Preliminary Thoughts on Verifying L4-Based Operating Systems

Marcus Völp
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
voelp@os.inf.tu-dresden.de

Abstract

Microkernels and microhypervisors, which we understand to be small hypervisors providing microkernel-like abstractions to build minimal-complex and de-privileged virtual-machine monitors, have become very attractive operating-system alternatives for PC- and embedded platforms.

This paper highlights initial thoughts towards verifying microkernel-based operating systems and in particular systems running on top of the L4.Sec microkernel.

1. Introduction

Their ability to co-host legacy operating systems next to security-sensitive and real-time-sensitive applications [3, 4] increased the interest in microkernels and microhypervisors as the underlying operating system in PC- and embedded platforms.

For use in embedded environments correctness of the operating system (and observance of other relevant properties) is crucial and should at best be formally verified.

Here, unlike monolithic systems, microkernels show a great advantage: the microkernel abstractions facilitate a construction paradigm in which all functionality is split into small servers. These servers are separated into different address spaces with the right to access only those parts of other address spaces that are necessary for fulfilling their functionality and for communicating requests and results with their clients. The advantage with regards to verification that comes from this construction principle is that the code size of these individual servers is reduced to an amount that can be handled with today's verification tools.

This paper presents initial thoughts on how verification of a microkernel-based system may work. We concentrate on L4.Sec [7] — an L4-family microkernel [8]. We are however confident, that the ideas will carry on also to other second-generation microkernels and microhypervisors.

The remainder of this paper is structured as follows: Section 2 introduces the L4.Sec microkernel. In Section 3 we have a more detailed look into why microkernel-based systems are the right systems to formally verify. Section 4 highlights our initial ideas towards verifying microkernel-based operating systems. Section 5 relates our approach to other work and we conclude in Section 6.

2. L4.Sec

L4 abstracts from the underlying hardware by introducing threads, address spaces and synchronous and blocking inter-process communication (IPC) to communicate between threads in different address spaces.

In L4.Sec access to all kernel objects (i.e., threads, address spaces, user memory, communication endpoints, etc.) is controlled via capabilities. A capability restricts which system calls can be performed on the kernel object it references. We say a thread owns a capability if there is a mapping of a local name (or virtual address for memory pages) to this capability in the thread's address space.

A thread can transfer any of its owned capabilities as part of an IPC to other threads, provided a communication endpoint to which this sending thread is authorised to send and from which the receiver is authorised to receive. The access rights transferred via this map operation can later on be revoked with the unmap operation from all address spaces that directly or indirectly received the capability from a thread in the unmapper's address space.

Upon startup the microkernel constructs an initial address space σ_0 that has an idempotent (virtual to physical) mapping to all physical memory available for use by user-level servers and applications. Upon request (e.g., from the loader) it will respond with mapping the requested page, provided this page is available memory. Figure 1 shows an example of a first server running on top of σ_0.

We will use σ_0 as our guiding example.

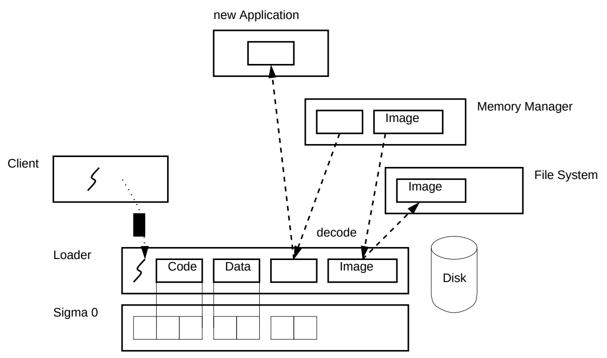

Figure 1: The correctness of σ_0 is indispensable for the loader, however, not the correctness of the memory- and file-server it uses.

3. Why Microkernel-Based Systems

In a monolithic operating system, each server has unrestricted access to the entire kernel address space, i.e., to the code and data of all other servers and applications and uncontrolled access to all datastructures used for implementing kernel objects (e.g., kernel threads). Thus a single failure (intentionally as a consequence of attacks, or unintentionally) in such a server may bring down the entire system.

In microkernel-based operating systems, these servers are separated and hold only capabilities to those kernel objects of other servers which they need to perform their functionality.

Let us consider again the example from Figure 1. The first task above σ_0 implements loader functionality and uses a file system to read the files to start. Consistency of the code and data section of the loader are indispensable for its proper functioning.

