means that the shared frames created are backed by the same physical memory region on every instance of seL4, and that the physical memory region backing the non-shared frames is disjoint between instances. I will call this the ***virtualisation theorem*** because it provides the formal foundation for VMMs to run VMs isolated[9] from other instances and, at the same time, rely on the shared frames to actually be shared in order to implement a virtual shared network between instances.

Kernel-Memory-Access Theorem In order to prove this theorem, we first have to prove that objects and capabilities are created correctly within an instance of seL4. Thus, the first proof step establishes the following lemma about bootstrapping part 2:

"After *init-kernel-instance* has initialised the kernel state of an instance,
the memory region covered by all kernel objects and capabilities (except frames),
and all memory accessed during this process, lie in the available memory region."

Using this lemma, the second proof step over bootstrapping part 1 shows that the assigned available memory region is disjoint between instances and then establishes the following lemma about the parallel high-level-instruction program of seL4's bootstrapping phase.

"For each *InstrStartCPU start-cpu-id start-il* high-level instruction in
mpms-il boot-kernel-mpms, there is no overlap between the physical memory region
read by the subsequent instructions and **written** by the instructions in *start-il*, and the one
written by the subsequent instructions and **read** by the instructions in *start-il*."

mpms-il boot-kernel-mpms is the parallel high-level-instruction program of type *instr-t list* taken from the final state of the *mpms-monad-t* after *boot-kernel* has returned.

Using this lemma, the third and final proof step—which is planned in future work—will establish the final kernel-memory-access theorem which is formulated on the level of memory-access histories generated by the semantics function *sem* when fed with the above parallel high-level-instruction program:

$$\textbf{theorem}\ \forall al \in mpss\text{-}al\ `\ sem\ init\text{-}mpss\ (cil\text{-}of\text{-}il\ (mpms\text{-}il\ boot\text{-}kernel\text{-}mpms)).\ has\text{-}seq\text{-}sem\ al$$

The function *cil-of-il* takes an *instr-t list* and returns a *cil-t* by assigning the *instr-t list* to the first CPU. The kernel-memory-access predicate over memory-access histories *has-seq-sem* ("**has seq**uential **sem**antics") is formulated as:

"For each physical address, after it is first read or written by CPU *cpu-id*,
no CPU other than *cpu-id* writes to it anymore."

[9]Like L4.verified, this work only covers isolation with respect to overt channels. While being sufficient to guarantee e.g. fault isolation, we would need covert channel analysis to guarantee complete isolation with respect to information flow.

Virtualisation Theorem This theorem has a system software's point of view. Instead of talking about memory accesses by the kernel, it is about physical memory covered by capabilities provided to the initial thread. The first proof step establishes the following lemma about bootstrapping part 2:

"After *init-kernel-instance* has initialised the kernel state of an instance,
the memory region covered by all shared frames exactly covers the shared memory region
and all non-shared frames reside in the available or device memory region."

Using this lemma, the second and final proof step over bootstrapping part 1 establishes the virtualisation theorem by proving that for all instances, the assigned shared memory region is the same, and the available and device memory regions are disjoint between instances.

Proof Size The Isabelle proofs of both theorems consist mostly of Hoare triples. In total, the proofs comprise 2000 LOC for part 1 of the bootstrapping phase and 3500 LOC for part 2.

6 Lifting seL4's Uniprocessor Proof Base

There are two reasons for the need to formally connect the abstract specification of seL4's bootstrapping phase to the abstract specification of the runtime phase (which was written in the L4.verified project):

1. L4.verified already proved a theorem about the abstract specification saying that the runtime phase preserves the following property if the bootstrapping phase establishes it: "The memory region covered by all kernel objects and capabilities in the system lies in any chosen region *R*." Formally connecting the two phases now enables to lift this theorem into my overall argument.
2. It enables to turn the top-level refinement theorem of uniprocessor seL4 into a refinement of any instance of seL4 running in the static setup of the multikernel design by only modifying a few dozen LOC out of a total of 200 kLOC of refinement proof.

Connecting Kernel States Connecting the two abstract specifications essentially means connecting two slightly different formalisations of the same data structures. This is done by a *state relation* of type $(\textit{upks-t} \times \textit{state-t})\ \textit{set}$ with *state-t* being the kernel state of seL4's ARM uniprocessor version used in L4.verified. The state relation comprises 150 LOC and is mostly straight-forward in connecting identical data structures that are formalised using different names or different underlying types (e.g. *nat* vs. *word*). With a state relation, I am also able to deal with the ARM vs. IA-32 discrepancy: Even though data structures used by these architectures are quite different on byte level, they are very similar, if not identical, on the abstract level[10].

Refinement Theorem The top-level refinement theorem used for uniprocessor seL4 and described in [4] can be summarised as follows. The *program* to be refined (seL4) is a **record** with 3 functions:

1. $\textit{Init} :: \textit{e-state-t} \Rightarrow \textit{i-state-t set}$, an initialisation function which takes an observable external state *e-state-t* and returns a set (to allow for non-determinism) of internal states *i-state-t*.
2. $\textit{Step} :: \textit{event-t} \Rightarrow (\textit{i-state-t} \times \textit{i-state-t})\ \textit{set}$, which reacts to an input *event-t* and (potentially non-deterministically) transforms the internal state accordingly.

[10]For example, page tables may have a different number of entries, a different size and different encoding, but on both architectures, we have a two-level page-table lookup mechanism that translates 32-bit virtual addresses to 32-bit physical addresses.

3. *Fin* :: *i-state-t* $\Rightarrow$ *e-state-t* reconstructs external states from internal states.