Clients typically only hold capabilities on an endpoint through which they can send messages to request the loader to start other applications. Otherwise they have no access to objects in the loader's address space.

For the actual startup, the loader needs memory in which it requests the file server to copy the image and which it must map into its address space to decode the image. For this, the file server and the memory manager (σ_0 or some higher-level memory manager) needs only to have access to this memory, but to no other objects in the loader's address space. By properly programming the loader (e.g., multi-threaded or exploiting asynchronous communication protocols to prevent blocking at the file system), it can handle client requests such, that failure in the memory server, in the file system or in clients cannot affect the requests from other client's.

Obviously, correctness of the file system and of the memory manager is indispensable for the started application, but as we have seen not for the loader.

Let V be the server we aim to verify. As the loader example illustrates we can classify the servers and applications in the environment of V into:

unrelated servers which hold no capability on a kernel object of V.

clients which hold sufficient capabilities on V to issue their requests. Servers should be implemented such that their correctness does not depend on the correctness of its clients. For the verification we will thus assume that clients will execute arbitrary code.

related servers are used by V to perform its task. Their access to the server's kernel objects is, however, restricted such that even erroneous or malicious related servers do not affect the proper functioning of V. For the verification we will also assume that related servers execute arbitrary code, however, we investigate their properties to assert towards the clients when their request is correctly handled. The assertions we aim to verify are thus typically of the form: provided correctness of related servers, the server will produce correct results; incorrect servers will affect only those requests for which these servers are indispensable.

indispensable servers hold capabilities on mission critical resources and must therefore be completely trustworthy. To verify V we first need to establish trust in all indispensable servers.

This classification and the need to establish trust only in indispensable servers facilitates a hierarchical verification of microkernel-based systems, in our example, the microkernel, sigma0, the loader, the file-system and memory manager and finally the started application or server. Each of these servers can typically be implemented in less than 5000 LOC, an amount that can be dealt with by today's verification tools.

In the next section, we investigate how such a verification may work.

4. Verifying Microkernel-Based Systems

Typically operating systems are written in C, sometimes in C++. To formally reason about servers, we need a model capturing the semantics of both the language features and the system calls of the microkernel that are used by the servers to verify. Furthermore, because these servers can, in general, be preempted at almost all points in their execution and because these servers may run on a multiprocessor system, the model must facilitate quasi- and true parallelism. In the following (Section 4.1), we sketch our co-algebraic model of the L4.Sec microkernel.

4.1. Model

We model the system state S of an L4-based operating system by three principal relations:

$spc : Phys\_Address \times Address \rightharpoonup Capability$ recording with which address an address space (physical address in the domain) can access which capability (tuples $Phys\_Address \times Rights$).

$mdb : Phys\_Address \times Address \rightharpoonup Mapping\_Node$ recording where each capability has been mapped to (the dashed arrows in the above figure).

$mem : Phys\_Address \rightharpoonup Kernel\_Object$ recording the placement and state of kernel objects. Due to the special memory management of L4.Sec, free memory can be used either as $user\_memory$ ($\subset Kernel\_Object$) or to hold the datastructures of other kernel objects (e.g., threads). $Kernel\_Object$ is an abstract datatype recording the internal state of these objects (e.g., the byte values of a user-memory page).

System calls we model (similar to [13]) as state transformers transitioning a state into a complex state: $S \rightarrow CS$ with $CS := S \times Val + S + ... + 1$. $S \times Val$ is the normal outcome, $S + ... + 1$ record various abnormal states as they occur, for example, after a break instruction to indicate that the remaining while body must be skipped. We label these states appropriately as $ok : S \times Val$, $break : S, ..., fail : 1$. *fail* indicates a model failure, i.e., an outcome which does not happen in a real system but which we use to simplify the state transformers.

4.1.1 Serialisation

To correctly model preemption and parallel execution we require that each state transformer represents an unpreemptible execution step and that state transformers are serialisable[9] [1]. That is, the parallel composition of each two state transformers $stf_1 || stf_2$ produces the same behaviour as some sequential composition: $stf_1; stf_2 \vee stf_2; stf_1$. Where system calls do not show these two properties we need to split them into multiple state transformers.

Figure 2 illustrates this for mapping a capability. The *map* operation must allocate mapping nodes which record the information necessary for a later unmap. After the allocation, however, the map operation may fail (e.g., because of an existing mapping in the destination space). In this case the node is freed and its memory becomes available again. This, however, renders a serialisation problem because a parallel object allocating function may find no free memory and fails. In a serial composition it could succeed

[1]This property is sometimes called atomic. We avoid this term because it has a different meaning in a multiprocessor context.

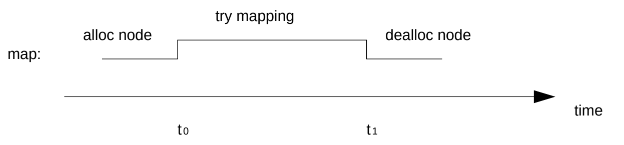

Figure 2: Serialisation of map system call: another object allocation function may fail in between t_0 and t_1 but it may succeed if executed sequentially with map.

because either the node has not yet been allocated or is freed again. To circumvent this problem we model map as three state transformers: two, allocating and deallocating mapping nodes and one performing the actual map operation.

A second serialisation problem where splitting is not feasible occurs with writing-back return values to thread control registers [2]. These registers can be implemented with user accessible memory, however, then partial write-backs would be visible. To limit the increase in complexity we instead exclude TCR memory from our memory model and introduce special state transformers that (according to the specification) allow the invoking thread only to access its own TCRs.

4.1.2 Traces

The individual programs executed we model as a relation $prg : Phys\_Address \rightarrow [S \rightarrow CS]$. For the verification of a particular server in all its possible environments, we insert the programs of the server and of its indispensable servers into the relation (via predicative subtyping). The programs of other threads including clients and related servers, we consider by quantifying over all possible programs of the remaining threads.

Similar to the approach in [6] we define a trace via a helper function $step : S \times \{0, \ldots, n-1\}^* \rightarrow CS$ that takes a state S and a sequence of thread IDs α and recursively executes the unpreemptible execution steps of the threads in α. Precisely, let t be the first thread ID in α (i.e., α is not the empty sequence and $\alpha = t \circ \alpha'$). If t is runnable in S, $step(S, t \circ \alpha')$ executes the next unpreemptible execution step e of t in S. If either t is not runnable in S or $e(S)$ fails, $step(S, t \circ \alpha')$ fails as well. Otherwise the $step$ function continues recursive execution of the remaining trace on the resulting state S' of $e(S)$ (i.e., $step(S', \alpha)$).

We will further require that $step$ executes abnormal behaviour to an end so that the result of this function yields a realistic transition if we further restrict the traces such that $step$ never $fails$.

Before exploiting these traces in the specification of indispensable servers and in the verification, we argue why all abnormal behaviour terminates and why all traces are infinite.

Abnormal behaviour may persist for the entire program for two reasons:

1. because the program is not well formed (e.g., when a break is issued outside a while body). We avoid this by quantifying only over well-formed programs or by issuing a model failure *fail* and consider for the final verification only those traces that do not result in *fail*. No such trace occurs in reality.

2. because the program misses to catch abnormal behaviour (e.g., if it fails to catch all exceptions). Such programs, however, have a well defined behaviour: for example, the thread of a program that ends with uncaught exceptions will be terminated.

Operating systems and most of its servers typically run forever and await further requests to handle. Even if all threads block, the operating system's idle thread remains runnable. Because of this, traces representing realistic scheduling behaviour will always find a runnable thread to execute. Thus realistic traces are infinite and for the verification we need only to consider these infinite traces.

4.2. Behaviour Specification

Rather than embedding the actual implementations of indispensable servers into the prg relation it is more convenient to verify against an abstract specification of the behaviour of indispensable servers. Then a positive verification result carries along also to other implementations preserving the behaviour.

Jacobs [6] defines three temporal operators for specifying binary-tree properties. In the following, we complement these with two additional temporal operators: next time P holds, Q holds as well ($\triangle(P, Q)$), and, certainly sometimes Q holds ($\otimes(Q)$).

We exemplify their use in a case study specifying the behaviour of σ_0 (Section 4.2.2).

4.2.1 Temporal Operators

Let $T \subseteq \{0, \ldots, n-1\}^*$ be the set of realistic, infinite traces. For any prefix β of a trace $\alpha \in T$ the $step(s, \beta)$ produces a realistic result state by transitioning only over runnable and existing threads.