The execution of a program starts from an external state *e-state*, steps through a sequence of input and results in a set of external states. The program *C refines A* ($C \sqsubseteq A$) if, with the same initial state and input events, execution of *C* returns a subset of the external states returned by executing *A*.

In the case of seL4's refinement, the *Step* function models the runtime phase while the *Init* function models the whole bootstrapping. *Init* ignores its parameter and returns the kernel states the monadic uniprocessor bootstrapping function *init-kernel* returns. As mentioned earlier, L4.verified excluded the bootstrapping phase from the refinement proof. They axiomatised that the *Init* function of kernel *C* *forward-simulates* the one of kernel *A* and proved that the *Step* function of kernel *C* forward-simulates the one of kernel *A* in order to prove overall refinement.

As a consequence of this design, the changes needed are restricted to the *Init* function which I parametrise with an instance ID *inst-id* and define as:

$$\textit{Init inst-id e-state} \equiv \textit{rel-upks-state} \text{ `` } (\textit{boot-kernel-get-upks inst-id})$$

The term *r* `` *S* denotes the image of set *S* under relation *r*, *rel-upks-state* is the state relation described above, and *boot-kernel-get-upks* is a function that takes an *inst-id*, calls *boot-kernel* and, from the returned set of *mpms-t* monad states, extracts the *upks-t* that was created for the instance with ID *inst-id*. The new top-level refinement theorem says that kernel instance C with ID *inst-id* refines kernel instance A with ID *inst-id*:

$$\textbf{theorem } C \textit{ inst-id} \sqsubseteq A \textit{ inst-id}$$

7 Related Work

The multikernel design was also chosen by Barrelfish [1] and Corey [2]. Barrelfish is an OS aimed to be highly scalable and suitable for heterogeneous multiprocessing. It follows a distributed-system approach by keeping kernel data structures local to a CPU or replicated on other CPUs. Synchronisation and coordination between CPUs is message-based and managed by the system software running on top. The API of the underlying microkernel is inspired by seL4. Corey is an OS with almost the same aims as Barrelfish. Kernel data is CPU-local too, but system software is allowed to choose which kernel data should be shared between CPUs. Neither of these projects address security or formal verification.

The *VFiasco* project and its successor *Robin* [11] aimed to verify the *Fiasco* microkernel and its successor *NOVA* directly on the level of C++ using PVS. They developed a precise model of a large subset of C++ deemed necessary but did not verify substantial parts of these microkernels. Shapiro et al. [10] specifically designed the new programming language *BitC* in order to implement and formally verify their *Coyotos* microkernel. BitC is tightly integrated into the verification framework and allows breaking type-safety in a controlled way. Neither one of these projects addresses multiprocessing.

The *Verisoft* project achieved considerable success in verifying a whole software stack from hardware up to the application including a non-optimising compiler and verification framework for their implementation language which allows to reason about concurrent processes [6] running on top of their *VAMOS* microkernel. The processes rely on synchronisation primitives provided by VAMOS which is a uniprocessor microkernel. Microsoft's *VCC* verification environment [5] allows to reason about concurrent system-level C code. Proofs are guided by creating C annotations such that the generated proof obligations can be discharged automatically. Thread-local data can be reasoned about in a sequential context using an ownership discipline while concurrent data structures (e.g. locks) are handled separately. The *Verisoft XT* project [12] used VCC to formally verify substantial parts of Microsoft's *Hyper-V* multiprocessor hypervisor. The proved properties are mainly function contracts and invariants on data types. It is unclear if higher-level properties (e.g. isolation, information flow) have been targeted/proved.

8 Conclusion

In order to extend the high assurance of seL4's formally verified uniprocessor version to multiprocessor systems, I presented the big-lock kernel and the multikernel designs which both avoid concurrency and, at the same time, allow system software running on top (e.g. a VMM) to use the power of all CPUs.

For the multikernel design, the paper contributes a multiprocessor execution model which takes a parallel program of high-level instructions and returns all memory-access histories potentially observable when running it. The program is generated by a monadic abstract specification of seL4's bootstrapping phase which, for its own correctness, relies on observing sequential memory-access semantics.

The kernel-memory-access theorem says that the abstract specification observes sequential memory-access semantics. The virtualisation theorem says that capabilities created during bootstrapping and given to the system software's initial thread conform to seL4's multiprocessor ABI.

Finally, a state relation connects seL4's multiprocessor and uniprocessor abstract specifications to enable an overall proof showing that the validity of seL4's refinement proof has been preserved.

References

[1] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *22nd SOSP*, Big Sky, MT, USA, Oct 2009. ACM.

[2] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *8th OSDI*, San Diego, CA, USA, Dec 2008.

[3] Andrew Boyton. A verified shared capability model. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *4th SSV*, volume 254 of *ENTCS*, pages 25–44, Aachen, Germany, Oct 2009. Elsevier.

[4] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer.

[5] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 23–42, Munich, Germany, 2009. Springer.

[6] Matthias Daum, Jan Dörrenbächer, Mareike Schmidt, and Burkhart Wolff. A verification approach for system-level concurrent programs. In *VSTTE 2008*, pages 161–176, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Gernot Heiser. Hypervisors for consumer electronics. In *6th IEEE Consumer Comm. & Networking Conf.*, Las Vegas, NV, USA, Jan 2009.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[9] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA WS Operat. Syst. Verification*, Sydney, Australia, Oct 2004.

[11] Hendrik Tews, Tjark Weber, and Marcus Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *3rd SSV*, volume 217 of *ENTCS*, pages 79–96, Sydney, Australia, Feb 2008. Elsevier.

[12] Verisoft XT project. http://www.verisoftxt.de.