In this setting we define in analogy to the three temporal operators next time ($\bigcirc(P)$), eventually ($\Diamond(P)$ and henceforth ($\Box(P)$) from Jacobs [6] two additional temporal operators:

certainly sometimes Q:

$$\begin{aligned}\otimes(Q)(s) \Leftrightarrow & \forall \alpha \in T . \exists \beta \in \mathit{Prefix}(\alpha). \\ & step(s, \beta) = (s', v) \Rightarrow Q(s')\end{aligned}$$

next time P also Q:

$$\begin{aligned}\triangle(P, Q)(s) \Leftrightarrow & \forall \alpha \in T . \exists \beta \in \mathit{Prefix}(\alpha). \\ & \forall \gamma \in \mathit{Prefix}(\beta). \\ & \quad step(s, \gamma) = (s', v') \Rightarrow \neg P(s') \wedge \\ & \quad step(s, \beta) = (s'', v'') \wedge P(s'') \Rightarrow \\ & \quad\; Q(s'')\end{aligned}$$

4.2.2 Case Study: σ_0

We specify the behaviour of σ_0 of mapping the requested page of physical memory when this page is available. This includes three parts:

1. σ_0 never modifies the memory that is available for their clients,

2. σ_0 receives all messages sent to it, and

3. σ_0 replies correctly to correct requests.

More formally, let $avl : Address \rightarrow bool$ be a predicate stating which memory is available for clients of σ_0. Obviously, avl depends on the implementation of σ_0 and on the underlying hardware architecture. In the verification of the loader we thus specify a similar predicate *required* and phrase the precondition *required* $\subseteq$ avl for the verification.

The first part is then:

$$\begin{aligned}\Box(\lambda s' . step(s', \sigma_0) = (s'', v) \wedge & \\ \forall a \in Address . avl(a) \Rightarrow & \\ s'.mem(a) = s''.mem(a))(s_0) &\end{aligned}$$

where s_0 is a state after σ_0 is properly initialised and $step(s', \sigma_0)$ denotes the following state when executing the next instruction of σ_0 in state s'. Note that we do not restrict write operations. Rather, we define the unchanged property in terms of the memory in the following state. This way an implementation of σ_0 could execute instructions like $or(address, 0)$, an instruction frequently used by higher level memory managers to ensure page faults are resolved prior to mapping memory as a reply to client requests.

To specify the second part we need an accessor function $next(t)$ to prg extracting the next state transformer executed by thread t. We write $s.next(t)$ to make explicit that we evaluate $next$ in state s. Furthermore we need to determine whether the server to verify contacts σ_0 or whether it invokes some other capability. We therefore introduce a predicate $\sigma_0 endpoint(p \in Physical\_Address)$ that is true if the kernel object located at address p is a communication endpoint at which σ_0 offers its interface. Specifying this endpoint as a predicate allows for σ_0 implementations offering their service at multiple communication endpoints (e.g., one endpoint per processor). Correspondingly,

$\sigma_0 endpoint(target(s.spc(space(t_i), a)))$ states whether the capability that is located in the address space of thread t_i ($space(t_i)$) at address a refers to such a σ_0 communication endpoint.

With these helper functions we write the second part as:

$$
\begin{aligned}
&step(s, t_i) = (s', v')\wedge\\
&(s.next(t_i) = ipc\_send(a) \vee s.next(t_i) = ipc\_call(a))\wedge\\
&\sigma_0 endpoint(target(s.spc(space(t_i), a))) \Rightarrow\\
&\otimes (\lambda s''.thread\_state(s''.mem(t_i)) = sending\wedge\\
&\qquad step(s'', t_i) = (s''', v''') \Rightarrow s''.next(t_i) = ipc\_xmit)(s')
\end{aligned}
$$

Where ipc_send, ipc_call are the serialisable and unpreemptible state transformers implementing the synchronous rendezvous when sending or sending and atomically receiving (call) a message respectively ipc_xmit implementing the message transmission.

In words, a message send to the σ_0 endpoint will certainly sometimes be transmitted. The fact that σ_0 causes this transition is due to an additional (not shown) requirement that no non-σ_0 thread has receive permissions on σ_0 endpoints.

To capture the third property we first define some more predicates: $valid\_request(s, t_i)$ checks the thread control registers containing the message (message registers) of t_i in state s and returns true iff the values of these registers correspond to a valid σ_0-protocol request. We also check the avl property and the reply endpoint. More formally:

$$
\begin{aligned}
&valid\_request(s, t_i) \Leftrightarrow\\
&\quad avl(s.message\_register(s.mem(t_i), 0))\wedge\\
&\quad s.message\_register(s.mem(t_i), 1) = flexpage(a, 1)\wedge\\
&\quad has\_type\_endpoint(s.mem(target(s.spc(space(t_i), a))))
\end{aligned}
$$

Flexpage is an L4 datatype encoding size aligned regions of some power of two size. Here we simply use $flexpage(a, s)$ as a constructor abstracting from the actual bit representation.

Similar we can define the predicate $valid\_reply(s, \sigma_0)$ to check whether a reply from σ_0 conforms to the protocol.

With $reply\_ep$ being the physical address of the reply endpoint associated with the request we can finally formulate the third part as:

$$
\begin{aligned}
&step(s, t_i) = (s', v')\wedge\\
&s.next(t_i) = ipc\_xmit\wedge\\
&valid\_request(s', t_i) \Rightarrow\\
&\quad \triangle(\lambda s''.step(s'', \sigma_0) = (s''', v''')\wedge\\
&\qquad s.spc(space(\sigma_0), a') = reply\_ep\wedge\\
&\qquad s''.next(\sigma_0) = ipc\_send(a'),\\
&\qquad \lambda s''''.valid\_reply(s'''', \sigma_0))(s')
\end{aligned}
$$

As already mentioned we will use such behaviour specifications in two ways:

1. as proof obligations for verifying an actual implementation (of σ_0), and

2. as behaviour specification of an indispensable server (e.g., of the loader).

5. Related Work

The authors are unaware of work addressing the verification of microkernel-based systems.

Many projects, however, are currently aiming at verifying mircokernels. Those related to the L4 microkernel are VFiasco [13], L4.verified [12] and the Robin EU-Project [1]. This tendency supports our believe that in the near future microkernel-based systems get verified. This paper presents first steps in this direction.

Co-algebraic specifications have been successfully used in verifications (e.g., Jacobs [6] in a verification of Peterson's algorithm). A good introduction into co-algebraic specification and verification we found in [5].

To integrate a language semantics with system call semantics any operational language semantics[10] describing the language operations as small serialisable steps is suitable. Norrish [11] describes such a semantics for C.

6. Conclusions

In this paper we presented initial thoughts towards the verification of microkernel-based systems.

We argued that microkernel-based system verification needs only to consider the server to be verified and those servers indispensable for the its proper functioning. We sketched a formal model suitable for this kind of verification. Related servers and clients need not to be correct and not even known precisely, however, their possible interference must be considered by quantifying over all programs that they could execute.

We exemplified how co-algebraic specification can be used to model the behaviour of indispensable servers and how to use such behaviour specifications to make verification results carry on to behaviour preserving implementations.

References

[1] Open robust infrastructures, Feb 2006. http://robin.tudos.org.

[2] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 eXperimental Kernel Reference Manual, Version X.2. Technical report, 2004. Latest version available from: `http://l4hq.org/docs/manuals/`.

[3] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 308–319, Dec. 1997.

[4] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *First International Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, California, USA, Dec. 2005.

[5] B. Jacobs. *Introduction to Coalgebra. Towards Mathematics of States and Observations*. unpublished. available at http://www.cs.ru.nl/ bart/PAPERS/index.html.

[6] B. Jacobs. Exercises in coalgebraic specification. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer Verlag, April 2000.

[7] B. Kauer and M. Völp. *L4.Sec Preliminary Microkernel Reference Manual*. Technische Universität Dresden, Oct 2005. Available at: http://os.inf.tu-dresden.de/L4/L4.Sec.

[8] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, Dec. 1995.

[9] R. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), Dec 1975.

[10] H. Nielson and F. Nielson. *Semantics with Applications - A Formal Introduction*. 1999. available at: http://www.daimi.au.dk/ hrn.

[11] M. Norrish. C formalised in hol. Technical report, University of Cambridge, 1998.

[12] G. K. R. Kolanski. Formalising the L4 microkernel API. In *The Australasian Theory Symposium (CATS 06)*, Jan 2006.

[13] H. Tews. Case study in coalgebraic specification: Memory management in the Fiasco microkernel. Technical report, TU Dresden, Inst. Theor. Informatik, Mar. 2000. Available from URL: `http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/vfiasco-report.ps.gz`.